
Algebras

Release 10.6

The Sage Development Team

Jun 27, 2025

CONTENTS

1	Catalog of Algebras	1
2	Quantum Groups	3
3	Free associative algebras and quotients	59
4	Finite dimensional algebras	121
5	Named associative algebras	151
6	Hecke algebras	803
7	Graded algebras	933
8	Various associative algebras	1007
9	Non-associative algebras	1043
10	Indices and Tables	1395
	Bibliography	1397
	Python Module Index	1399
	Index	1403

CATALOG OF ALGEBRAS

The `algebras` object may be used to access examples of various algebras currently implemented in Sage. Using tab-completion on this object is an easy way to discover and quickly create the algebras that are available (as listed here).

Let `<tab>` indicate pressing the Tab key. So begin by typing `algebras.<tab>` to see the currently implemented named algebras.

- `algebras.AlternatingCentralExtensionQuantumOnsager`
- `algebras.ArikiKoike`
- `algebras.AskeyWilson`
- `algebras.Blob`
- `algebras.Brauer`
- `algebras.Clifford`
- `algebras.ClusterAlgebra`
- `algebras.CubicHecke`
- `algebras.Descent`
- `algebras.DifferentialWeyl`
- `algebras.DownUp`
- `algebras.Exterior`
- `algebras.FiniteDimensional`
- `algebras.FQSym`
- `algebras.Free`
- `algebras.FreeZinbiel`
- `algebras.FreePreLie`
- `algebras.FreeDendriform`
- `algebras.FSym`
- `algebras.GradedCommutative`
- `algebras.Group`
- `algebras.GrossmanLarson`
- `algebras.Hall`

- *algebras.Incidence*
- *algebras.IwahoriHecke*
- *algebras.Moebius*
- *algebras.Jordan*
- *algebras.Lie*
- *algebras.MalvenutoReutenauer*
- *algebras.NilCoxeter*
- *algebras.Octonion*
- *algebras.OrlikTerao*
- *algebras.OrlikSolomon*
- *algebras.QuantumClifford*
- *algebras.QuantumGL*
- *algebras.QuantumMatrixCoordinate*
- *algebras.QuantumOscillator*
- *algebras.QSym*
- *algebras.Partition*
- *algebras.PlanarPartition*
- *algebras.qCommutingPolynomials*
- *algebras.qCommutingLaurentPolynomials*
- *algebras.QuantumGroup*
- *algebras.Quaternion*
- *algebras.RationalCherednik*
- *algebras.Schur*
- *algebras.Shuffle*
- *algebras.Steenrod*
- *algebras.TemperleyLieb*
- *algebras.Tensor*
- *algebras.WQSym*
- *algebras.Yangian*
- *algebras.YokonumaHecke*

QUANTUM GROUPS

2.1 Alternating Central Extension Quantum Onsager Algebra

AUTHORS:

- Travis Scrimshaw (2021-03): Initial version

class sage.algebras.quantum_groups.ace_quantum_onsager.ACEQuantumOnsagerAlgebra(R, q)

Bases: CombinatorialFreeModule

The alternating central extension of the q -Onsager algebra.

The *alternating central extension* $\mathcal{A}_q$ of the q -Onsager algebra O_q is a current algebra of O_q introduced by Baseilhac and Koizumi [BK2005]. A presentation was given by Baseilhac and Shigechi [BS2010], which was then reformulated in terms of currents in [Ter2021] and then used to prove that the generators form a PBW basis.

Note

This is only for the q -Onsager algebra with parameter $c = q^{-1}(q - q^{-1})^2$.

EXAMPLES:

```
sage: A = algebras.AlternatingCentralExtensionQuantumOnsager(QQ)
sage: AG = A.algebra_generators()
```

```
>>> from sage.all import *
>>> A = algebras.AlternatingCentralExtensionQuantumOnsager(QQ)
>>> AG = A.algebra_generators()
```

We construct the generators $\mathcal{G}_3$, $\mathcal{W}_{-5}$, $\mathcal{W}_2$, and $\tilde{\mathcal{G}}_4$ and perform some computations:

```
sage: G3 = AG[0,3]
sage: Wm5 = AG[1,-5]
sage: W2 = AG[1,2]
sage: Gt4 = AG[2,4]
sage: [G3, Wm5, W2, Gt4]
[G[3], W[-5], W[2], Gt[4]]
sage: Gt4 * G3
G[3]*Gt[4] + ((-q^12+3*q^8-3*q^4+1)/q^6)*W[-6]*W[1]
+ ((-q^12+3*q^8-3*q^4+1)/q^6)*W[-5]*W[2]
+ ((q^12-3*q^8+3*q^4-1)/q^6)*W[-4]*W[1]
+ ((-q^12+3*q^8-3*q^4+1)/q^6)*W[-4]*W[3]
```

(continues on next page)

(continued from previous page)

```

+ ((-q^12+3*q^8-3*q^4+1)/q^6)*W[-3]*W[-2]
+ ((q^12-3*q^8+3*q^4-1)/q^6)*W[-3]*W[2]
+ ((q^12-3*q^8+3*q^4-1)/q^6)*W[-2]*W[5]
+ ((-q^12+3*q^8-3*q^4+1)/q^6)*W[-1]*W[4]
+ ((q^12-3*q^8+3*q^4-1)/q^6)*W[-1]*W[6]
+ ((-q^12+3*q^8-3*q^4+1)/q^6)*W[0]*W[5]
+ ((q^12-3*q^8+3*q^4-1)/q^6)*W[0]*W[7]
+ ((q^12-3*q^8+3*q^4-1)/q^6)*W[3]*W[4]
sage: Wm5 * G3
((q^2-1)/q^2)*G[1]*W[-7] + ((-q^2+1)/q^2)*G[1]*W[7]
+ ((q^2-1)/q^2)*G[2]*W[-6] + ((-q^2+1)/q^2)*G[2]*W[6] + G[3]*W[-5]
+ ((-q^2+1)/q^2)*G[6]*W[-2] + ((q^2-1)/q^2)*G[6]*W[2]
+ ((-q^2+1)/q^2)*G[7]*W[-1] + ((q^2-1)/q^2)*G[7]*W[1]
+ ((-q^2+1)/q^2)*G[8]*W[0] + ((-q^8+2*q^4-1)/q^5)*W[-8]
+ ((q^8-2*q^4+1)/q^5)*W[8]
sage: W2 * G3
(q^2-1)*G[1]*W[-2] + (-q^2+1)*G[1]*W[4] + (-q^2+1)*G[3]*W[0]
+ q^2*G[3]*W[2] + (q^2-1)*G[4]*W[1] + ((-q^8+2*q^4-1)/q^3)*W[-3]
+ ((q^8-2*q^4+1)/q^3)*W[5]
sage: W2 * Wm5
(q^4/(q^8+2*q^6-2*q^2-1))*G[1]*Gt[6] + (-q^4/(q^8+2*q^6-2*q^2-1))*G[6]*Gt[1]
+ W[-5]*W[2] + (q/(q^2+1))*G[7] + (-q/(q^2+1))*Gt[7]
sage: Gt4 * Wm5
((q^2-1)/q^2)*W[-8]*Gt[1] + ((q^2-1)/q^2)*W[-7]*Gt[2]
+ ((q^2-1)/q^2)*W[-6]*Gt[3] + W[-5]*Gt[4] + ((-q^2+1)/q^2)*W[-3]*Gt[6]
+ ((-q^2+1)/q^2)*W[-2]*Gt[7] + ((-q^2+1)/q^2)*W[-1]*Gt[8]
+ ((-q^2+1)/q^2)*W[0]*Gt[9] + ((q^2-1)/q^2)*W[1]*Gt[8]
+ ((q^2-1)/q^2)*W[2]*Gt[7] + ((q^2-1)/q^2)*W[3]*Gt[6]
+ ((-q^2+1)/q^2)*W[6]*Gt[3] + ((-q^2+1)/q^2)*W[7]*Gt[2]
+ ((-q^2+1)/q^2)*W[8]*Gt[1] + ((-q^8+2*q^4-1)/q^5)*W[-9]
+ ((q^8-2*q^4+1)/q^5)*W[9]
sage: Gt4 * W2
(q^2-1)*W[-3]*Gt[1] + (-q^2+1)*W[0]*Gt[4] + (q^2-1)*W[1]*Gt[5]
+ q^2*W[2]*Gt[4] + (-q^2+1)*W[5]*Gt[1] + ((-q^8+2*q^4-1)/q^3)*W[-4]
+ ((q^8-2*q^4+1)/q^3)*W[6]

```

```

>>> from sage.all import *
>>> G3 = AG(Integer(0), Integer(3))
>>> Wm5 = AG(Integer(1), -Integer(5))
>>> W2 = AG(Integer(1), Integer(2))
>>> Gt4 = AG(Integer(2), Integer(4))
>>> [G3, Wm5, W2, Gt4]
[G[3], W[-5], W[2], Gt[4]]
>>> Gt4 * G3
G[3]*Gt[4] + ((-q^12+3*q^8-3*q^4+1)/q^6)*W[-6]*W[1]
+ ((-q^12+3*q^8-3*q^4+1)/q^6)*W[-5]*W[2]
+ ((q^12-3*q^8+3*q^4-1)/q^6)*W[-4]*W[1]
+ ((-q^12+3*q^8-3*q^4+1)/q^6)*W[-4]*W[3]
+ ((-q^12+3*q^8-3*q^4+1)/q^6)*W[-3]*W[-2]
+ ((q^12-3*q^8+3*q^4-1)/q^6)*W[-3]*W[2]
+ ((q^12-3*q^8+3*q^4-1)/q^6)*W[-2]*W[5]
+ ((-q^12+3*q^8-3*q^4+1)/q^6)*W[-1]*W[4]

```

(continues on next page)

(continued from previous page)

```

+ ((q^12-3*q^8+3*q^4-1)/q^6)*W[-1]*W[6]
+ ((-q^12+3*q^8-3*q^4+1)/q^6)*W[0]*W[5]
+ ((q^12-3*q^8+3*q^4-1)/q^6)*W[0]*W[7]
+ ((q^12-3*q^8+3*q^4-1)/q^6)*W[3]*W[4]
>>> Wm5 * G3
((q^2-1)/q^2)*G[1]*W[-7] + ((-q^2+1)/q^2)*G[1]*W[7]
+ ((q^2-1)/q^2)*G[2]*W[-6] + ((-q^2+1)/q^2)*G[2]*W[6] + G[3]*W[-5]
+ ((-q^2+1)/q^2)*G[6]*W[-2] + ((q^2-1)/q^2)*G[6]*W[2]
+ ((-q^2+1)/q^2)*G[7]*W[-1] + ((q^2-1)/q^2)*G[7]*W[1]
+ ((-q^2+1)/q^2)*G[8]*W[0] + ((-q^8+2*q^4-1)/q^5)*W[-8]
+ ((q^8-2*q^4+1)/q^5)*W[8]
>>> W2 * G3
(q^2-1)*G[1]*W[-2] + (-q^2+1)*G[1]*W[4] + (-q^2+1)*G[3]*W[0]
+ q^2*G[3]*W[2] + (q^2-1)*G[4]*W[1] + ((-q^8+2*q^4-1)/q^3)*W[-3]
+ ((q^8-2*q^4+1)/q^3)*W[5]
>>> W2 * Wm5
(q^4/(q^8+2*q^6-2*q^2-1))*G[1]*Gt[6] + (-q^4/(q^8+2*q^6-2*q^2-1))*G[6]*Gt[1]
+ W[-5]*W[2] + (q/(q^2+1))*G[7] + (-q/(q^2+1))*Gt[7]
>>> Gt4 * Wm5
((q^2-1)/q^2)*W[-8]*Gt[1] + ((q^2-1)/q^2)*W[-7]*Gt[2]
+ ((q^2-1)/q^2)*W[-6]*Gt[3] + W[-5]*Gt[4] + ((-q^2+1)/q^2)*W[-3]*Gt[6]
+ ((-q^2+1)/q^2)*W[-2]*Gt[7] + ((-q^2+1)/q^2)*W[-1]*Gt[8]
+ ((-q^2+1)/q^2)*W[0]*Gt[9] + ((q^2-1)/q^2)*W[1]*Gt[8]
+ ((q^2-1)/q^2)*W[2]*Gt[7] + ((q^2-1)/q^2)*W[3]*Gt[6]
+ ((-q^2+1)/q^2)*W[6]*Gt[3] + ((-q^2+1)/q^2)*W[7]*Gt[2]
+ ((-q^2+1)/q^2)*W[8]*Gt[1] + ((-q^8+2*q^4-1)/q^5)*W[-9]
+ ((q^8-2*q^4+1)/q^5)*W[9]
>>> Gt4 * W2
(q^2-1)*W[-3]*Gt[1] + (-q^2+1)*W[0]*Gt[4] + (q^2-1)*W[1]*Gt[5]
+ q^2*W[2]*Gt[4] + (-q^2+1)*W[5]*Gt[1] + ((-q^8+2*q^4-1)/q^3)*W[-4]
+ ((q^8-2*q^4+1)/q^3)*W[6]

```

REFERENCES:

- [BK2005]
- [BS2010]
- [Ter2021]

algebra_generators()

Return the algebra generators of `self`.

EXAMPLES:

```

sage: A = algebras.AlternatingCentralExtensionQuantumOnsager(QQ)
sage: A.algebra_generators()
Lazy family (generator map(i))_{i in Disjoint union of
Family (Positive integers, Integer Ring, Positive integers)}

```

```

>>> from sage.all import *
>>> A = algebras.AlternatingCentralExtensionQuantumOnsager(QQ)
>>> A.algebra_generators()
Lazy family (generator map(i))_{i in Disjoint union of

```

(continues on next page)

(continued from previous page)

```
Family (Positive integers, Integer Ring, Positive integers)
```

dagger()The antiautomorphism $\dagger$.

EXAMPLES:

```
sage: A = algebras.AlternatingCentralExtensionQuantumOnsager(QQ)
sage: G = A.algebra_generators()
sage: x = A.an_element()^2
sage: A.dagger(A.dagger(x)) == x
True
sage: A.dagger(G[1,-1] * G[1,1]) == A.dagger(G[1,1]) * A.dagger(G[1,-1])
True
sage: A.dagger(G[0,2] * G[1,3]) == A.dagger(G[1,3]) * A.dagger(G[0,2])
True
sage: A.dagger(G[2,2] * G[1,3]) == A.dagger(G[1,3]) * A.dagger(G[2,2])
True
```

```
>>> from sage.all import *
>>> A = algebras.AlternatingCentralExtensionQuantumOnsager(QQ)
>>> G = A.algebra_generators()
>>> x = A.an_element()**Integer(2)
>>> A.dagger(A.dagger(x)) == x
True
>>> A.dagger(G[Integer(1),-Integer(1)] * G[Integer(1),Integer(1)]) == A.
↪dagger(G[Integer(1),Integer(1)]) * A.dagger(G[Integer(1),-Integer(1)])
True
>>> A.dagger(G[Integer(0),Integer(2)] * G[Integer(1),Integer(3)]) == A.
↪dagger(G[Integer(1),Integer(3)]) * A.dagger(G[Integer(0),Integer(2)])
True
>>> A.dagger(G[Integer(2),Integer(2)] * G[Integer(1),Integer(3)]) == A.
↪dagger(G[Integer(1),Integer(3)]) * A.dagger(G[Integer(2),Integer(2)])
True
```

degree_on_basis(m)Return the degree of the basis element indexed by m .

EXAMPLES:

```
sage: A = algebras.AlternatingCentralExtensionQuantumOnsager(QQ)
sage: G = A.algebra_generators()
sage: A.degree_on_basis(G[0,1].leading_support())
2
sage: A.degree_on_basis(G[0,2].leading_support())
4
sage: A.degree_on_basis(G[1,-1].leading_support())
3
sage: A.degree_on_basis(G[1,0].leading_support())
1
sage: A.degree_on_basis(G[1,1].leading_support())
1
sage: A.degree_on_basis(G[2,1].leading_support())
```

(continues on next page)

(continued from previous page)

```

2
sage: A.degree_on_basis(G[2,2].leading_support())
4
sage: [x.degree() for x in A.some_elements()]
[1, 5, 3, 1, 5, 2, 4, 2, 4]

```

```

>>> from sage.all import *
>>> A = algebras.AlternatingCentralExtensionQuantumOnsager(QQ)
>>> G = A.algebra_generators()
>>> A.degree_on_basis(G[Integer(0), Integer(1)].leading_support())
2
>>> A.degree_on_basis(G[Integer(0), Integer(2)].leading_support())
4
>>> A.degree_on_basis(G[Integer(1), -Integer(1)].leading_support())
3
>>> A.degree_on_basis(G[Integer(1), Integer(0)].leading_support())
1
>>> A.degree_on_basis(G[Integer(1), Integer(1)].leading_support())
1
>>> A.degree_on_basis(G[Integer(2), Integer(1)].leading_support())
2
>>> A.degree_on_basis(G[Integer(2), Integer(2)].leading_support())
4
>>> [x.degree() for x in A.some_elements()]
[1, 5, 3, 1, 5, 2, 4, 2, 4]

```

gens()

Return the algebra generators of self.

EXAMPLES:

```

sage: A = algebras.AlternatingCentralExtensionQuantumOnsager(QQ)
sage: A.algebra_generators()
Lazy family (generator map(i))_{i in Disjoint union of
Family (Positive integers, Integer Ring, Positive integers)}

```

```

>>> from sage.all import *
>>> A = algebras.AlternatingCentralExtensionQuantumOnsager(QQ)
>>> A.algebra_generators()
Lazy family (generator map(i))_{i in Disjoint union of
Family (Positive integers, Integer Ring, Positive integers)}

```

one_basis()

Return the basis element indexing 1.

EXAMPLES:

```

sage: A = algebras.AlternatingCentralExtensionQuantumOnsager(QQ)
sage: ob = A.one_basis(); ob
1
sage: ob.parent()
Free abelian monoid indexed by Disjoint union of
Family (Positive integers, Integer Ring, Positive integers)

```

```

>>> from sage.all import *
>>> A = algebras.AlternatingCentralExtensionQuantumOnsager(QQ)
>>> ob = A.one_basis(); ob
1
>>> ob.parent()
Free abelian monoid indexed by Disjoint union of
Family (Positive integers, Integer Ring, Positive integers)

```

product_on_basis (*lhs, rhs*)

Return the product of the two basis elements *lhs* and *rhs*.

EXAMPLES:

```

sage: A = algebras.AlternatingCentralExtensionQuantumOnsager(QQ)
sage: G = A.algebra_generators()
sage: q = A.q()
sage: rho = -(q^2 - q^-2)^2

```

```

>>> from sage.all import *
>>> A = algebras.AlternatingCentralExtensionQuantumOnsager(QQ)
>>> G = A.algebra_generators()
>>> q = A.q()
>>> rho = -(q**Integer(2) - q**(-Integer(2))**Integer(2))

```

We verify the PBW ordering:

```

sage: G[0,1] * G[1,1] # indirect doctest
G[1]*W[1]
sage: G[1,1] * G[0,1]
q^2*G[1]*W[1] + ((-q^8+2*q^4-1)/q^3)*W[0] + ((q^8-2*q^4+1)/q^3)*W[2]
sage: G[1,-1] * G[1,1]
W[-1]*W[1]
sage: G[1,1] * G[1,-1]
W[-1]*W[1] + (q/(q^2+1))*G[2] + (-q/(q^2+1))*Gt[2]
sage: G[1,1] * G[2,1]
W[1]*Gt[1]
sage: G[2,1] * G[1,1]
q^2*W[1]*Gt[1] + ((-q^8+2*q^4-1)/q^3)*W[0] + ((q^8-2*q^4+1)/q^3)*W[2]
sage: G[0,1] * G[2,1]
G[1]*Gt[1]
sage: G[2,1] * G[0,1]
G[1]*Gt[1] + ((-q^12+3*q^8-3*q^4+1)/q^6)*W[-1]*W[1]
+ ((-q^12+3*q^8-3*q^4+1)/q^6)*W[0]^2
+ ((q^12-3*q^8+3*q^4-1)/q^6)*W[0]*W[2]
+ ((q^12-3*q^8+3*q^4-1)/q^6)*W[1]^2

```

```

>>> from sage.all import *
>>> G[Integer(0),Integer(1)] * G[Integer(1),Integer(1)] # indirect doctest
G[1]*W[1]
>>> G[Integer(1),Integer(1)] * G[Integer(0),Integer(1)]
q^2*G[1]*W[1] + ((-q^8+2*q^4-1)/q^3)*W[0] + ((q^8-2*q^4+1)/q^3)*W[2]
>>> G[Integer(1),-Integer(1)] * G[Integer(1),Integer(1)]
W[-1]*W[1]

```

(continues on next page)

(continued from previous page)

```

>>> G[Integer(1), Integer(1)] * G[Integer(1), -Integer(1)]
W[-1]*W[1] + (q/(q^2+1))*G[2] + (-q/(q^2+1))*Gt[2]
>>> G[Integer(1), Integer(1)] * G[Integer(2), Integer(1)]
W[1]*Gt[1]
>>> G[Integer(2), Integer(1)] * G[Integer(1), Integer(1)]
q^2*W[1]*Gt[1] + ((-q^8+2*q^4-1)/q^3)*W[0] + ((q^8-2*q^4+1)/q^3)*W[2]
>>> G[Integer(0), Integer(1)] * G[Integer(2), Integer(1)]
G[1]*Gt[1]
>>> G[Integer(2), Integer(1)] * G[Integer(0), Integer(1)]
G[1]*Gt[1] + ((-q^12+3*q^8-3*q^4+1)/q^6)*W[-1]*W[1]
+ ((-q^12+3*q^8-3*q^4+1)/q^6)*W[0]^2
+ ((q^12-3*q^8+3*q^4-1)/q^6)*W[0]*W[2]
+ ((q^12-3*q^8+3*q^4-1)/q^6)*W[1]^2

```

We verify some of the defining relations (see Equations (3-14) in [Ter2021]), which are used to construct the PBW basis:

```

sage: G[0,1] * G[0,2] == G[0,2] * G[0,1]
True
sage: G[1,-1] * G[1,-2] == G[1,-2] * G[1,-1]
True
sage: G[1,1] * G[1,2] == G[1,2] * G[1,1]
True
sage: G[2,1] * G[2,2] == G[2,2] * G[2,1]
True
sage: G[1,0] * G[1,2] - G[1,2] * G[1,0] == G[1,-1] * G[1,1] - G[1,1] * G[1,-1]
True
sage: G[1,0] * G[1,2] - G[1,2] * G[1,0] == (G[2,2] - G[0,2]) / (q + ~q)
True
sage: q * G[1,0] * G[0,2] - ~q * G[0,2] * G[1,0] == q * G[2,2] * G[1,0] - ~q *
↪ G[1,0] * G[2,2]
True
sage: q * G[1,0] * G[0,2] - ~q * G[0,2] * G[1,0] == rho * G[1,-2] - rho * G[1,
↪ 2]
True
sage: q * G[0,2] * G[1,1] - ~q * G[1,1] * G[0,2] == q * G[1,1] * G[2,2] - ~q *
↪ G[2,2] * G[1,1]
True
sage: q * G[0,2] * G[1,1] - ~q * G[1,1] * G[0,2] == rho * G[1,3] - rho * G[1,
↪ 1]
True
sage: G[1,-2] * G[1,2] - G[1,2] * G[1,-2] == G[1,-1] * G[1,3] - G[1,3] * G[1,
↪ 1]
True
sage: G[1,-2] * G[0,2] - G[0,2] * G[1,-2] == G[1,-1] * G[0,3] - G[0,3] * G[1,
↪ 1]
True
sage: G[1,1] * G[0,2] - G[0,2] * G[1,1] == G[1,2] * G[0,1] - G[0,1] * G[1,2]
True
sage: G[1,-2] * G[2,2] - G[2,2] * G[1,-2] == G[1,-1] * G[2,3] - G[2,3] * G[1,
↪ 1]
True

```

(continues on next page)

(continued from previous page)

```
sage: G[1,1] * G[2,2] - G[2,2] * G[1,1] == G[1,2] * G[2,1] - G[2,1] * G[1,2]
True
sage: G[0,1] * G[2,2] - G[2,2] * G[0,1] == G[0,2] * G[2,1] - G[2,1] * G[0,2]
True
```

```
>>> from sage.all import *
>>> G[Integer(0),Integer(1)] * G[Integer(0),Integer(2)] == G[Integer(0),
↪Integer(2)] * G[Integer(0),Integer(1)]
True
>>> G[Integer(1),-Integer(1)] * G[Integer(1),-Integer(2)] == G[Integer(1),-
↪Integer(2)] * G[Integer(1),-Integer(1)]
True
>>> G[Integer(1),Integer(1)] * G[Integer(1),Integer(2)] == G[Integer(1),
↪Integer(2)] * G[Integer(1),Integer(1)]
True
>>> G[Integer(2),Integer(1)] * G[Integer(2),Integer(2)] == G[Integer(2),
↪Integer(2)] * G[Integer(2),Integer(1)]
True
>>> G[Integer(1),Integer(0)] * G[Integer(1),Integer(2)] - G[Integer(1),
↪Integer(2)] * G[Integer(1),Integer(0)] == G[Integer(1),-Integer(1)] *
↪G[Integer(1),Integer(1)] - G[Integer(1),Integer(1)] * G[Integer(1),-
↪Integer(1)]
True
>>> G[Integer(1),Integer(0)] * G[Integer(1),Integer(2)] - G[Integer(1),
↪Integer(2)] * G[Integer(1),Integer(0)] == (G[Integer(2),Integer(2)] -
↪G[Integer(0),Integer(2)]) / (q + ~q)
True
>>> q * G[Integer(1),Integer(0)] * G[Integer(0),Integer(2)] - ~q *
↪G[Integer(0),Integer(2)] * G[Integer(1),Integer(0)] == q * G[Integer(2),
↪Integer(2)] * G[Integer(1),Integer(0)] - ~q * G[Integer(1),Integer(0)] *
↪G[Integer(2),Integer(2)]
True
>>> q * G[Integer(1),Integer(0)] * G[Integer(0),Integer(2)] - ~q *
↪G[Integer(0),Integer(2)] * G[Integer(1),Integer(0)] == rho * G[Integer(1),-
↪Integer(2)] - rho * G[Integer(1),Integer(2)]
True
>>> q * G[Integer(0),Integer(2)] * G[Integer(1),Integer(1)] - ~q *
↪G[Integer(1),Integer(1)] * G[Integer(0),Integer(2)] == q * G[Integer(1),
↪Integer(1)] * G[Integer(2),Integer(2)] - ~q * G[Integer(2),Integer(2)] *
↪G[Integer(1),Integer(1)]
True
>>> q * G[Integer(0),Integer(2)] * G[Integer(1),Integer(1)] - ~q *
↪G[Integer(1),Integer(1)] * G[Integer(0),Integer(2)] == rho * G[Integer(1),
↪Integer(3)] - rho * G[Integer(1),-Integer(1)]
True
>>> G[Integer(1),-Integer(2)] * G[Integer(1),Integer(2)] - G[Integer(1),
↪Integer(2)] * G[Integer(1),-Integer(2)] == G[Integer(1),-Integer(1)] *
↪G[Integer(1),Integer(3)] - G[Integer(1),Integer(3)] * G[Integer(1),-
↪Integer(1)]
True
>>> G[Integer(1),-Integer(2)] * G[Integer(0),Integer(2)] - G[Integer(0),
↪Integer(2)] * G[Integer(1),-Integer(2)] == G[Integer(1),-Integer(1)] *
↪
```

(continues on next page)

(continued from previous page)

```

↪G[Integer(0),Integer(3)] - G[Integer(0),Integer(3)] * G[Integer(1),-
↪Integer(1)]
True
>>> G[Integer(1),Integer(1)] * G[Integer(0),Integer(2)] - G[Integer(0),
↪Integer(2)] * G[Integer(1),Integer(1)] == G[Integer(1),Integer(2)] * ↪
↪G[Integer(0),Integer(1)] - G[Integer(0),Integer(1)] * G[Integer(1),
↪Integer(2)]
True
>>> G[Integer(1),-Integer(2)] * G[Integer(2),Integer(2)] - G[Integer(2),
↪Integer(2)] * G[Integer(1),-Integer(2)] == G[Integer(1),-Integer(1)] * ↪
↪G[Integer(2),Integer(3)] - G[Integer(2),Integer(3)] * G[Integer(1),-
↪Integer(1)]
True
>>> G[Integer(1),Integer(1)] * G[Integer(2),Integer(2)] - G[Integer(2),
↪Integer(2)] * G[Integer(1),Integer(1)] == G[Integer(1),Integer(2)] * ↪
↪G[Integer(2),Integer(1)] - G[Integer(2),Integer(1)] * G[Integer(1),
↪Integer(2)]
True
>>> G[Integer(0),Integer(1)] * G[Integer(2),Integer(2)] - G[Integer(2),
↪Integer(2)] * G[Integer(0),Integer(1)] == G[Integer(0),Integer(2)] * ↪
↪G[Integer(2),Integer(1)] - G[Integer(2),Integer(1)] * G[Integer(0),
↪Integer(2)]
True

```

q()Return the parameter q of self.

EXAMPLES:

```

sage: A = algebras.AlternatingCentralExtensionQuantumOnsager(QQ)
sage: A.q()
q

```

```

>>> from sage.all import *
>>> A = algebras.AlternatingCentralExtensionQuantumOnsager(QQ)
>>> A.q()
q

```

quantum_onsager_pbw_generator(i)Return the image of the PBW generator of the q -Onsager algebra in self.

INPUT:

- i – a pair (k, m) such that
 - $k=0$ and m is an integer
 - $k=1$ and m is a positive integer

EXAMPLES:

```

sage: A = algebras.AlternatingCentralExtensionQuantumOnsager(QQ)
sage: A.quantum_onsager_pbw_generator((0,0))
W[1]
sage: A.quantum_onsager_pbw_generator((0,1))

```

(continues on next page)

(continued from previous page)

```

(q^3/(q^4-1))*W[1]*Gt[1] - q^2*W[0] + (q^2+1)*W[2]
sage: A.quantum_onsager_pbw_generator((0,2))
(q^6/(q^8-2*q^4+1))*W[1]*Gt[1]^2 + (-q^5/(q^4-1))*W[0]*Gt[1]
+ (q^3/(q^2-1))*W[1]*Gt[2] + (q^3/(q^2-1))*W[2]*Gt[1]
+ (-q^4-q^2)*W[-1] - q^2*W[1] + (q^4+2*q^2+1)*W[3]
sage: A.quantum_onsager_pbw_generator((0,-1))
W[0]
sage: A.quantum_onsager_pbw_generator((0,-2))
(q/(q^4-1))*W[0]*Gt[1] + ((q^2+1)/q^2)*W[-1] - 1/q^2*W[1]
sage: A.quantum_onsager_pbw_generator((0,-3))
(q^2/(q^8-2*q^4+1))*W[0]*Gt[1]^2 + (1/(q^3-q))*W[-1]*Gt[1]
+ (1/(q^3-q))*W[0]*Gt[2] - (1/(q^5-q))*W[1]*Gt[1]
+ ((q^4+2*q^2+1)/q^4)*W[-2] - 1/q^2*W[0] + ((-q^2-1)/q^4)*W[2]
sage: A.quantum_onsager_pbw_generator((1,1))
((-q^2+1)/q^2)*W[0]*W[1] + (1/(q^3+q))*G[1] - (1/(q^3+q))*Gt[1]
sage: A.quantum_onsager_pbw_generator((1,2))
-1/q*W[0]*W[1]*Gt[1] + (1/(q^6+q^4-q^2-1))*G[1]*Gt[1]
+ ((-q^4+1)/q^4)*W[-1]*W[1] + (q^2-1)*W[0]^2
+ ((-q^4+1)/q^2)*W[0]*W[2] + ((q^2-1)/q^4)*W[1]^2
- (1/(q^6+q^4-q^2-1))*Gt[1]^2 + 1/q^3*G[2] - 1/q^3*Gt[2]

```

```

>>> from sage.all import *
>>> A = algebras.AlternatingCentralExtensionQuantumOnsager(QQ)
>>> A.quantum_onsager_pbw_generator((Integer(0), Integer(0)))
W[1]
>>> A.quantum_onsager_pbw_generator((Integer(0), Integer(1)))
(q^3/(q^4-1))*W[1]*Gt[1] - q^2*W[0] + (q^2+1)*W[2]
>>> A.quantum_onsager_pbw_generator((Integer(0), Integer(2)))
(q^6/(q^8-2*q^4+1))*W[1]*Gt[1]^2 + (-q^5/(q^4-1))*W[0]*Gt[1]
+ (q^3/(q^2-1))*W[1]*Gt[2] + (q^3/(q^2-1))*W[2]*Gt[1]
+ (-q^4-q^2)*W[-1] - q^2*W[1] + (q^4+2*q^2+1)*W[3]
>>> A.quantum_onsager_pbw_generator((Integer(0), -Integer(1)))
W[0]
>>> A.quantum_onsager_pbw_generator((Integer(0), -Integer(2)))
(q/(q^4-1))*W[0]*Gt[1] + ((q^2+1)/q^2)*W[-1] - 1/q^2*W[1]
>>> A.quantum_onsager_pbw_generator((Integer(0), -Integer(3)))
(q^2/(q^8-2*q^4+1))*W[0]*Gt[1]^2 + (1/(q^3-q))*W[-1]*Gt[1]
+ (1/(q^3-q))*W[0]*Gt[2] - (1/(q^5-q))*W[1]*Gt[1]
+ ((q^4+2*q^2+1)/q^4)*W[-2] - 1/q^2*W[0] + ((-q^2-1)/q^4)*W[2]
>>> A.quantum_onsager_pbw_generator((Integer(1), Integer(1)))
((-q^2+1)/q^2)*W[0]*W[1] + (1/(q^3+q))*G[1] - (1/(q^3+q))*Gt[1]
>>> A.quantum_onsager_pbw_generator((Integer(1), Integer(2)))
-1/q*W[0]*W[1]*Gt[1] + (1/(q^6+q^4-q^2-1))*G[1]*Gt[1]
+ ((-q^4+1)/q^4)*W[-1]*W[1] + (q^2-1)*W[0]^2
+ ((-q^4+1)/q^2)*W[0]*W[2] + ((q^2-1)/q^4)*W[1]^2
- (1/(q^6+q^4-q^2-1))*Gt[1]^2 + 1/q^3*G[2] - 1/q^3*Gt[2]

```

sigma()The automorphism σ .

EXAMPLES:


```

sage: A = algebras.AlternatingCentralExtensionQuantumOnsager(QQ)
sage: G = A.algebra_generators()
sage: x = A.an_element()^2
sage: A.sigma(A.sigma(x)) == x
True
sage: A.sigma(G[1,-1] * G[1,1]) == A.sigma(G[1,-1]) * A.sigma(G[1,1])
True
sage: A.sigma(G[0,2] * G[1,3]) == A.sigma(G[0,2]) * A.sigma(G[1,3])
True

```

```

>>> from sage.all import *
>>> A = algebras.AlternatingCentralExtensionQuantumOnsager(QQ)
>>> G = A.algebra_generators()
>>> x = A.an_element()**Integer(2)
>>> A.sigma(A.sigma(x)) == x
True
>>> A.sigma(G[Integer(1),-Integer(1)] * G[Integer(1),Integer(1)]) == A.
↪sigma(G[Integer(1),-Integer(1)]) * A.sigma(G[Integer(1),Integer(1)])
True
>>> A.sigma(G[Integer(0),Integer(2)] * G[Integer(1),Integer(3)]) == A.
↪sigma(G[Integer(0),Integer(2)]) * A.sigma(G[Integer(1),Integer(3)])
True

```

some_elements()

Return some elements of self.

EXAMPLES:

```

sage: A = algebras.AlternatingCentralExtensionQuantumOnsager(QQ)
sage: A.some_elements()
[W[0], W[3], W[-1], W[1], W[-2], G[1], G[2], Gt[1], Gt[2]]

```

```

>>> from sage.all import *
>>> A = algebras.AlternatingCentralExtensionQuantumOnsager(QQ)
>>> A.some_elements()
[W[0], W[3], W[-1], W[1], W[-2], G[1], G[2], Gt[1], Gt[2]]

```

2.2 Fock Space

AUTHORS:

- Travis Scrimshaw (2013-05-03): Initial version

class sage.algebras.quantum_groups.fock_space.**FockSpace**(*n, multicharge, q, base_ring*)

Bases: Parent, UniqueRepresentation

The (fermionic) Fock space of $U_q(\widehat{\mathfrak{sl}}_n)$ with multicharge $(\gamma_1, \dots, \gamma_m)$.

Fix a positive integer $n > 1$ and fix a sequence $\gamma = (\gamma_1, \dots, \gamma_m)$, where $\gamma_i \in \mathbf{Z}/n\mathbf{Z}$. (fermionic) Fock space $\mathcal{F}$ with multicharge γ is a $U_q(\widehat{\mathfrak{gl}}_n)$ -representation with a basis $\{|\lambda\rangle\}$, where λ is a partition tuple of level m . By considering $\mathcal{F}$ as a $U_q(\widehat{\mathfrak{sl}}_n)$ -representation, it is not irreducible, but the submodule generated by $|\emptyset^m\rangle$ is isomorphic to the highest weight module $V(\mu)$, where the highest weight $\mu = \sum_i \Lambda_{\gamma_i}$.

Let $R_i(\lambda)$ and $A_i(\lambda)$ be the set of removable and addable, respectively, i -cells of λ , where an i -cell is a cell of residue i (i.e., content modulo n). The action of $U_q(\widehat{\mathfrak{sl}}_n)$ is given as follows:

$$\begin{aligned} e_i|\lambda\rangle &= \sum_{c \in R_i(\lambda)} q^{M_i(\lambda, c)} |\lambda + c\rangle, \\ f_i|\lambda\rangle &= \sum_{c \in A_i(\lambda)} q^{N_i(\lambda, c)} |\lambda - c\rangle, \\ q^{h_i}|\lambda\rangle &= q^{N_i(\lambda)} |\lambda\rangle, \\ q^d|\lambda\rangle &= q^{-N^{(0)}(\lambda)} |\lambda\rangle, \end{aligned}$$

where

- $M_i(\lambda, c)$ (resp. $N_i(\lambda, c)$) is the number of removable (resp. addable) i -cells of λ below (resp. above) c minus the number of addable (resp. removable) i -cells of λ below (resp. above) c ,
- $N_i(\lambda)$ is the number of addable i -cells minus the number of removable i -cells, and
- $N^{(0)}(\lambda)$ is the total number of 0-cells of λ .

Another interpretation of Fock space is as a semi-infinite wedge product (which each factor we can think of as fermions). This allows a description of the $U_q(\widehat{\mathfrak{gl}}_n)$ action, as well as an explicit description of the bar involution. In particular, the bar involution is the unique semi-linear map satisfying

- $q \mapsto q^{-1}$,
- $\overline{|\emptyset\rangle} = |\emptyset\rangle$, and
- $\overline{f_i|\lambda\rangle} = f_i\overline{|\lambda\rangle}$.

We then define the *canonical basis* or *(lower) global crystal basis* as the unique basis of $\mathcal{F}$ such that

- $\overline{G(\lambda)} = G(\lambda)$,
- $G(\lambda) \equiv |\lambda\rangle \pmod{q\mathbf{Z}[q]}$.

It is also known that this basis is upper unitriangular with respect to dominance order and that both the natural basis and the canonical basis of $\mathcal{F}$ are $\mathbf{Z}$ -graded by $|\lambda|$. Additionally, the transition matrices $(d_{\lambda, \nu})_{\lambda, \nu \vdash n}$ given by

$$G(\nu) = \sum_{\lambda \vdash |\nu|} d_{\lambda, \nu} |\lambda\rangle$$

described the decomposition matrices of the Hecke algebras when restricting to $V(\mu)$ [Ariki1996].

To go between the canonical basis and the natural basis, for level 1 Fock space, we follow the LLT algorithm [LLT1996]. Indeed, we first construct a basis $\{A(\nu)\}$ that is an approximation to the lower global crystal basis, in the sense that it is bar-invariant, and then use Gaussian elimination to construct the lower global crystal basis. For higher level Fock space, we follow [Fayers2010], where the higher level is considered as a tensor product space of the corresponding level 1 Fock spaces.

There are three bases currently implemented:

- The natural basis: F .
- The approximation basis that comes from LLT(-type) algorithms: A .
- The lower global crystal basis: G .

Todo

- Implement the approximation and lower global crystal bases on all partition tuples.

- Implement the bar involution.
- Implement the full $U_q(\widehat{\mathfrak{gl}})$ -action.

INPUT:

- n – the value n
- multicharge – (default: $[0]$) the multicharge
- q – (optional) the parameter q
- base_ring – (optional) the base ring containing q

EXAMPLES:

We start by constructing the natural basis and doing some computations:

```
sage: Fock = FockSpace(3)
sage: F = Fock.natural()
sage: u = F.highest_weight_vector()
sage: u.f(0,2,(1,2),0)
|2, 2, 1> + q*|2, 1, 1, 1>
sage: u.f(0,2,(1,2),0,2)
|3, 2, 1> + q*|3, 1, 1, 1> + q*|2, 2, 2> + q^2*|2, 1, 1, 1, 1>
sage: x = u.f(0,2,(1,2),0,2)
sage: [x.h(i) for i in range(3)]
[q*|3, 2, 1> + q^2*|3, 1, 1, 1> + q^2*|2, 2, 2> + q^3*|2, 1, 1, 1, 1>,
 |3, 2, 1> + q*|3, 1, 1, 1> + q*|2, 2, 2> + q^2*|2, 1, 1, 1, 1>,
 |3, 2, 1> + q*|3, 1, 1, 1> + q*|2, 2, 2> + q^2*|2, 1, 1, 1, 1>]
sage: [x.h_inverse(i) for i in range(3)]
[1/q*|3, 2, 1> + |3, 1, 1, 1> + |2, 2, 2> + q*|2, 1, 1, 1, 1>,
 |3, 2, 1> + q*|3, 1, 1, 1> + q*|2, 2, 2> + q^2*|2, 1, 1, 1, 1>,
 |3, 2, 1> + q*|3, 1, 1, 1> + q*|2, 2, 2> + q^2*|2, 1, 1, 1, 1>]
sage: x.d()
1/q^2*|3, 2, 1> + 1/q*|3, 1, 1, 1> + 1/q*|2, 2, 2> + |2, 1, 1, 1, 1>
```

```
>>> from sage.all import *
>>> Fock = FockSpace(Integer(3))
>>> F = Fock.natural()
>>> u = F.highest_weight_vector()
>>> u.f(Integer(0),Integer(2),(Integer(1),Integer(2)),Integer(0))
|2, 2, 1> + q*|2, 1, 1, 1>
>>> u.f(Integer(0),Integer(2),(Integer(1),Integer(2)),Integer(0),Integer(2))
|3, 2, 1> + q*|3, 1, 1, 1> + q*|2, 2, 2> + q^2*|2, 1, 1, 1, 1>
>>> x = u.f(Integer(0),Integer(2),(Integer(1),Integer(2)),Integer(0),Integer(2))
>>> [x.h(i) for i in range(Integer(3))]
[q*|3, 2, 1> + q^2*|3, 1, 1, 1> + q^2*|2, 2, 2> + q^3*|2, 1, 1, 1, 1>,
 |3, 2, 1> + q*|3, 1, 1, 1> + q*|2, 2, 2> + q^2*|2, 1, 1, 1, 1>,
 |3, 2, 1> + q*|3, 1, 1, 1> + q*|2, 2, 2> + q^2*|2, 1, 1, 1, 1>]
>>> [x.h_inverse(i) for i in range(Integer(3))]
[1/q*|3, 2, 1> + |3, 1, 1, 1> + |2, 2, 2> + q*|2, 1, 1, 1, 1>,
 |3, 2, 1> + q*|3, 1, 1, 1> + q*|2, 2, 2> + q^2*|2, 1, 1, 1, 1>,
 |3, 2, 1> + q*|3, 1, 1, 1> + q*|2, 2, 2> + q^2*|2, 1, 1, 1, 1>]
>>> x.d()
1/q^2*|3, 2, 1> + 1/q*|3, 1, 1, 1> + 1/q*|2, 2, 2> + |2, 1, 1, 1, 1>
```

Next, we construct the approximation and lower global crystal bases and convert to the natural basis:

```
sage: A = Fock.A()
sage: G = Fock.G()
sage: F(A[4,2,2,1])
|4, 2, 2, 1> + q*|4, 2, 1, 1, 1>
sage: F(G[4,2,2,1])
|4, 2, 2, 1> + q*|4, 2, 1, 1, 1>
sage: F(A[7,3,2,1,1])
|7, 3, 2, 1, 1> + q*|7, 2, 2, 2, 1> + q^2*|7, 2, 2, 1, 1, 1>
+ q*|6, 3, 3, 1, 1> + q^2*|6, 2, 2, 2, 2> + q^3*|6, 2, 2, 1, 1, 1, 1>
+ q*|5, 5, 2, 1, 1> + q^2*|5, 4, 3, 1, 1> + (q^2+1)*|4, 4, 3, 2, 1>
+ (q^3+q)*|4, 4, 3, 1, 1, 1> + (q^3+q)*|4, 4, 2, 2, 2>
+ (q^4+q^2)*|4, 4, 2, 1, 1, 1, 1> + q*|4, 3, 3, 3, 1>
+ q^2*|4, 3, 2, 1, 1, 1, 1> + q^2*|4, 2, 2, 2, 2, 2>
+ q^3*|4, 2, 2, 2, 1, 1, 1, 1> + q^2*|3, 3, 3, 3, 2>
+ q^3*|3, 3, 3, 1, 1, 1, 1, 1> + q^3*|3, 2, 2, 2, 2, 2, 1>
+ q^4*|3, 2, 2, 2, 2, 1, 1, 1>
sage: F(G[7,3,2,1,1])
|7, 3, 2, 1, 1> + q*|7, 2, 2, 2, 1> + q^2*|7, 2, 2, 1, 1, 1>
+ q*|6, 3, 3, 1, 1> + q^2*|6, 2, 2, 2, 2>
+ q^3*|6, 2, 2, 1, 1, 1, 1> + q*|5, 5, 2, 1, 1>
+ q^2*|5, 4, 3, 1, 1> + q^2*|4, 4, 3, 2, 1>
+ q^3*|4, 4, 3, 1, 1, 1> + q^3*|4, 4, 2, 2, 2>
+ q^4*|4, 4, 2, 1, 1, 1, 1>
sage: A(F(G[7,3,2,1,1]))
A[7, 3, 2, 1, 1] - A[4, 4, 3, 2, 1]
sage: G(F(A[7,3,2,1,1]))
G[7, 3, 2, 1, 1] + G[4, 4, 3, 2, 1]
sage: A(F(G[8,4,3,2,2,1]))
A[8, 4, 3, 2, 2, 1] - A[6, 4, 4, 2, 2, 1, 1] - A[5, 5, 4, 3, 2, 1]
+ ((-q^2-1)/q)*A[5, 4, 4, 3, 2, 1, 1]
sage: G(F(A[8,4,3,2,2,1]))
G[8, 4, 3, 2, 2, 1] + G[6, 4, 4, 2, 2, 1, 1] + G[5, 5, 4, 3, 2, 1]
+ ((q^2+1)/q)*G[5, 4, 4, 3, 2, 1, 1]
```

```
>>> from sage.all import *
>>> A = Fock.A()
>>> G = Fock.G()
>>> F(A[Integer(4),Integer(2),Integer(2),Integer(1)])
|4, 2, 2, 1> + q*|4, 2, 1, 1, 1>
>>> F(G[Integer(4),Integer(2),Integer(2),Integer(1)])
|4, 2, 2, 1> + q*|4, 2, 1, 1, 1>
>>> F(A[Integer(7),Integer(3),Integer(2),Integer(1),Integer(1)])
|7, 3, 2, 1, 1> + q*|7, 2, 2, 2, 1> + q^2*|7, 2, 2, 1, 1, 1>
+ q*|6, 3, 3, 1, 1> + q^2*|6, 2, 2, 2, 2> + q^3*|6, 2, 2, 1, 1, 1, 1>
+ q*|5, 5, 2, 1, 1> + q^2*|5, 4, 3, 1, 1> + (q^2+1)*|4, 4, 3, 2, 1>
+ (q^3+q)*|4, 4, 3, 1, 1, 1> + (q^3+q)*|4, 4, 2, 2, 2>
+ (q^4+q^2)*|4, 4, 2, 1, 1, 1, 1> + q*|4, 3, 3, 3, 1>
+ q^2*|4, 3, 2, 1, 1, 1, 1> + q^2*|4, 2, 2, 2, 2, 2>
+ q^3*|4, 2, 2, 2, 1, 1, 1, 1> + q^2*|3, 3, 3, 3, 2>
+ q^3*|3, 3, 3, 1, 1, 1, 1, 1> + q^3*|3, 2, 2, 2, 2, 2, 1>
+ q^4*|3, 2, 2, 2, 2, 1, 1, 1>
```

(continues on next page)

(continued from previous page)

```

>>> F(G[Integer(7),Integer(3),Integer(2),Integer(1),Integer(1)])
|7, 3, 2, 1, 1> + q*|7, 2, 2, 2, 1> + q^2*|7, 2, 2, 1, 1>
+ q*|6, 3, 3, 1, 1> + q^2*|6, 2, 2, 2, 2>
+ q^3*|6, 2, 2, 1, 1, 1> + q*|5, 5, 2, 1, 1>
+ q^2*|5, 4, 3, 1, 1> + q^2*|4, 4, 3, 2, 1>
+ q^3*|4, 4, 3, 1, 1, 1> + q^3*|4, 4, 2, 2, 2>
+ q^4*|4, 4, 2, 1, 1, 1, 1>
>>> A(F(G[Integer(7),Integer(3),Integer(2),Integer(1),Integer(1)]))
A[7, 3, 2, 1, 1] - A[4, 4, 3, 2, 1]
>>> G(F(A[Integer(7),Integer(3),Integer(2),Integer(1),Integer(1)]))
G[7, 3, 2, 1, 1] + G[4, 4, 3, 2, 1]
>>> A(F(G[Integer(8),Integer(4),Integer(3),Integer(2),Integer(2),Integer(1)]))
A[8, 4, 3, 2, 2, 1] - A[6, 4, 4, 2, 2, 1, 1] - A[5, 5, 4, 3, 2, 1]
+ ((-q^2-1)/q)*A[5, 4, 4, 3, 2, 1, 1]
>>> G(F(A[Integer(8),Integer(4),Integer(3),Integer(2),Integer(2),Integer(1)]))
G[8, 4, 3, 2, 2, 1] + G[6, 4, 4, 2, 2, 1, 1] + G[5, 5, 4, 3, 2, 1]
+ ((q^2+1)/q)*G[5, 4, 4, 3, 2, 1, 1]

```

We can also construct higher level Fock spaces and perform similar computations:

```

sage: Fock = FockSpace(3, [1,0])
sage: F = Fock.natural()
sage: A = Fock.A()
sage: G = Fock.G()
sage: F(G[[2,1],[4,1,1]])
|2, 1], [4, 1, 1]> + q*|2, 1], [3, 2, 1]>
+ q^2*|2, 1], [3, 1, 1, 1]> + q^2*|2], [4, 2, 1]>
+ q^3*|2], [4, 1, 1, 1]> + q^4*|2], [3, 2, 1, 1]>
+ q*|[1, 1, 1], [4, 1, 1]> + q^2*|[1, 1, 1], [3, 2, 1]>
+ q^3*|[1, 1, 1], [3, 1, 1, 1]> + q^2*|[1, 1], [3, 2, 2]>
+ q^3*|[1, 1], [3, 1, 1, 1, 1]> + q^3*|[1], [4, 2, 2]>
+ q^4*|[1], [4, 1, 1, 1, 1]> + q^4*|[1], [3, 2, 2, 1]>
+ q^5*|[1], [3, 2, 1, 1, 1]>
sage: A(F(G[[2,1],[4,1,1]]))
A([2, 1], [4, 1, 1]) - A([2], [4, 2, 1])
sage: G(F(A[[2,1],[4,1,1]]))
G([2, 1], [4, 1, 1]) + G([2], [4, 2, 1])

```

```

>>> from sage.all import *
>>> Fock = FockSpace(Integer(3), [Integer(1),Integer(0)])
>>> F = Fock.natural()
>>> A = Fock.A()
>>> G = Fock.G()
>>> F(G[[Integer(2),Integer(1)],[Integer(4),Integer(1),Integer(1)]])
|2, 1], [4, 1, 1]> + q*|2, 1], [3, 2, 1]>
+ q^2*|2, 1], [3, 1, 1, 1]> + q^2*|2], [4, 2, 1]>
+ q^3*|2], [4, 1, 1, 1]> + q^4*|2], [3, 2, 1, 1]>
+ q*|[1, 1, 1], [4, 1, 1]> + q^2*|[1, 1, 1], [3, 2, 1]>
+ q^3*|[1, 1, 1], [3, 1, 1, 1]> + q^2*|[1, 1], [3, 2, 2]>
+ q^3*|[1, 1], [3, 1, 1, 1, 1]> + q^3*|[1], [4, 2, 2]>
+ q^4*|[1], [4, 1, 1, 1, 1]> + q^4*|[1], [3, 2, 2, 1]>
+ q^5*|[1], [3, 2, 1, 1, 1]>

```

(continues on next page)

(continued from previous page)

```
>>> A(F(G[[Integer(2), Integer(1)], [Integer(4), Integer(1), Integer(1)]]))
A([2, 1], [4, 1, 1]) - A([2], [4, 2, 1])
>>> G(F(A[[Integer(2), Integer(1)], [Integer(4), Integer(1), Integer(1)]]))
G([2, 1], [4, 1, 1]) + G([2], [4, 2, 1])
```

For level 0, the truncated Fock space of [GW1999] is implemented. This can be used to improve the speed of the computation of the lower global crystal basis, provided the truncation is not too small:

```
sage: FS = FockSpace(2)
sage: F = FS.natural()
sage: G = FS.G()
sage: FS3 = FockSpace(2, truncated=3)
sage: F3 = FS3.natural()
sage: G3 = FS3.G()
sage: F(G[6, 2, 1])
|6, 2, 1> + q*|5, 3, 1> + q^2*|5, 2, 2> + q^3*|5, 2, 1, 1>
+ q*|4, 2, 1, 1, 1> + q^2*|3, 3, 1, 1, 1> + q^3*|3, 2, 2, 1, 1>
+ q^4*|3, 2, 1, 1, 1, 1>
sage: F3(G3[6, 2, 1])
|6, 2, 1> + q*|5, 3, 1> + q^2*|5, 2, 2>
sage: FS5 = FockSpace(2, truncated=5)
sage: F5 = FS5.natural()
sage: G5 = FS5.G()
sage: F5(G5[6, 2, 1])
|6, 2, 1> + q*|5, 3, 1> + q^2*|5, 2, 2> + q^3*|5, 2, 1, 1>
+ q*|4, 2, 1, 1, 1> + q^2*|3, 3, 1, 1, 1> + q^3*|3, 2, 2, 1, 1>
```

```
>>> from sage.all import *
>>> FS = FockSpace(Integer(2))
>>> F = FS.natural()
>>> G = FS.G()
>>> FS3 = FockSpace(Integer(2), truncated=Integer(3))
>>> F3 = FS3.natural()
>>> G3 = FS3.G()
>>> F(G[Integer(6), Integer(2), Integer(1)])
|6, 2, 1> + q*|5, 3, 1> + q^2*|5, 2, 2> + q^3*|5, 2, 1, 1>
+ q*|4, 2, 1, 1, 1> + q^2*|3, 3, 1, 1, 1> + q^3*|3, 2, 2, 1, 1>
+ q^4*|3, 2, 1, 1, 1, 1>
>>> F3(G3[Integer(6), Integer(2), Integer(1)])
|6, 2, 1> + q*|5, 3, 1> + q^2*|5, 2, 2>
>>> FS5 = FockSpace(Integer(2), truncated=Integer(5))
>>> F5 = FS5.natural()
>>> G5 = FS5.G()
>>> F5(G5[Integer(6), Integer(2), Integer(1)])
|6, 2, 1> + q*|5, 3, 1> + q^2*|5, 2, 2> + q^3*|5, 2, 1, 1>
+ q*|4, 2, 1, 1, 1> + q^2*|3, 3, 1, 1, 1> + q^3*|3, 2, 2, 1, 1>
```

REFERENCES:

- [Ariki1996]
- [LLT1996]
- [Fayers2010]

- [GW1999]

class `A(F)`

Bases: `CombinatorialFreeModule`, `BindableClass`

The A basis of the Fock space which is the approximation of the lower global crystal basis.

The approximation basis A is a basis that is constructed from the highest weight element by applying divided difference operators using the ladder construction of [LLT1996] and [GW1999]. Thus, this basis is bar invariant and upper unitriangular (using dominance order on partitions) when expressed in the natural basis. This basis is then converted to the lower global crystal basis by using Gaussian elimination.

EXAMPLES:

We construct Example 6.5 and 6.7 in [LLT1996]:

```
sage: FS = FockSpace(2)
sage: F = FS.natural()
sage: G = FS.G()
sage: A = FS.A()
sage: F(A[5])
|5> + |3, 2> + 2*q*|3, 1, 1> + q^2*|2, 2, 1> + q^2*|1, 1, 1, 1, 1>
sage: F(A[4,1])
|4, 1> + q*|2, 1, 1, 1>
sage: F(A[3,2])
|3, 2> + q*|3, 1, 1> + q^2*|2, 2, 1>
sage: F(G[5])
|5> + q*|3, 1, 1> + q^2*|1, 1, 1, 1, 1>
```

```
>>> from sage.all import *
>>> FS = FockSpace(Integer(2))
>>> F = FS.natural()
>>> G = FS.G()
>>> A = FS.A()
>>> F(A[Integer(5)])
|5> + |3, 2> + 2*q*|3, 1, 1> + q^2*|2, 2, 1> + q^2*|1, 1, 1, 1, 1>
>>> F(A[Integer(4),Integer(1)])
|4, 1> + q*|2, 1, 1, 1>
>>> F(A[Integer(3),Integer(2)])
|3, 2> + q*|3, 1, 1> + q^2*|2, 2, 1>
>>> F(G[Integer(5)])
|5> + q*|3, 1, 1> + q^2*|1, 1, 1, 1, 1>
```

We construct the examples in Section 5.1 of [Fayers2010]:

```
sage: FS = FockSpace(2, [0, 0])
sage: F = FS.natural()
sage: A = FS.A()
sage: F(A[[2,1],[1]])
|[2, 1], [1]> + q*|[2], [2]> + q^2*|[2], [1, 1]> + q^2*|[1, 1], [2]>
+ q^3*|[1, 1], [1, 1]> + q^4*|[1], [2, 1]>
sage: F(A[[4],[ ]])
|[4], [ ]> + q*|[3, 1], [ ]> + q*|[2, 1, 1], [ ]>
+ (q^2+1)*|[2, 1], [1]> + 2*q*|[2], [2]> + 2*q^2*|[2], [1, 1]>
+ q^2*|[1, 1, 1, 1], [ ]> + 2*q^2*|[1, 1], [2]>
+ 2*q^3*|[1, 1], [1, 1]> + (q^4+q^2)*|[1], [2, 1]>
```

(continues on next page)

(continued from previous page)

```
+ q^2*|[], [4]> + q^3*|[], [3, 1]> + q^3*|[], [2, 1, 1]>
+ q^4*|[], [1, 1, 1, 1]>
```

```
>>> from sage.all import *
>>> FS = FockSpace(Integer(2), [Integer(0), Integer(0)])
>>> F = FS.natural()
>>> A = FS.A()
>>> F(A[[Integer(2), Integer(1)], [Integer(1)]]))
|2, 1], [1]> + q*|2], [2]> + q^2*|2], [1, 1]> + q^2*|[1, 1], [2]>
+ q^3*|[1, 1], [1, 1]> + q^4*|[1], [2, 1]>
>>> F(A[[Integer(4)], []]))
|[4], []> + q*|[3, 1], []> + q*|[2, 1, 1], []>
+ (q^2+1)*|[2, 1], [1]> + 2*q*|[2], [2]> + 2*q^2*|[2], [1, 1]>
+ q^2*|[1, 1, 1, 1], []> + 2*q^2*|[1, 1], [2]>
+ 2*q^3*|[1, 1], [1, 1]> + (q^4+q^2)*|[1], [2, 1]>
+ q^2*|[], [4]> + q^3*|[], [3, 1]> + q^3*|[], [2, 1, 1]>
+ q^4*|[], [1, 1, 1, 1]>
```

options = Current options for FockSpace – display: ket

class F(F)

Bases: `CombinatorialFreeModule`, `BindableClass`

The natural basis of the Fock space.

This is the basis indexed by partitions. This has an action of the quantum group $U_q(\widehat{\mathfrak{sl}}_n)$ described in [FockSpace](#).

EXAMPLES:

We construct the natural basis and perform some computations:

```
sage: F = FockSpace(4).natural()
sage: q = F.q()
sage: u = F.highest_weight_vector()
sage: u
|>
sage: u.f(0,1,2)
|3>
sage: u.f(0,1,3)
|2, 1>
sage: u.f(0,1,2,0)
0
sage: u.f(0,1,3,2)
|3, 1> + q*|2, 1, 1>
sage: u.f(0,1,2,3)
|4> + q*|3, 1>
sage: u.f(0,1,3,2,2,0)
((q^2+1)/q)*|3, 2, 1>
sage: x = (q^4 * u + u.f(0,1,3,(2,2)))
sage: x
|3, 1, 1> + q^4*|>
sage: x.f(0,1,3)
|4, 3, 1> + q*|4, 2, 1, 1> + q*|3, 3, 2>
```

(continues on next page)

(continued from previous page)

```

+ q^2*|3, 2, 2, 1> + q^4*|2, 1>
sage: x.h_inverse(2)
q^2*|3, 1, 1> + q^4*|>
sage: x.h_inverse(0)
1/q*|3, 1, 1> + q^3*|>
sage: x.d()
1/q*|3, 1, 1> + q^4*|>
sage: x.e(2)
|3, 1> + q*|2, 1, 1>

```

```

>>> from sage.all import *
>>> F = FockSpace(Integer(4)).natural()
>>> q = F.q()
>>> u = F.highest_weight_vector()
>>> u
|>
>>> u.f(Integer(0), Integer(1), Integer(2))
|3>
>>> u.f(Integer(0), Integer(1), Integer(3))
|2, 1>
>>> u.f(Integer(0), Integer(1), Integer(2), Integer(0))
0
>>> u.f(Integer(0), Integer(1), Integer(3), Integer(2))
|3, 1> + q*|2, 1, 1>
>>> u.f(Integer(0), Integer(1), Integer(2), Integer(3))
|4> + q*|3, 1>
>>> u.f(Integer(0), Integer(1), Integer(3), Integer(2), Integer(2), Integer(0))
((q^2+1)/q)*|3, 2, 1>
>>> x = (q**Integer(4) * u + u.f(Integer(0), Integer(1), Integer(3), (Integer(2),
↳ Integer(2))))
>>> x
|3, 1, 1> + q^4*|>
>>> x.f(Integer(0), Integer(1), Integer(3))
|4, 3, 1> + q*|4, 2, 1, 1> + q*|3, 3, 2>
+ q^2*|3, 2, 2, 1> + q^4*|2, 1>
>>> x.h_inverse(Integer(2))
q^2*|3, 1, 1> + q^4*|>
>>> x.h_inverse(Integer(0))
1/q*|3, 1, 1> + q^3*|>
>>> x.d()
1/q*|3, 1, 1> + q^4*|>
>>> x.e(Integer(2))
|3, 1> + q*|2, 1, 1>

```

class ElementBases: `IndexedFreeModuleElement`

An element in the Fock space.

d()Apply the action of d on self.

EXAMPLES:

```

sage: F = FockSpace(2)
sage: F.highest_weight_vector().d()
|>
sage: F[2,1,1].d()
1/q^2*|2, 1, 1>
sage: F[5,3,3,1,1,1].d()
1/q^7*|5, 3, 3, 1, 1, 1>

sage: F = FockSpace(4, [2,0,1])
sage: F.highest_weight_vector().d()
|[], [], []>
sage: F[[2,1],[1],[2]].d()
1/q*|[2, 1], [1], [2]>
sage: F[[4,2,2,1],[1],[5,2]].d()
1/q^5*|[4, 2, 2, 1], [1], [5, 2]>

```

```

>>> from sage.all import *
>>> F = FockSpace(Integer(2))
>>> F.highest_weight_vector().d()
|>
>>> F[Integer(2),Integer(1),Integer(1)].d()
1/q^2*|2, 1, 1>
>>> F[Integer(5),Integer(3),Integer(3),Integer(1),Integer(1),
↳Integer(1)].d()
1/q^7*|5, 3, 3, 1, 1, 1>

>>> F = FockSpace(Integer(4), [Integer(2),Integer(0),Integer(1)])
>>> F.highest_weight_vector().d()
|[], [], []>
>>> F[[Integer(2),Integer(1)],[Integer(1)],[Integer(2)]].d()
1/q*|[2, 1], [1], [2]>
>>> F[[Integer(4),Integer(2),Integer(2),Integer(1)],[Integer(1)],
↳[Integer(5),Integer(2)]].d()
1/q^5*|[4, 2, 2, 1], [1], [5, 2]>

```

e(*data)

Apply the action of the divided power operator $e_i^{(p)} = e_i^p / [p]_q$ on self.

INPUT:

- *data – list of indices or pairs (i, p)

EXAMPLES:

```

sage: F = FockSpace(2)
sage: F[2,1,1].e(1)
1/q*|1, 1, 1>
sage: F[2,1,1].e(0)
|2, 1>
sage: F[2,1,1].e(0).e(1)
|2> + q*|1, 1>
sage: F[2,1,1].e(0).e(1).e(1)
((q^2+1)/q)*|1>
sage: F[2,1,1].e(0).e((1, 2))
|1>

```

(continues on next page)

(continued from previous page)

```

sage: F[2,1,1].e(0, 1, 1, 1)
0
sage: F[2,1,1].e(0, (1, 3))
0
sage: F[2,1,1].e(0, (1,2), 0)
|>
sage: F[2,1,1].e(1, 0, 1, 0)
1/q*|>

sage: F = FockSpace(4, [2, 0, 1])
sage: F[[2,1],[1],[2]]
|[2, 1], [1], [2]>
sage: F[[2,1],[1],[2]].e(2)
|[2, 1], [1], [1]>
sage: F[[2,1],[1],[2]].e(1)
1/q*|[2], [1], [2]>
sage: F[[2,1],[1],[2]].e(0)
1/q*|[2, 1], [], [2]>
sage: F[[2,1],[1],[2]].e(3)
1/q^2*|[1, 1], [1], [2]>
sage: F[[2,1],[1],[2]].e(3, 2, 1)
1/q^2*|[1, 1], [1], []> + 1/q^2*|[1], [1], [1]>
sage: F[[2,1],[1],[2]].e(3, 2, 1, 0, 1, 2)
2/q^3*|[], [], []>

```

```

>>> from sage.all import *
>>> F = FockSpace(Integer(2))
>>> F[Integer(2), Integer(1), Integer(1)].e(Integer(1))
1/q*|1, 1, 1>
>>> F[Integer(2), Integer(1), Integer(1)].e(Integer(0))
|2, 1>
>>> F[Integer(2), Integer(1), Integer(1)].e(Integer(0)).e(Integer(1))
|2> + q*|1, 1>
>>> F[Integer(2), Integer(1), Integer(1)].e(Integer(0)).e(Integer(1)).
↪e(Integer(1))
((q^2+1)/q)*|1>
>>> F[Integer(2), Integer(1), Integer(1)].e(Integer(0)).e((Integer(1), ↪
↪Integer(2)))
|1>
>>> F[Integer(2), Integer(1), Integer(1)].e(Integer(0), Integer(1), ↪
↪Integer(1), Integer(1))
0
>>> F[Integer(2), Integer(1), Integer(1)].e(Integer(0), (Integer(1), ↪
↪Integer(3)))
0
>>> F[Integer(2), Integer(1), Integer(1)].e(Integer(0), (Integer(1),
↪Integer(2)), Integer(0))
|>
>>> F[Integer(2), Integer(1), Integer(1)].e(Integer(1), Integer(0), ↪
↪Integer(1), Integer(0))
1/q*|>

```

(continues on next page)

(continued from previous page)

```

>>> F = FockSpace(Integer(4), [Integer(2), Integer(0), Integer(1)])
>>> F[[Integer(2), Integer(1)], [Integer(1)], [Integer(2)]]
| [2, 1], [1], [2]>
>>> F[[Integer(2), Integer(1)], [Integer(1)], [Integer(2)]] .e(Integer(2))
| [2, 1], [1], [1]>
>>> F[[Integer(2), Integer(1)], [Integer(1)], [Integer(2)]] .e(Integer(1))
1/q*| [2], [1], [2]>
>>> F[[Integer(2), Integer(1)], [Integer(1)], [Integer(2)]] .e(Integer(0))
1/q*| [2, 1], [], [2]>
>>> F[[Integer(2), Integer(1)], [Integer(1)], [Integer(2)]] .e(Integer(3))
1/q^2*| [1, 1], [1], [2]>
>>> F[[Integer(2), Integer(1)], [Integer(1)], [Integer(2)]] .e(Integer(3), ↵
↵ Integer(2), Integer(1))
1/q^2*| [1, 1], [1], []> + 1/q^2*| [1], [1], [1]>
>>> F[[Integer(2), Integer(1)], [Integer(1)], [Integer(2)]] .e(Integer(3), ↵
↵ Integer(2), Integer(1), Integer(0), Integer(1), Integer(2))
2/q^3*| [], [], []>

```

f(*data)

Apply the action of the divided power operator $f_i^{(p)} = f_i^p / [p]_q$ on self.

INPUT:

- *data – list of indices or pairs (i, p)

EXAMPLES:

```

sage: F = FockSpace(2)
sage: mg = F.highest_weight_vector()
sage: mg.f(0)
| 1>
sage: mg.f(0).f(1)
| 2> + q*| 1, 1>
sage: mg.f(0).f(0)
0
sage: mg.f((0, 2))
0
sage: mg.f(0, 1, 1)
((q^2+1)/q)*| 2, 1>
sage: mg.f(0, (1, 2))
| 2, 1>
sage: mg.f(0, 1, 0)
| 3> + q*| 1, 1, 1>

sage: F = FockSpace(4, [2, 0, 1])
sage: mg = F.highest_weight_vector()
sage: mg.f(0)
| [], [1], []>
sage: mg.f(2)
| [1], [], []>
sage: mg.f(1)
| [], [], [1]>
sage: mg.f(1, 0)
| [], [1], [1]> + q*| [], [], [1, 1]>

```

(continues on next page)

(continued from previous page)

```

sage: mg.f(0, 1)
|[], [2], []> + q*|[], [1], [1]>
sage: mg.f(0, 1, 3)
|[], [2, 1], []> + q*|[], [1, 1], [1]>
sage: mg.f(3)
0

```

```

>>> from sage.all import *
>>> F = FockSpace(Integer(2))
>>> mg = F.highest_weight_vector()
>>> mg.f(Integer(0))
|1>
>>> mg.f(Integer(0)).f(Integer(1))
|2> + q*|1, 1>
>>> mg.f(Integer(0)).f(Integer(0))
0
>>> mg.f((Integer(0), Integer(2)))
0
>>> mg.f(Integer(0), Integer(1), Integer(1))
((q^2+1)/q)*|2, 1>
>>> mg.f(Integer(0), (Integer(1), Integer(2)))
|2, 1>
>>> mg.f(Integer(0), Integer(1), Integer(0))
|3> + q*|1, 1, 1>

>>> F = FockSpace(Integer(4), [Integer(2), Integer(0), Integer(1)])
>>> mg = F.highest_weight_vector()
>>> mg.f(Integer(0))
|[], [1], []>
>>> mg.f(Integer(2))
|[1], [], []>
>>> mg.f(Integer(1))
|[], [], [1]>
>>> mg.f(Integer(1), Integer(0))
|[], [1], [1]> + q*|[], [], [1, 1]>
>>> mg.f(Integer(0), Integer(1))
|[], [2], []> + q*|[], [1], [1]>
>>> mg.f(Integer(0), Integer(1), Integer(3))
|[], [2, 1], []> + q*|[], [1, 1], [1]>
>>> mg.f(Integer(3))
0

```

h(*data)Apply the action of h_i on self.

EXAMPLES:

```

sage: F = FockSpace(2)
sage: F[2,1,1].h(0)
q*|2, 1, 1>
sage: F[2,1,1].h(1)
|2, 1, 1>

```

(continues on next page)

(continued from previous page)

```
sage: F[2,1,1].h(0, 0)
q^2*|2, 1, 1>
```

```
sage: F = FockSpace(4, [2,0,1])
sage: elt = F[[2,1],[1],[2]]
sage: elt.h(0)
q^2*|[2, 1], [1], [2]>
sage: elt.h(1)
|[2, 1], [1], [2]>
sage: elt.h(2)
|[2, 1], [1], [2]>
sage: elt.h(3)
q*|[2, 1], [1], [2]>
```

```
>>> from sage.all import *
>>> F = FockSpace(Integer(2))
>>> F[Integer(2),Integer(1),Integer(1)].h(Integer(0))
q*|2, 1, 1>
>>> F[Integer(2),Integer(1),Integer(1)].h(Integer(1))
|2, 1, 1>
>>> F[Integer(2),Integer(1),Integer(1)].h(Integer(0), Integer(0))
q^2*|2, 1, 1>

>>> F = FockSpace(Integer(4), [Integer(2),Integer(0),Integer(1)])
>>> elt = F[[Integer(2),Integer(1)],[Integer(1)],[Integer(2)]]
>>> elt.h(Integer(0))
q^2*|[2, 1], [1], [2]>
>>> elt.h(Integer(1))
|[2, 1], [1], [2]>
>>> elt.h(Integer(2))
|[2, 1], [1], [2]>
>>> elt.h(Integer(3))
q*|[2, 1], [1], [2]>
```

h_inverse(*data)

Apply the action of h_i^{-1} on self.

EXAMPLES:

```
sage: F = FockSpace(2)
sage: F[2,1,1].h_inverse(0)
1/q*|2, 1, 1>
sage: F[2,1,1].h_inverse(1)
|2, 1, 1>
sage: F[2,1,1].h_inverse(0, 0)
1/q^2*|2, 1, 1>

sage: F = FockSpace(4, [2,0,1])
sage: elt = F[[2,1],[1],[2]]
sage: elt.h_inverse(0)
1/q^2*|[2, 1], [1], [2]>
sage: elt.h_inverse(1)
```

(continues on next page)

(continued from previous page)

```
|[2, 1], [1], [2]>
sage: elt.h_inverse(2)
|[2, 1], [1], [2]>
sage: elt.h_inverse(3)
1/q*|[2, 1], [1], [2]>
```

```
>>> from sage.all import *
>>> F = FockSpace(Integer(2))
>>> F[Integer(2), Integer(1), Integer(1)].h_inverse(Integer(0))
1/q*|[2, 1, 1>
>>> F[Integer(2), Integer(1), Integer(1)].h_inverse(Integer(1))
|[2, 1, 1>
>>> F[Integer(2), Integer(1), Integer(1)].h_inverse(Integer(0),
↳Integer(0))
1/q^2*|[2, 1, 1>

>>> F = FockSpace(Integer(4), [Integer(2), Integer(0), Integer(1)])
>>> elt = F[[Integer(2), Integer(1)], [Integer(1)], [Integer(2)]]
>>> elt.h_inverse(Integer(0))
1/q^2*|[2, 1], [1], [2]>
>>> elt.h_inverse(Integer(1))
|[2, 1], [1], [2]>
>>> elt.h_inverse(Integer(2))
|[2, 1], [1], [2]>
>>> elt.h_inverse(Integer(3))
1/q*|[2, 1], [1], [2]>
```

options = Current options for FockSpace - display: ket

class G(F)

Bases: `CombinatorialFreeModule`, `BindableClass`

The lower global crystal basis living inside of Fock space.

EXAMPLES:

We construct some of the tables/entries given in Section 10 of [LLT1996]. For $\widehat{\mathfrak{sl}}_2$:

```
sage: FS = FockSpace(2)
sage: F = FS.natural()
sage: G = FS.G()
sage: F(G[2])
|2> + q*|1, 1>
sage: F(G[3])
|3> + q*|1, 1, 1>
sage: F(G[2,1])
|2, 1>
sage: F(G[4])
|4> + q*|3, 1> + q*|2, 1, 1> + q^2*|1, 1, 1, 1>
sage: F(G[3,1])
|3, 1> + q*|2, 2> + q^2*|2, 1, 1>
sage: F(G[5])
|5> + q*|3, 1, 1> + q^2*|1, 1, 1, 1, 1>
```

(continues on next page)

(continued from previous page)

```

sage: F(G[4,2])
|4, 2> + q*|4, 1, 1> + q*|3, 3> + q^2*|3, 1, 1, 1>
+ q^2*|2, 2, 2> + q^3*|2, 2, 1, 1>
sage: F(G[4,2,1])
|4, 2, 1> + q*|3, 3, 1> + q^2*|3, 2, 2> + q^3*|3, 2, 1, 1>
sage: F(G[6,2])
|6, 2> + q*|6, 1, 1> + q*|5, 3> + q^2*|5, 1, 1, 1> + q*|4, 3, 1>
+ q^2*|4, 2, 2> + (q^3+q)*|4, 2, 1, 1> + q^2*|4, 1, 1, 1, 1>
+ q^2*|3, 3, 1, 1> + q^3*|3, 2, 2, 1> + q^3*|3, 1, 1, 1, 1, 1>
+ q^3*|2, 2, 2, 1, 1> + q^4*|2, 2, 1, 1, 1, 1>
sage: F(G[5,3,1])
|5, 3, 1> + q*|5, 2, 2> + q^2*|5, 2, 1, 1> + q*|4, 4, 1>
+ q^2*|4, 2, 1, 1, 1> + q^2*|3, 3, 3> + q^3*|3, 3, 1, 1, 1>
+ q^3*|3, 2, 2, 2> + q^4*|3, 2, 2, 1, 1>
sage: F(G[4,3,2,1])
|4, 3, 2, 1>
sage: F(G[7,2,1])
|7, 2, 1> + q*|5, 2, 1, 1, 1> + q^2*|3, 2, 1, 1, 1, 1, 1>
sage: F(G[10,1])
|10, 1> + q*|8, 1, 1, 1, 1> + q^2*|6, 1, 1, 1, 1, 1>
+ q^3*|4, 1, 1, 1, 1, 1, 1>
+ q^4*|2, 1, 1, 1, 1, 1, 1, 1>
sage: F(G[6,3,2])
|6, 3, 2> + q*|6, 3, 1, 1> + q^2*|6, 2, 2, 1> + q^3*|5, 3, 2, 1>
+ q*|4, 3, 2, 1, 1> + q^2*|4, 3, 1, 1, 1, 1>
+ q^3*|4, 2, 2, 1, 1, 1> + q^4*|3, 3, 2, 1, 1, 1>
sage: F(G[5,3,2,1])
|5, 3, 2, 1> + q*|4, 4, 2, 1> + q^2*|4, 3, 3, 1>
+ q^3*|4, 3, 2, 2> + q^4*|4, 3, 2, 1, 1>

```

```

>>> from sage.all import *
>>> FS = FockSpace(Integer(2))
>>> F = FS.natural()
>>> G = FS.G()
>>> F(G[Integer(2)])
|2> + q*|1, 1>
>>> F(G[Integer(3)])
|3> + q*|1, 1, 1>
>>> F(G[Integer(2),Integer(1)])
|2, 1>
>>> F(G[Integer(4)])
|4> + q*|3, 1> + q*|2, 1, 1> + q^2*|1, 1, 1, 1>
>>> F(G[Integer(3),Integer(1)])
|3, 1> + q*|2, 2> + q^2*|2, 1, 1>
>>> F(G[Integer(5)])
|5> + q*|3, 1, 1> + q^2*|1, 1, 1, 1, 1>
>>> F(G[Integer(4),Integer(2)])
|4, 2> + q*|4, 1, 1> + q*|3, 3> + q^2*|3, 1, 1, 1>
+ q^2*|2, 2, 2> + q^3*|2, 2, 1, 1>
>>> F(G[Integer(4),Integer(2),Integer(1)])
|4, 2, 1> + q*|3, 3, 1> + q^2*|3, 2, 2> + q^3*|3, 2, 1, 1>
>>> F(G[Integer(6),Integer(2)])

```

(continues on next page)

(continued from previous page)

```

|6, 2> + q*|6, 1, 1> + q*|5, 3> + q^2*|5, 1, 1, 1> + q*|4, 3, 1>
+ q^2*|4, 2, 2> + (q^3+q)*|4, 2, 1, 1> + q^2*|4, 1, 1, 1, 1>
+ q^2*|3, 3, 1, 1> + q^3*|3, 2, 2, 1> + q^3*|3, 1, 1, 1, 1, 1>
+ q^3*|2, 2, 2, 1, 1> + q^4*|2, 2, 1, 1, 1, 1>
>>> F(G(Integer(5),Integer(3),Integer(1)))
|5, 3, 1> + q*|5, 2, 2> + q^2*|5, 2, 1, 1> + q*|4, 4, 1>
+ q^2*|4, 2, 1, 1, 1> + q^2*|3, 3, 3> + q^3*|3, 3, 1, 1, 1>
+ q^3*|3, 2, 2, 2> + q^4*|3, 2, 2, 1, 1>
>>> F(G(Integer(4),Integer(3),Integer(2),Integer(1)))
|4, 3, 2, 1>
>>> F(G(Integer(7),Integer(2),Integer(1)))
|7, 2, 1> + q*|5, 2, 1, 1, 1> + q^2*|3, 2, 1, 1, 1, 1, 1>
>>> F(G(Integer(10),Integer(1)))
|10, 1> + q*|8, 1, 1, 1, 1> + q^2*|6, 1, 1, 1, 1, 1, 1>
+ q^3*|4, 1, 1, 1, 1, 1, 1, 1>
+ q^4*|2, 1, 1, 1, 1, 1, 1, 1, 1, 1>
>>> F(G(Integer(6),Integer(3),Integer(2)))
|6, 3, 2> + q*|6, 3, 1, 1> + q^2*|6, 2, 2, 1> + q^3*|5, 3, 2, 1>
+ q*|4, 3, 2, 1, 1> + q^2*|4, 3, 1, 1, 1, 1>
+ q^3*|4, 2, 2, 1, 1, 1> + q^4*|3, 3, 2, 1, 1, 1>
>>> F(G(Integer(5),Integer(3),Integer(2),Integer(1)))
|5, 3, 2, 1> + q*|4, 4, 2, 1> + q^2*|4, 3, 3, 1>
+ q^3*|4, 3, 2, 2> + q^4*|4, 3, 2, 1, 1>

```

For $\widehat{\mathfrak{sl}}_3$:

```

sage: FS = FockSpace(3)
sage: F = FS.natural()
sage: G = FS.G()
sage: F(G[2])
|2>
sage: F(G[1,1])
|1, 1>
sage: F(G[3])
|3> + q*|2, 1>
sage: F(G[2,1])
|2, 1> + q*|1, 1, 1>
sage: F(G[4])
|4> + q*|2, 2>
sage: F(G[3,1])
|3, 1>
sage: F(G[2,2])
|2, 2> + q*|1, 1, 1, 1>
sage: F(G[2,1,1])
|2, 1, 1>
sage: F(G[5])
|5> + q*|2, 2, 1>
sage: F(G[2,2,1])
|2, 2, 1> + q*|2, 1, 1, 1>
sage: F(G[4,1,1])
|4, 1, 1> + q*|3, 2, 1> + q^2*|3, 1, 1, 1>
sage: F(G[5,2])

```

(continues on next page)

(continued from previous page)

```

|5, 2> + q*|4, 3> + q^2*|4, 2, 1>
sage: F(G[8])
|8> + q*|5, 2, 1> + q*|3, 3, 1, 1> + q^2*|2, 2, 2, 2>
sage: F(G[7,2])
|7, 2> + q*|4, 2, 2, 1>
sage: F(G[6,2,2])
|6, 2, 2> + q*|6, 1, 1, 1, 1> + q*|4, 4, 2> + q^2*|3, 3, 2, 1, 1>

```

```

>>> from sage.all import *
>>> FS = FockSpace(Integer(3))
>>> F = FS.natural()
>>> G = FS.G()
>>> F(G[Integer(2)])
|2>
>>> F(G[Integer(1),Integer(1)])
|1, 1>
>>> F(G[Integer(3)])
|3> + q*|2, 1>
>>> F(G[Integer(2),Integer(1)])
|2, 1> + q*|1, 1, 1>
>>> F(G[Integer(4)])
|4> + q*|2, 2>
>>> F(G[Integer(3),Integer(1)])
|3, 1>
>>> F(G[Integer(2),Integer(2)])
|2, 2> + q*|1, 1, 1, 1>
>>> F(G[Integer(2),Integer(1),Integer(1)])
|2, 1, 1>
>>> F(G[Integer(5)])
|5> + q*|2, 2, 1>
>>> F(G[Integer(2),Integer(2),Integer(1)])
|2, 2, 1> + q*|2, 1, 1, 1>
>>> F(G[Integer(4),Integer(1),Integer(1)])
|4, 1, 1> + q*|3, 2, 1> + q^2*|3, 1, 1, 1>
>>> F(G[Integer(5),Integer(2)])
|5, 2> + q*|4, 3> + q^2*|4, 2, 1>
>>> F(G[Integer(8)])
|8> + q*|5, 2, 1> + q*|3, 3, 1, 1> + q^2*|2, 2, 2, 2>
>>> F(G[Integer(7),Integer(2)])
|7, 2> + q*|4, 2, 2, 1>
>>> F(G[Integer(6),Integer(2),Integer(2)])
|6, 2, 2> + q*|6, 1, 1, 1, 1> + q*|4, 4, 2> + q^2*|3, 3, 2, 1, 1>

```

For $\widehat{\mathfrak{sl}}_4$:

```

sage: FS = FockSpace(4)
sage: F = FS.natural()
sage: G = FS.G()
sage: F(G[4])
|4> + q*|3, 1>
sage: F(G[3,1])
|3, 1> + q*|2, 1, 1>

```

(continues on next page)

(continued from previous page)

```

sage: F(G[2,2])
|2, 2>
sage: F(G[2,1,1])
|2, 1, 1> + q*|1, 1, 1, 1>
sage: F(G[3,2])
|3, 2> + q*|2, 2, 1>
sage: F(G[2,2,2])
|2, 2, 2> + q*|1, 1, 1, 1, 1, 1>
sage: F(G[6,1])
|6, 1> + q*|4, 3>
sage: F(G[3,2,2,1])
|3, 2, 2, 1> + q*|3, 1, 1, 1, 1, 1> + q*|2, 2, 2, 2>
+ q^2*|2, 1, 1, 1, 1, 1, 1>
sage: F(G[7,2])
|7, 2> + q*|6, 2, 1> + q*|5, 4> + q^2*|5, 3, 1>
sage: F(G[5,2,2,1])
|5, 2, 2, 1> + q*|5, 1, 1, 1, 1, 1> + q*|4, 2, 2, 1, 1>
+ q^2*|4, 2, 1, 1, 1, 1, 1>

```

```

>>> from sage.all import *
>>> FS = FockSpace(Integer(4))
>>> F = FS.natural()
>>> G = FS.G()
>>> F(G[Integer(4)])
|4> + q*|3, 1>
>>> F(G[Integer(3),Integer(1)])
|3, 1> + q*|2, 1, 1>
>>> F(G[Integer(2),Integer(2)])
|2, 2>
>>> F(G[Integer(2),Integer(1),Integer(1)])
|2, 1, 1> + q*|1, 1, 1, 1>
>>> F(G[Integer(3),Integer(2)])
|3, 2> + q*|2, 2, 1>
>>> F(G[Integer(2),Integer(2),Integer(2)])
|2, 2, 2> + q*|1, 1, 1, 1, 1, 1>
>>> F(G[Integer(6),Integer(1)])
|6, 1> + q*|4, 3>
>>> F(G[Integer(3),Integer(2),Integer(2),Integer(1)])
|3, 2, 2, 1> + q*|3, 1, 1, 1, 1, 1> + q*|2, 2, 2, 2>
+ q^2*|2, 1, 1, 1, 1, 1, 1>
>>> F(G[Integer(7),Integer(2)])
|7, 2> + q*|6, 2, 1> + q*|5, 4> + q^2*|5, 3, 1>
>>> F(G[Integer(5),Integer(2),Integer(2),Integer(1)])
|5, 2, 2, 1> + q*|5, 1, 1, 1, 1, 1> + q*|4, 2, 2, 1, 1>
+ q^2*|4, 2, 1, 1, 1, 1, 1>

```

We construct the examples in Section 5.1 of [Fayers2010]:

```

sage: FS = FockSpace(2, [0, 0])
sage: F = FS.natural()
sage: G = FS.G()
sage: F(G[[2,1],[1]])

```

(continues on next page)

(continued from previous page)

```

| [2, 1], [1]> + q*| [2], [2]> + q^2*| [2], [1, 1]>
+ q^2*| [1, 1], [2]> + q^3*| [1, 1], [1, 1]> + q^4*| [1], [2, 1]>
sage: F(G[[4],[ ]])
| [4], [ ]> + q*| [3, 1], [ ]> + q*| [2, 1, 1], [ ]> + q^2*| [2, 1], [1]>
+ q*| [2], [2]> + q^2*| [2], [1, 1]> + q^2*| [1, 1, 1, 1], [ ]>
+ q^2*| [1, 1], [2]> + q^3*| [1, 1], [1, 1]> + q^2*| [1], [2, 1]>
+ q^2*| [ ], [4]> + q^3*| [ ], [3, 1]> + q^3*| [ ], [2, 1, 1]>
+ q^4*| [ ], [1, 1, 1, 1]>

```

```

>>> from sage.all import *
>>> FS = FockSpace(Integer(2), [Integer(0), Integer(0)])
>>> F = FS.natural()
>>> G = FS.G()
>>> F(G[[Integer(2), Integer(1)], [Integer(1)]]))
| [2, 1], [1]> + q*| [2], [2]> + q^2*| [2], [1, 1]>
+ q^2*| [1, 1], [2]> + q^3*| [1, 1], [1, 1]> + q^4*| [1], [2, 1]>
>>> F(G[[Integer(4)], [ ]])
| [4], [ ]> + q*| [3, 1], [ ]> + q*| [2, 1, 1], [ ]> + q^2*| [2, 1], [1]>
+ q*| [2], [2]> + q^2*| [2], [1, 1]> + q^2*| [1, 1, 1, 1], [ ]>
+ q^2*| [1, 1], [2]> + q^3*| [1, 1], [1, 1]> + q^2*| [1], [2, 1]>
+ q^2*| [ ], [4]> + q^3*| [ ], [3, 1]> + q^3*| [ ], [2, 1, 1]>
+ q^4*| [ ], [1, 1, 1, 1]>

```

options = Current options for FockSpace - display: ket

a_realization()

Return a realization of self.

EXAMPLES:

```

sage: FS = FockSpace(2)
sage: FS.a_realization()
Fock space of rank 2 of multicharge (0,) over
Fraction Field of Univariate Polynomial Ring in q over Integer Ring
in the natural basis

```

```

>>> from sage.all import *
>>> FS = FockSpace(Integer(2))
>>> FS.a_realization()
Fock space of rank 2 of multicharge (0,) over
Fraction Field of Univariate Polynomial Ring in q over Integer Ring
in the natural basis

```

approximation

alias of `A`

canonical

alias of `G`

highest_weight_vector()

Return the module generator of self in the natural basis.

EXAMPLES:

```
sage: FS = FockSpace(2)
sage: FS.highest_weight_vector()
|>
sage: FS = FockSpace(4, [2, 0, 1])
sage: FS.highest_weight_vector()
|[], [], []>
```

```
>>> from sage.all import *
>>> FS = FockSpace(Integer(2))
>>> FS.highest_weight_vector()
|>
>>> FS = FockSpace(Integer(4), [Integer(2), Integer(0), Integer(1)])
>>> FS.highest_weight_vector()
|[], [], []>
```

inject_shorthands (*verbose=True*)

Import standard shorthands into the global namespace.

INPUT:

- *verbose* – boolean (default: True); if True, prints the defined shorthands

EXAMPLES:

```
sage: FS = FockSpace(4)
sage: FS.inject_shorthands()
Injecting A as shorthand for Fock space of rank 4
of multicharge (0,) over Fraction Field
of Univariate Polynomial Ring in q over Integer Ring
in the approximation basis
Injecting F as shorthand for Fock space of rank 4
of multicharge (0,) over Fraction Field
of Univariate Polynomial Ring in q over Integer Ring
in the natural basis
Injecting G as shorthand for Fock space of rank 4
of multicharge (0,) over Fraction Field
of Univariate Polynomial Ring in q over Integer Ring
in the lower global crystal basis
```

```
>>> from sage.all import *
>>> FS = FockSpace(Integer(4))
>>> FS.inject_shorthands()
Injecting A as shorthand for Fock space of rank 4
of multicharge (0,) over Fraction Field
of Univariate Polynomial Ring in q over Integer Ring
in the approximation basis
Injecting F as shorthand for Fock space of rank 4
of multicharge (0,) over Fraction Field
of Univariate Polynomial Ring in q over Integer Ring
in the natural basis
Injecting G as shorthand for Fock space of rank 4
of multicharge (0,) over Fraction Field
of Univariate Polynomial Ring in q over Integer Ring
in the lower global crystal basis
```

lower_global_crystal

alias of G

multicharge()

Return the multicharge of `self`.

EXAMPLES:

```
sage: F = FockSpace(2)
sage: F.multicharge()
(0,)

sage: F = FockSpace(4, [2, 0, 1])
sage: F.multicharge()
(2, 0, 1)
```

```
>>> from sage.all import *
>>> F = FockSpace(Integer(2))
>>> F.multicharge()
(0,)

>>> F = FockSpace(Integer(4), [Integer(2), Integer(0), Integer(1)])
>>> F.multicharge()
(2, 0, 1)
```

natural

alias of F

options = Current options for FockSpace - display: ket

q()

Return the parameter q of `self`.

EXAMPLES:

```
sage: F = FockSpace(2)
sage: F.q()
q

sage: F = FockSpace(2, q=-1)
sage: F.q()
-1
```

```
>>> from sage.all import *
>>> F = FockSpace(Integer(2))
>>> F.q()
q

>>> F = FockSpace(Integer(2), q=-Integer(1))
>>> F.q()
-1
```

class sage.algebras.quantum_groups.fock_space.**FockSpaceBases** (*base*)

Bases: `Category_realization_of_parent`

The category of bases of a (truncated) Fock space.

class ParentMethods

Bases: object

highest_weight_vector()Return the highest weight vector of `self`.

EXAMPLES:

```
sage: FS = FockSpace(2)
sage: F = FS.natural()
sage: F.highest_weight_vector()
|>
sage: A = FS.A()
sage: A.highest_weight_vector()
A[]
sage: G = FS.G()
sage: G.highest_weight_vector()
G[]
```

```
>>> from sage.all import *
>>> FS = FockSpace(Integer(2))
>>> F = FS.natural()
>>> F.highest_weight_vector()
|>
>>> A = FS.A()
>>> A.highest_weight_vector()
A[]
>>> G = FS.G()
>>> G.highest_weight_vector()
G[]
```

multicharge()Return the multicharge of `self`.

EXAMPLES:

```
sage: FS = FockSpace(4)
sage: A = FS.A()
sage: A.multicharge()
(0,)

sage: FS = FockSpace(4, [1, 0, 2])
sage: G = FS.G()
sage: G.multicharge()
(1, 0, 2)
```

```
>>> from sage.all import *
>>> FS = FockSpace(Integer(4))
>>> A = FS.A()
>>> A.multicharge()
(0,)

>>> FS = FockSpace(Integer(4), [Integer(1), Integer(0), Integer(2)])
>>> G = FS.G()
```

(continues on next page)

(continued from previous page)

```
>>> G.multicharge()
(1, 0, 2)
```

q()Return the parameter q of `self`.

EXAMPLES:

```
sage: FS = FockSpace(2)
sage: A = FS.A()
sage: A.q()
q

sage: FS = FockSpace(2, q=-1)
sage: G = FS.G()
sage: G.q()
-1
```

```
>>> from sage.all import *
>>> FS = FockSpace(Integer(2))
>>> A = FS.A()
>>> A.q()
q

>>> FS = FockSpace(Integer(2), q=-Integer(1))
>>> G = FS.G()
>>> G.q()
-1
```

some_elements()Return some elements of `self`.

EXAMPLES:

```
sage: F = FockSpace(3).natural()
sage: F.some_elements()[::13]
[3*|2> + 2*|1> + 2*|>,
 |5>,
 |3, 1, 1, 1>,
 |3, 2, 2>,
 |5, 1, 1, 1>,
 |2, 2, 1, 1, 1, 1>,
 |5, 2, 1, 1>,
 |3, 2, 1, 1, 1, 1>]

sage: F = FockSpace(3, [0,1]).natural()
sage: F.some_elements()[::13]
[2*|[1], []> + 4*|[1], [1]> + |[1], []>,
 |[1, 1], [1]>,
 |[1, 1, 1], [1]>,
 |[5], []>,
 |[3], [1, 1]>,
 |[1], [2, 2]>]
```

(continues on next page)

(continued from previous page)

```
|[4, 1, 1], []>,
|[2, 1, 1, 1], [1]>]
```

```
>>> from sage.all import *
>>> F = FockSpace(Integer(3)).natural()
>>> F.some_elements()[::Integer(13)]
[3*|2> + 2*|1> + 2*|>,
 |5>,
 |3, 1, 1, 1>,
 |3, 2, 2>,
 |5, 1, 1, 1>,
 |2, 2, 1, 1, 1, 1>,
 |5, 2, 1, 1>,
 |3, 2, 1, 1, 1, 1>]

>>> F = FockSpace(Integer(3), [Integer(0), Integer(1)]).natural()
>>> F.some_elements()[::Integer(13)]
[2*|[1], []> + 4*|[], [1]> + |[[], []>,
 |[1, 1], [1]>,
 |[1, 1, 1], [1]>,
 |[5], []>,
 |[3], [1, 1]>,
 |[1], [2, 2]>,
 |[4, 1, 1], []>,
 |[2, 1, 1, 1], [1]>]
```

super_categories()

The super categories of self.

EXAMPLES:

```
sage: from sage.algebras.quantum_groups.fock_space import FockSpaceBases
sage: F = FockSpace(2)
sage: bases = FockSpaceBases(F)
sage: bases.super_categories()
[Category of vector spaces with basis over Fraction Field
 of Univariate Polynomial Ring in q over Integer Ring,
 Category of realizations of Fock space of rank 2 of multicharge (0,)
 over Fraction Field of Univariate Polynomial Ring in q over Integer Ring]
```

```
>>> from sage.all import *
>>> from sage.algebras.quantum_groups.fock_space import FockSpaceBases
>>> F = FockSpace(Integer(2))
>>> bases = FockSpaceBases(F)
>>> bases.super_categories()
[Category of vector spaces with basis over Fraction Field
 of Univariate Polynomial Ring in q over Integer Ring,
 Category of realizations of Fock space of rank 2 of multicharge (0,)
 over Fraction Field of Univariate Polynomial Ring in q over Integer Ring]
```

sage.algebras.quantum_groups.fock_space.**FockSpaceOptions**(*get_value, **set_value)

Set and display the global options for elements of the Fock space classes. If no parameters are set, then the function returns a copy of the options dictionary.

The options to Fock space can be accessed as the method `FockSpaceOptions` of `FockSpace` and related parent classes.

OPTIONS:

- `display` – (default: `ket`) Specifies how terms of the natural basis of Fock space should be printed
 - `ket` – displayed as a ket in bra-ket notation
 - `list` – displayed as a list

EXAMPLES:

```
sage: FS = FockSpace(4)
sage: F = FS.natural()
sage: x = F.an_element()
sage: y = x.f(3,2,2,0,1)
sage: y
((3*q^2+3)/q)*|3, 3, 1> + (3*q^2+3)*|3, 2, 1, 1>
sage: Partitions.options.display = 'diagram'
sage: y
((3*q^2+3)/q)*|3, 3, 1> + (3*q^2+3)*|3, 2, 1, 1>
sage: ascii_art(y)
((3*q^2+3)/q)*|***\  + (3*q^2+3)*|***\
                |*** >                |**  \
                |*   /                  |*   /
                |*   /                  |*   /

sage: FockSpace.options.display = 'list'
sage: ascii_art(y)
((3*q^2+3)/q)*F      + (3*q^2+3)*F
                ***          ***
                ***          **
                *            *
                *            *

sage: Partitions.options.display = 'compact_high'
sage: y
((3*q^2+3)/q)*F3^2,1 + (3*q^2+3)*F3,2,1^2

sage: Partitions.options._reset()
sage: FockSpace.options._reset()
```

```
>>> from sage.all import *
>>> FS = FockSpace(Integer(4))
>>> F = FS.natural()
>>> x = F.an_element()
>>> y = x.f(Integer(3), Integer(2), Integer(2), Integer(0), Integer(1))
>>> y
((3*q^2+3)/q)*|3, 3, 1> + (3*q^2+3)*|3, 2, 1, 1>
>>> Partitions.options.display = 'diagram'
>>> y
((3*q^2+3)/q)*|3, 3, 1> + (3*q^2+3)*|3, 2, 1, 1>
>>> ascii_art(y)
((3*q^2+3)/q)*|***\  + (3*q^2+3)*|***\
                |*** >                |**  \
                |*   /                  |*   /
                |*   /                  |*   /
```

(continues on next page)

(continued from previous page)

```

>>> FockSpace.options.display = 'list'
>>> ascii_art(y)
((3*q^2+3)/q)*F      + (3*q^2+3)*F
      ***              ***
      ***              **
      *                *
                        *

>>> Partitions.options.display = 'compact_high'
>>> y
((3*q^2+3)/q)*F3^2,1 + (3*q^2+3)*F3,2,1^2

>>> Partitions.options._reset()
>>> FockSpace.options._reset()

```

See `GlobalOptions` for more features of these options.

class `sage.algebras.quantum_groups.fock_space.FockSpaceTruncated(n, k, q, base_ring)`

Bases: `FockSpace`

This is the Fock space given by partitions of length no more than k .

This can be formed as the quotient $\mathcal{F}/\mathcal{F}_k$, where $\mathcal{F}_k$ is the submodule spanned by all diagrams of length (strictly) more than k .

We have three bases:

- The natural basis indexed by truncated n -regular partitions: F .
- The approximation basis that comes from LLT(-type) algorithms: A .
- The lower global crystal basis: G .

See also

`FockSpace`

EXAMPLES:

```

sage: F = FockSpace(2, truncated=2)
sage: mg = F.highest_weight_vector()
sage: mg.f(0)
|1>
sage: mg.f(0).f(1)
|2> + q*|1, 1>
sage: mg.f(0).f(1).f(0)
|3>

```

```

>>> from sage.all import *
>>> F = FockSpace(Integer(2), truncated=Integer(2))
>>> mg = F.highest_weight_vector()
>>> mg.f(Integer(0))
|1>
>>> mg.f(Integer(0)).f(Integer(1))
|2> + q*|1, 1>

```

(continues on next page)

(continued from previous page)

```
>>> mg.f(Integer(0)).f(Integer(1)).f(Integer(0))
|3>
```

Compare this to the full Fock space:

```
sage: F = FockSpace(2)
sage: mg = F.highest_weight_vector()
sage: mg.f(0).f(1).f(0)
|3> + q*|1, 1, 1>
```

```
>>> from sage.all import *
>>> F = FockSpace(Integer(2))
>>> mg = F.highest_weight_vector()
>>> mg.f(Integer(0)).f(Integer(1)).f(Integer(0))
|3> + q*|1, 1, 1>
```

REFERENCES:

- [GW1999]

class `A(F, algorithm='GW')`

Bases: `CombinatorialFreeModule`, `BindableClass`

The A basis of the Fock space, which is the approximation basis of the lower global crystal basis.

INPUT:

- `algorithm` – (default: 'GW') the algorithm to use when computing this basis in the Fock space; the possible values are:
 - 'GW' – use the algorithm given by Goodman and Wenzl in [GW1999]
 - 'LLT' – use the LLT algorithm given in [LLT1996]

Note

The bases produced by the two algorithms are not the same in general.

EXAMPLES:

```
sage: FS = FockSpace(5, truncated=4)
sage: F = FS.natural()
sage: A = FS.A()
```

```
>>> from sage.all import *
>>> FS = FockSpace(Integer(5), truncated=Integer(4))
>>> F = FS.natural()
>>> A = FS.A()
```

We demonstrate that they are different bases, but both algorithms still compute the basis G :

```
sage: A2 = FS.A('LLT')
sage: G = FS.G()
sage: F(A[12, 9])
```

(continues on next page)

(continued from previous page)

```

|12, 9> + q*|12, 4, 4, 1> + q*|8, 8, 5> + (q^2+1)*|8, 8, 4, 1>
sage: F(A2[12,9])
|12, 9> + q*|12, 4, 4, 1> + q*|8, 8, 5> + (q^2+2)*|8, 8, 4, 1>
sage: G._G_to_fock_basis(Partition([12,9]), 'GW')
|12, 9> + q*|12, 4, 4, 1> + q*|8, 8, 5> + q^2*|8, 8, 4, 1>
sage: G._G_to_fock_basis(Partition([12,9]), 'LLT')
|12, 9> + q*|12, 4, 4, 1> + q*|8, 8, 5> + q^2*|8, 8, 4, 1>

```

```

>>> from sage.all import *
>>> A2 = FS.A('LLT')
>>> G = FS.G()
>>> F(A[Integer(12), Integer(9)])
|12, 9> + q*|12, 4, 4, 1> + q*|8, 8, 5> + (q^2+1)*|8, 8, 4, 1>
>>> F(A2[Integer(12), Integer(9)])
|12, 9> + q*|12, 4, 4, 1> + q*|8, 8, 5> + (q^2+2)*|8, 8, 4, 1>
>>> G._G_to_fock_basis(Partition([Integer(12), Integer(9)]), 'GW')
|12, 9> + q*|12, 4, 4, 1> + q*|8, 8, 5> + q^2*|8, 8, 4, 1>
>>> G._G_to_fock_basis(Partition([Integer(12), Integer(9)]), 'LLT')
|12, 9> + q*|12, 4, 4, 1> + q*|8, 8, 5> + q^2*|8, 8, 4, 1>

```

options = Current options for FockSpace - display: ket

class F(F)

Bases: `CombinatorialFreeModule`, `BindableClass`

The natural basis of the truncated Fock space.

This is the natural basis of the full Fock space projected onto the truncated Fock space. It inherits the $U_q(\widehat{sl}_n)$ -action from the action on the full Fock space.

EXAMPLES:

```

sage: FS = FockSpace(4)
sage: F = FS.natural()
sage: FS3 = FockSpace(4, truncated=3)
sage: F3 = FS3.natural()
sage: u = F.highest_weight_vector()
sage: u3 = F3.highest_weight_vector()

sage: u3.f(0,3,2,1)
|2, 1, 1>
sage: u.f(0,3,2,1)
|2, 1, 1> + q*|1, 1, 1, 1>

sage: u.f(0,3,2,1,1)
((q^2+1)/q)*|2, 1, 1, 1>
sage: u3.f(0,3,2,1,1)
0

```

```

>>> from sage.all import *
>>> FS = FockSpace(Integer(4))
>>> F = FS.natural()
>>> FS3 = FockSpace(Integer(4), truncated=Integer(3))

```

(continues on next page)

(continued from previous page)

```

>>> F3 = FS3.natural()
>>> u = F.highest_weight_vector()
>>> u3 = F3.highest_weight_vector()

>>> u3.f(Integer(0), Integer(3), Integer(2), Integer(1))
|2, 1, 1>
>>> u.f(Integer(0), Integer(3), Integer(2), Integer(1))
|2, 1, 1> + q*|1, 1, 1, 1>

>>> u.f(Integer(0), Integer(3), Integer(2), Integer(1), Integer(1))
((q^2+1)/q)*|2, 1, 1, 1>
>>> u3.f(Integer(0), Integer(3), Integer(2), Integer(1), Integer(1))
0

```

class ElementBases: *Element*

An element in the truncated Fock space.

options = Current options for FockSpace - display: ket**class G(F)**Bases: *CombinatorialFreeModule*, *BindableClass*

The lower global crystal basis living inside of a truncated Fock space.

EXAMPLES:

```

sage: FS = FockSpace(4, truncated=2)
sage: F = FS.natural()
sage: G = FS.G()
sage: F(G[3,1])
|3, 1>
sage: F(G[6,2])
|6, 2> + q*|5, 3>
sage: F(G[14])
|14> + q*|11, 3>

sage: FS = FockSpace(3, truncated=4)
sage: F = FS.natural()
sage: G = FS.G()
sage: F(G[4,1])
|4, 1> + q*|3, 2>
sage: F(G[4,2,2])
|4, 2, 2> + q*|3, 2, 2, 1>

```

```

>>> from sage.all import *
>>> FS = FockSpace(Integer(4), truncated=Integer(2))
>>> F = FS.natural()
>>> G = FS.G()
>>> F(G[Integer(3), Integer(1)])
|3, 1>
>>> F(G[Integer(6), Integer(2)])
|6, 2> + q*|5, 3>

```

(continues on next page)

(continued from previous page)

```

>>> F(G[Integer(14)])
|14> + q*|11, 3>

>>> FS = FockSpace(Integer(3), truncated=Integer(4))
>>> F = FS.natural()
>>> G = FS.G()
>>> F(G[Integer(4), Integer(1)])
|4, 1> + q*|3, 2>
>>> F(G[Integer(4), Integer(2), Integer(2)])
|4, 2, 2> + q*|3, 2, 2, 1>

```

We check against the tables in [LLT1996] (after truncating):

```

sage: FS = FockSpace(3, truncated=3)
sage: F = FS.natural()
sage: G = FS.G()
sage: F(G[10])
|10> + q*|8, 2> + q*|7, 2, 1>
sage: F(G[6,4])
|6, 4> + q*|6, 2, 2> + q^2*|4, 4, 2>
sage: F(G[5,5])
|5, 5> + q*|4, 3, 3>

sage: FS = FockSpace(4, truncated=3)
sage: F = FS.natural()
sage: G = FS.G()
sage: F(G[3,3,1])
|3, 3, 1>
sage: F(G[3,2,2])
|3, 2, 2>
sage: F(G[7])
|7> + q*|3, 3, 1>

```

```

>>> from sage.all import *
>>> FS = FockSpace(Integer(3), truncated=Integer(3))
>>> F = FS.natural()
>>> G = FS.G()
>>> F(G[Integer(10)])
|10> + q*|8, 2> + q*|7, 2, 1>
>>> F(G[Integer(6), Integer(4)])
|6, 4> + q*|6, 2, 2> + q^2*|4, 4, 2>
>>> F(G[Integer(5), Integer(5)])
|5, 5> + q*|4, 3, 3>

>>> FS = FockSpace(Integer(4), truncated=Integer(3))
>>> F = FS.natural()
>>> G = FS.G()
>>> F(G[Integer(3), Integer(3), Integer(1)])
|3, 3, 1>
>>> F(G[Integer(3), Integer(2), Integer(2)])
|3, 2, 2>
>>> F(G[Integer(7)])

```

(continues on next page)

(continued from previous page)

```
|7> + q*|3, 3, 1>
```

options = Current options for FockSpace - display: ket

approximation
alias of $\mathbb{A}$

canonical
alias of G

lower_global_crystal
alias of G

natural
alias of F

2.3 q -Numbers

Note

These are the quantum group q -analogs, not the usual q -analogs typically used in combinatorics (see `sage.combinat.q_analogues`).

`sage.algebras.quantum_groups.q_numbers.q_binomial(n, k, q=None)`

Return the q -binomial coefficient.

Let $[n]_q!$ denote the q -factorial of n given by `sage.algebras.quantum_groups.q_numbers.q_factorial()`. The q -binomial coefficient is defined by

$$\begin{bmatrix} n \\ k \end{bmatrix}_q = \frac{[n]_q!}{[n-k]_q! \cdot [k]_q!}.$$

INPUT:

- n, k – the nonnegative integers n and k defined above
- q – (default: $q \in \mathbb{Z}[q, q^{-1}]$) the parameter q (should be invertible)

If q is unspecified, then it is taken to be the generator q for a Laurent polynomial ring over the integers.

Note

This is not the “usual” q -binomial but a variant useful for quantum groups. For the version used in combinatorics, see `sage.combinat.q_analogues`.

Warning

This method uses division by q -factorials. If $[k]_q!$ or $[n-k]_q!$ are zero-divisors, or division is not implemented in the ring containing q , then it will not work.

EXAMPLES:


```

sage: from sage.algebras.quantum_groups.q_numbers import q_binomial
sage: q_binomial(2, 1)
q^-1 + q
sage: q_binomial(2, 0)
1
sage: q_binomial(4, 1)
q^-3 + q^-1 + q + q^3
sage: q_binomial(4, 3)
q^-3 + q^-1 + q + q^3

```

```

>>> from sage.all import *
>>> from sage.algebras.quantum_groups.q_numbers import q_binomial
>>> q_binomial(Integer(2), Integer(1))
q^-1 + q
>>> q_binomial(Integer(2), Integer(0))
1
>>> q_binomial(Integer(4), Integer(1))
q^-3 + q^-1 + q + q^3
>>> q_binomial(Integer(4), Integer(3))
q^-3 + q^-1 + q + q^3

```

`sage.algebras.quantum_groups.q_numbers.q_factorial(n, q=None)`

Return the q -analogue of the factorial $n!$.

The q -factorial is defined by:

$$[n]_q! = [n]_q \cdot [n-1]_q \cdots [2]_q \cdot [1]_q,$$

where $[n]_q$ denotes the q -integer defined in `sage.algebras.quantum_groups.q_numbers.q_int()`.

INPUT:

- n – the nonnegative integer n defined above
- q – (default: $q \in \mathbb{Z}[q, q^{-1}]$) the parameter q (should be invertible)

If q is unspecified, then it defaults to using the generator q for a Laurent polynomial ring over the integers.

Note

This is not the “usual” q -factorial but a variant useful for quantum groups. For the version used in combinatorics, see `sage.combinat.q_analogues`.

EXAMPLES:

```

sage: from sage.algebras.quantum_groups.q_numbers import q_factorial
sage: q_factorial(3)
q^-3 + 2*q^-1 + 2*q + q^3
sage: p = LaurentPolynomialRing(QQ, 'q').gen()
sage: q_factorial(3, p)
q^-3 + 2*q^-1 + 2*q + q^3
sage: p = ZZ['p'].gen()
sage: q_factorial(3, p)
(p^6 + 2*p^4 + 2*p^2 + 1)/p^3

```

```

>>> from sage.all import *
>>> from sage.algebras.quantum_groups.q_numbers import q_factorial
>>> q_factorial(Integer(3))
q^-3 + 2*q^-1 + 2*q + q^3
>>> p = LaurentPolynomialRing(QQ, 'q').gen()
>>> q_factorial(Integer(3), p)
q^-3 + 2*q^-1 + 2*q + q^3
>>> p = ZZ['p'].gen()
>>> q_factorial(Integer(3), p)
(p^6 + 2*p^4 + 2*p^2 + 1)/p^3

```

The q -analog of $n!$ is only defined for n a nonnegative integer ([Issue #11411](#)):

```

sage: q_factorial(-2)
Traceback (most recent call last):
...
ValueError: argument (-2) must be a nonnegative integer

```

```

>>> from sage.all import *
>>> q_factorial(-Integer(2))
Traceback (most recent call last):
...
ValueError: argument (-2) must be a nonnegative integer

```

`sage.algebras.quantum_groups.q_numbers.q_int` ($n, q=None$)

Return the q -analog of the nonnegative integer n .

The q -analog of the nonnegative integer n is given by

$$[n]_q = \frac{q^n - q^{-n}}{q - q^{-1}} = q^{n-1} + q^{n-3} + \cdots + q^{-n+3} + q^{-n+1}.$$

INPUT:

- n – the nonnegative integer n defined above
- q – (default: $q \in \mathbb{Z}[q, q^{-1}]$) the parameter q (should be invertible)

If q is unspecified, then it defaults to using the generator q for a Laurent polynomial ring over the integers.

Note

This is not the “usual” q -analog of n (or q -integer) but a variant useful for quantum groups. For the version used in combinatorics, see `sage.combinat.q_analogues`.

EXAMPLES:

```

sage: from sage.algebras.quantum_groups.q_numbers import q_int
sage: q_int(2)
q^-1 + q
sage: q_int(3)
q^-2 + 1 + q^2
sage: q_int(5)
q^-4 + q^-2 + 1 + q^2 + q^4

```

(continues on next page)

(continued from previous page)

```
sage: q_int(5, 1)
5
```

```
>>> from sage.all import *
>>> from sage.algebras.quantum_groups.q_numbers import q_int
>>> q_int(Integer(2))
q^-1 + q
>>> q_int(Integer(3))
q^-2 + 1 + q^2
>>> q_int(Integer(5))
q^-4 + q^-2 + 1 + q^2 + q^4
>>> q_int(Integer(5), Integer(1))
5
```

2.4 Quantum Group Representations

AUTHORS:

- Travis Scrimshaw (2018): initial version

class sage.algebras.quantum_groups.representations.**AdjointRepresentation**(R, C, q)

Bases: *CyclicRepresentation*

An (generalized) adjoint representation of a quantum group.

We define an (*generalized*) *adjoint representation* V of a quantum group U_q to be a cyclic U_q -module with a weight space decomposition $V = \bigoplus_{\mu} V_{\mu}$ such that $\dim V_{\mu} \leq 1$ unless $\mu = 0$. Moreover, we require that there exists a basis $\{y_j | j \in J\}$ for V_0 such that $e_i y_j = 0$ for all $j \neq i \in I$.

For a base ring R , we construct an adjoint representation from its (combinatorial) crystal B by $V = R\{v_b | b \in B\}$ with

$$e_i v_b = \begin{cases} v_{e_i b} / [\varphi_i(e_i b)]_{q_i}, & \text{if } \text{wt}(b) \neq 0, \\ v_{e_i b} + \sum_{j \neq i} [-A_{ij}]_{q_i} / [2]_{q_i} v_{y_j} & \text{otherwise} \end{cases}$$

$$f_i v_b = \begin{cases} v_{f_i b} / [\varepsilon_i(f_i b)]_{q_i}, & \text{if } \text{wt}(b) \neq 0, \\ v_{f_i b} + \sum_{j \neq i} [-A_{ij}]_{q_i} / [2]_{q_i} v_{y_j} & \text{otherwise} \end{cases}$$

$$K_i v_b = q^{\langle h_i, \text{wt}(b) \rangle} v_b,$$

where $(A_{ij})_{i,j \in I}$ is the Cartan matrix, and we consider $v_0 := 0$.

INPUT:

- C – the crystal corresponding to the representation
- R – the base ring
- q – (default: the generator of R) the parameter q of the quantum group

Warning

This assumes that q is generic.

EXAMPLES:

```

sage: from sage.algebras.quantum_groups.representations import_
      ↪ AdjointRepresentation
sage: R = ZZ['q'].fraction_field()
sage: C = crystals.Tableaux(['D', 4], shape=[1, 1])
sage: V = AdjointRepresentation(R, C)
sage: V
V((1, 1, 0, 0))
sage: v = V.an_element(); v
2*B[[[1], [2]]] + 2*B[[[1], [3]]] + 3*B[[[2], [3]]]
sage: v.e(2)
2*B[[[1], [2]]]
sage: v.f(2)
2*B[[[1], [3]]]
sage: v.f(4)
2*B[[[1], [-4]]] + 3*B[[[2], [-4]]]
sage: v.K(3)
2*B[[[1], [2]]] + 2*q*B[[[1], [3]]] + 3*q*B[[[2], [3]]]
sage: v.K(2, -2)
2/q^2*B[[[1], [2]]] + 2*q^2*B[[[1], [3]]] + 3*B[[[2], [3]]]

sage: La = RootSystem(['F', 4, 1]).weight_space().fundamental_weights()
sage: K = crystals.ProjectedListZeroLSPaths(La[4])
sage: A = AdjointRepresentation(R, K)
sage: A
V(-Lambda[0] + Lambda[4])

```

```

>>> from sage.all import *
>>> from sage.algebras.quantum_groups.representations import AdjointRepresentation
>>> R = ZZ['q'].fraction_field()
>>> C = crystals.Tableaux(['D', Integer(4)], shape=[Integer(1), Integer(1)])
>>> V = AdjointRepresentation(R, C)
>>> V
V((1, 1, 0, 0))
>>> v = V.an_element(); v
2*B[[[1], [2]]] + 2*B[[[1], [3]]] + 3*B[[[2], [3]]]
>>> v.e(Integer(2))
2*B[[[1], [2]]]
>>> v.f(Integer(2))
2*B[[[1], [3]]]
>>> v.f(Integer(4))
2*B[[[1], [-4]]] + 3*B[[[2], [-4]]]
>>> v.K(Integer(3))
2*B[[[1], [2]]] + 2*q*B[[[1], [3]]] + 3*q*B[[[2], [3]]]
>>> v.K(Integer(2), -Integer(2))
2/q^2*B[[[1], [2]]] + 2*q^2*B[[[1], [3]]] + 3*B[[[2], [3]]]

>>> La = RootSystem(['F', Integer(4), Integer(1)]).weight_space().fundamental_
      ↪ weights()
>>> K = crystals.ProjectedListZeroLSPaths(La[Integer(4)])
>>> A = AdjointRepresentation(R, K)
>>> A
V(-Lambda[0] + Lambda[4])

```

Sort the summands uniformly in Python 2 and Python 3:

```
sage: A.print_options(sorting_key=lambda x: str(x))
sage: v = A.an_element(); v
2*B[(-Lambda[0] + Lambda[3] - Lambda[4],)]
+ 2*B[(-Lambda[0] + Lambda[4],)]
+ 3*B[(Lambda[0] - Lambda[1] + Lambda[4],)]
sage: v.e(0)
2*B[(Lambda[0] - Lambda[1] + Lambda[3] - Lambda[4],)]
+ 2*B[(Lambda[0] - Lambda[1] + Lambda[4],)]
sage: v.f(0)
3*B[(-Lambda[0] + Lambda[4],)]
```

```
>>> from sage.all import *
>>> A.print_options(sorting_key=lambda x: str(x))
>>> v = A.an_element(); v
2*B[(-Lambda[0] + Lambda[3] - Lambda[4],)]
+ 2*B[(-Lambda[0] + Lambda[4],)]
+ 3*B[(Lambda[0] - Lambda[1] + Lambda[4],)]
>>> v.e(Integer(0))
2*B[(Lambda[0] - Lambda[1] + Lambda[3] - Lambda[4],)]
+ 2*B[(Lambda[0] - Lambda[1] + Lambda[4],)]
>>> v.f(Integer(0))
3*B[(-Lambda[0] + Lambda[4],)]
```

REFERENCES:

- [OS2018]

`e_on_basis(i, b)`

Return the action of e_i on the basis element indexed by b .

INPUT:

- i – an element of the index set
- b – an element of basis keys

EXAMPLES:

```
sage: from sage.algebras.quantum_groups.representations import_
↳ AdjointRepresentation
sage: K = crystals.KirillovReshetikhin(['D', 3, 2], 1, 1)
sage: R = ZZ['q'].fraction_field()
sage: V = AdjointRepresentation(R, K)
sage: mg0 = K.module_generators[0]; mg0
[]
sage: mg1 = K.module_generators[1]; mg1
[[1]]
sage: V.e_on_basis(0, mg0)
((q^2+1)/q)*B[[-1]]
sage: V.e_on_basis(0, mg1)
B[]
sage: V.e_on_basis(1, mg0)
0
sage: V.e_on_basis(1, mg1)
```

(continues on next page)

(continued from previous page)

```

0
sage: V.e_on_basis(2, mg0)
0
sage: V.e_on_basis(2, mg1)
0

sage: K = crystals.KirillovReshetikhin(['D', 4, 3], 1, 1)
sage: V = AdjointRepresentation(R, K)
sage: V.e_on_basis(0, K.module_generator())
B[[[]] + (q/(q^2+1))*B[[[0]]]

```

```

>>> from sage.all import *
>>> from sage.algebras.quantum_groups.representations import _
↳ AdjointRepresentation
>>> K = crystals.KirillovReshetikhin(['D', Integer(3), Integer(2)], Integer(1),
↳ Integer(1))
>>> R = ZZ['q'].fraction_field()
>>> V = AdjointRepresentation(R, K)
>>> mg0 = K.module_generators[Integer(0)]; mg0
[]
>>> mg1 = K.module_generators[Integer(1)]; mg1
[[1]]
>>> V.e_on_basis(Integer(0), mg0)
((q^2+1)/q)*B[[[-1]]]
>>> V.e_on_basis(Integer(0), mg1)
B[[[]]
>>> V.e_on_basis(Integer(1), mg0)
0
>>> V.e_on_basis(Integer(1), mg1)
0
>>> V.e_on_basis(Integer(2), mg0)
0
>>> V.e_on_basis(Integer(2), mg1)
0

>>> K = crystals.KirillovReshetikhin(['D', Integer(4), Integer(3)], Integer(1),
↳ Integer(1))
>>> V = AdjointRepresentation(R, K)
>>> V.e_on_basis(Integer(0), K.module_generator())
B[[[]] + (q/(q^2+1))*B[[[0]]]

```

f_on_basis(*i*, *b*)Return the action of f_i on the basis element indexed by *b*.

INPUT:

- *i* – an element of the index set
- *b* – an element of basis keys

EXAMPLES:

```

sage: from sage.algebras.quantum_groups.representations import _
↳ AdjointRepresentation

```

(continues on next page)

(continued from previous page)

```

sage: K = crystals.KirillovReshetikhin(['D',3,2], 1,1)
sage: R = ZZ['q'].fraction_field()
sage: V = AdjointRepresentation(R, K)
sage: mg0 = K.module_generators[0]; mg0
[]
sage: mg1 = K.module_generators[1]; mg1
[[1]]
sage: V.f_on_basis(0, mg0)
((q^2+1)/q)*B[[[1]]]
sage: V.f_on_basis(0, mg1)
0
sage: V.f_on_basis(1, mg0)
0
sage: V.f_on_basis(1, mg1)
B[[[2]]]
sage: V.f_on_basis(2, mg0)
0
sage: V.f_on_basis(2, mg1)
0

sage: K = crystals.KirillovReshetikhin(['D',4,3], 1,1)
sage: V = AdjointRepresentation(R, K)
sage: lw = K.module_generator().to_lowest_weight([1,2])[0]
sage: V.f_on_basis(0, lw)
B[[[]] + (q/(q^2+1))*B[[[0]]]

```

```

>>> from sage.all import *
>>> from sage.algebras.quantum_groups.representations import _
↪AdjointRepresentation
>>> K = crystals.KirillovReshetikhin(['D',Integer(3),Integer(2)], Integer(1),
↪Integer(1))
>>> R = ZZ['q'].fraction_field()
>>> V = AdjointRepresentation(R, K)
>>> mg0 = K.module_generators[Integer(0)]; mg0
[]
>>> mg1 = K.module_generators[Integer(1)]; mg1
[[1]]
>>> V.f_on_basis(Integer(0), mg0)
((q^2+1)/q)*B[[[1]]]
>>> V.f_on_basis(Integer(0), mg1)
0
>>> V.f_on_basis(Integer(1), mg0)
0
>>> V.f_on_basis(Integer(1), mg1)
B[[[2]]]
>>> V.f_on_basis(Integer(2), mg0)
0
>>> V.f_on_basis(Integer(2), mg1)
0

>>> K = crystals.KirillovReshetikhin(['D',Integer(4),Integer(3)], Integer(1),
↪Integer(1))

```

(continues on next page)

(continued from previous page)

```

>>> V = AdjointRepresentation(R, K)
>>> lw = K.module_generator().to_lowest_weight([Integer(1),
↪Integer(2)])[Integer(0)]
>>> V.f_on_basis(Integer(0), lw)
B[[[]] + (q/(q^2+1))*B[[[0]]]

```

class sage.algebras.quantum_groups.representations.**CyclicRepresentation**(*R, C, q*)

Bases: *QuantumGroupRepresentation*

A cyclic quantum group representation that is indexed by either a highest weight crystal or Kirillov-Reshetikhin crystal.

The crystal *C* must either allow *C*.module_generator(), otherwise it is assumed to be generated by *C*.module_generators[0].

This is meant as an abstract base class for AdjointRepresentation and MinusculeRepresentation.

module_generator()

Return the module generator of self.

EXAMPLES:

```

sage: from sage.algebras.quantum_groups.representations import _
↪AdjointRepresentation
sage: C = crystals.Tableaux(['G', 2], shape=[1, 1])
sage: R = ZZ['q'].fraction_field()
sage: V = AdjointRepresentation(R, C)
sage: V.module_generator()
B[[[1], [2]]]

sage: K = crystals.KirillovReshetikhin(['D', 4, 2], 1, 1)
sage: A = AdjointRepresentation(R, K)
sage: A.module_generator()
B[[[1]]]

```

```

>>> from sage.all import *
>>> from sage.algebras.quantum_groups.representations import _
↪AdjointRepresentation
>>> C = crystals.Tableaux(['G', Integer(2)], shape=[Integer(1), Integer(1)])
>>> R = ZZ['q'].fraction_field()
>>> V = AdjointRepresentation(R, C)
>>> V.module_generator()
B[[[1], [2]]]

>>> K = crystals.KirillovReshetikhin(['D', Integer(4), Integer(2)], Integer(1),
↪Integer(1))
>>> A = AdjointRepresentation(R, K)
>>> A.module_generator()
B[[[1]]]

```

class sage.algebras.quantum_groups.representations.**MinusculeRepresentation**(*R, C, q*)

Bases: *CyclicRepresentation*

A minuscule representation of a quantum group.

A quantum group representation V is *minuscule* if it is cyclic, there is a weight space decomposition $V = \bigoplus_{\mu} V_{\mu}$ with $\dim V_{\mu} \leq 1$, and $e_i^2 V = 0$ and $f_i^2 V = 0$.

For a base ring R , we construct a minuscule representation from its (combinatorial) crystal B by $V = R\{v_b | b \in B\}$ with $e_i v_b = v_{e_i b}$, $f_i v_b = v_{f_i b}$, and $K_i v_b = q^{(h_i, \text{wt}(b))} v_b$, where we consider $v_0 := 0$.

INPUT:

- C – the crystal corresponding to the representation
- R – the base ring
- q – (default: the generator of R) the parameter q of the quantum group

Warning

This assumes that q is generic.

EXAMPLES:

```
sage: from sage.algebras.quantum_groups.representations import_
      ↪ Minusculerepresentation
sage: R = ZZ['q'].fraction_field()
sage: C = crystals.Tableaux(['B', 3], shape=[1/2, 1/2, 1/2])
sage: V = Minusculerepresentation(R, C)
sage: V
V((1/2, 1/2, 1/2))
sage: v = V.an_element(); v
2*B[[+++, []]] + 2*B[[++-, []]] + 3*B[[+-+, []]]
sage: v.e(3)
2*B[[+++, []]]
sage: v.f(1)
3*B[[--+, []]]
sage: v.f(3)
2*B[[++-, []]] + 3*B[[+--, []]]
sage: v.K(2)
2*B[[+++, []]] + 2*q^2*B[[++-, []]] + 3/q^2*B[[+-+, []]]
sage: v.K(3, -2)
2/q^2*B[[+++, []]] + 2*q^2*B[[++-, []]] + 3/q^2*B[[+-+, []]]

sage: K = crystals.KirillovReshetikhin(['D', 4, 2], 3, 1)
sage: A = Minusculerepresentation(R, K)
sage: A
V(-Lambda[0] + Lambda[3])
sage: v = A.an_element(); v
2*B[[+++, []]] + 2*B[[++-, []]] + 3*B[[+-+, []]]
sage: v.f(0)
0
sage: v.e(0)
2*B[[--+, []]] + 2*B[[+--, []]] + 3*B[[+-+, []]]
```

```
>>> from sage.all import *
>>> from sage.algebras.quantum_groups.representations import_
      ↪ Minusculerepresentation
```

(continues on next page)

(continued from previous page)

```

>>> R = ZZ['q'].fraction_field()
>>> C = crystals.Tableaux(['B', Integer(3)], shape=[Integer(1)/Integer(2),
↳ Integer(1)/Integer(2), Integer(1)/Integer(2)])
>>> V = MinusculeRepresentation(R, C)
>>> V
V((1/2, 1/2, 1/2))
>>> v = V.an_element(); v
2*B[[+++, []]] + 2*B[[++-, []]] + 3*B[[+--, []]]
>>> v.e(Integer(3))
2*B[[+++, []]]
>>> v.f(Integer(1))
3*B[[+--, []]]
>>> v.f(Integer(3))
2*B[[++-, []]] + 3*B[[+--, []]]
>>> v.K(Integer(2))
2*B[[+++, []]] + 2*q^2*B[[++-, []]] + 3/q^2*B[[+--, []]]
>>> v.K(Integer(3), -Integer(2))
2/q^2*B[[+++, []]] + 2*q^2*B[[++-, []]] + 3/q^2*B[[+--, []]]

>>> K = crystals.KirillovReshetikhin(['D', Integer(4), Integer(2)], Integer(3),
↳ Integer(1))
>>> A = MinusculeRepresentation(R, K)
>>> A
V(-Lambda[0] + Lambda[3])
>>> v = A.an_element(); v
2*B[[+++, []]] + 2*B[[++-, []]] + 3*B[[+--, []]]
>>> v.f(Integer(0))
0
>>> v.e(Integer(0))
2*B[[+--, []]] + 2*B[[+--, []]] + 3*B[[+--, []]]

```

REFERENCES:

- [OS2018]

e_on_basis(*i*, *b*)

Return the action of e_i on the basis element indexed by *b*.

INPUT:

- *i* – an element of the index set
- *b* – an element of basis keys

EXAMPLES:

```

sage: from sage.algebras.quantum_groups.representations import_
↳ MinusculeRepresentation
sage: C = crystals.Tableaux(['A', 3], shape=[1, 1])
sage: R = ZZ['q'].fraction_field()
sage: V = MinusculeRepresentation(R, C)
sage: lw = C.lowest_weight_vectors()[0]
sage: V.e_on_basis(1, lw)
0
sage: V.e_on_basis(2, lw)

```

(continues on next page)

(continued from previous page)

```

B[[[2], [4]]]
sage: V.e_on_basis(3, lw)
0
sage: hw = C.highest_weight_vectors()[0]
sage: all(V.e_on_basis(i, hw) == V.zero() for i in V.index_set())
True

```

```

>>> from sage.all import *
>>> from sage.algebras.quantum_groups.representations import_
↳ MinusculeRepresentation
>>> C = crystals.Tableaux(['A', Integer(3)], shape=[Integer(1), Integer(1)])
>>> R = ZZ['q'].fraction_field()
>>> V = MinusculeRepresentation(R, C)
>>> lw = C.lowest_weight_vectors()[Integer(0)]
>>> V.e_on_basis(Integer(1), lw)
0
>>> V.e_on_basis(Integer(2), lw)
B[[[2], [4]]]
>>> V.e_on_basis(Integer(3), lw)
0
>>> hw = C.highest_weight_vectors()[Integer(0)]
>>> all(V.e_on_basis(i, hw) == V.zero() for i in V.index_set())
True

```

f_on_basis(*i*, *b*)Return the action of f_i on the basis element indexed by *b*.

INPUT:

- *i* – an element of the index set
- *b* – an element of basis keys

EXAMPLES:

```

sage: from sage.algebras.quantum_groups.representations import_
↳ MinusculeRepresentation
sage: C = crystals.Tableaux(['A', 3], shape=[1, 1])
sage: R = ZZ['q'].fraction_field()
sage: V = MinusculeRepresentation(R, C)
sage: hw = C.highest_weight_vectors()[0]
sage: V.f_on_basis(1, hw)
0
sage: V.f_on_basis(2, hw)
B[[[1], [3]]]
sage: V.f_on_basis(3, hw)
0
sage: lw = C.lowest_weight_vectors()[0]
sage: all(V.f_on_basis(i, lw) == V.zero() for i in V.index_set())
True

```

```

>>> from sage.all import *
>>> from sage.algebras.quantum_groups.representations import_

```

(continues on next page)

(continued from previous page)

```

↪ MinusculRepresentation
>>> C = crystals.Tableaux(['A', Integer(3)], shape=[Integer(1), Integer(1)])
>>> R = ZZ['q'].fraction_field()
>>> V = MinusculRepresentation(R, C)
>>> hw = C.highest_weight_vectors()[Integer(0)]
>>> V.f_on_basis(Integer(1), hw)
0
>>> V.f_on_basis(Integer(2), hw)
B[[[1], [3]]]
>>> V.f_on_basis(Integer(3), hw)
0
>>> lw = C.lowest_weight_vectors()[Integer(0)]
>>> all(V.f_on_basis(i, lw) == V.zero() for i in V.index_set())
True

```

class sage.algebras.quantum_groups.representations.**QuantumGroupRepresentation**(*R, C, q*)

Bases: **CombinatorialFreeModule**

A representation of a quantum group whose basis is indexed by the corresponding (combinatorial) crystal.

INPUT:

- *C* – the crystal corresponding to the representation
- *R* – the base ring
- *q* – (default: the generator of *R*) the parameter *q* of the quantum group

K_on_basis(*i, b, power=1*)

Return the action of K_i on the basis element indexed by *b* to the power *power*.

INPUT:

- *i* – an element of the index set
- *b* – an element of basis keys
- *power* – (default: 1) the power of K_i

EXAMPLES:

```

sage: from sage.algebras.quantum_groups.representations import _
↪ MinusculRepresentation
sage: C = crystals.Tableaux(['A', 3], shape=[1, 1])
sage: R = ZZ['q'].fraction_field()
sage: V = MinusculRepresentation(R, C)
sage: [[V.K_on_basis(i, b) for i in V.index_set()] for b in C]
[[B[[[1], [2]]], q*B[[[1], [2]]], B[[[1], [2]]]],
 [q*B[[[1], [3]]], 1/q*B[[[1], [3]]], q*B[[[1], [3]]]],
 [1/q*B[[[2], [3]]], B[[[2], [3]]], q*B[[[2], [3]]]],
 [q*B[[[1], [4]]], B[[[1], [4]]], 1/q*B[[[1], [4]]]],
 [1/q*B[[[2], [4]]], q*B[[[2], [4]]], 1/q*B[[[2], [4]]]],
 [B[[[3], [4]]], 1/q*B[[[3], [4]]], B[[[3], [4]]]]]
sage: [[V.K_on_basis(i, b, -1) for i in V.index_set()] for b in C]
[[B[[[1], [2]]], 1/q*B[[[1], [2]]], B[[[1], [2]]]],
 [1/q*B[[[1], [3]]], q*B[[[1], [3]]], 1/q*B[[[1], [3]]]],
 [q*B[[[2], [3]]], B[[[2], [3]]], 1/q*B[[[2], [3]]]],

```

(continues on next page)

(continued from previous page)

```
[1/q*B[[[1], [4]]], B[[[1], [4]]], q*B[[[1], [4]]]],
[q*B[[[2], [4]]], 1/q*B[[[2], [4]]], q*B[[[2], [4]]]],
[B[[[3], [4]]], q*B[[[3], [4]]], B[[[3], [4]]]]]
```

```
>>> from sage.all import *
>>> from sage.algebras.quantum_groups.representations import _
↳ MinusculaRepresentation
>>> C = crystals.Tableaux(['A', Integer(3)], shape=[Integer(1), Integer(1)])
>>> R = ZZ['q'].fraction_field()
>>> V = MinusculaRepresentation(R, C)
>>> [[V.K_on_basis(i, b) for i in V.index_set()] for b in C]
[[B[[[1], [2]]], q*B[[[1], [2]]], B[[[1], [2]]]],
 [q*B[[[1], [3]]], 1/q*B[[[1], [3]]], q*B[[[1], [3]]]],
 [1/q*B[[[2], [3]]], B[[[2], [3]]], q*B[[[2], [3]]]],
 [q*B[[[1], [4]]], B[[[1], [4]]], 1/q*B[[[1], [4]]]],
 [1/q*B[[[2], [4]]], q*B[[[2], [4]]], 1/q*B[[[2], [4]]]],
 [B[[[3], [4]]], 1/q*B[[[3], [4]]], B[[[3], [4]]]]]
>>> [[V.K_on_basis(i, b, -Integer(1)) for i in V.index_set()] for b in C]
[[B[[[1], [2]]], 1/q*B[[[1], [2]]], B[[[1], [2]]]],
 [1/q*B[[[1], [3]]], q*B[[[1], [3]]], 1/q*B[[[1], [3]]]],
 [q*B[[[2], [3]]], B[[[2], [3]]], 1/q*B[[[2], [3]]]],
 [1/q*B[[[1], [4]]], B[[[1], [4]]], q*B[[[1], [4]]]],
 [q*B[[[2], [4]]], 1/q*B[[[2], [4]]], q*B[[[2], [4]]]],
 [B[[[3], [4]]], q*B[[[3], [4]]], B[[[3], [4]]]]]
```

cartan_type()

Return the Cartan type of self.

EXAMPLES:

```
sage: from sage.algebras.quantum_groups.representations import _
↳ AdjointRepresentation
sage: C = crystals.Tableaux(['C', 3], shape=[1])
sage: R = ZZ['q'].fraction_field()
sage: V = AdjointRepresentation(R, C)
sage: V.cartan_type()
['C', 3]
```

```
>>> from sage.all import *
>>> from sage.algebras.quantum_groups.representations import _
↳ AdjointRepresentation
>>> C = crystals.Tableaux(['C', Integer(3)], shape=[Integer(1)])
>>> R = ZZ['q'].fraction_field()
>>> V = AdjointRepresentation(R, C)
>>> V.cartan_type()
['C', 3]
```


FREE ASSOCIATIVE ALGEBRAS AND QUOTIENTS

3.1 Free algebras

AUTHORS:

- David Kohel (2005-09)
- William Stein (2006-11-01): add all doctests; implemented many things.
- Simon King (2011-04): Put free algebras into the category framework. Reimplement free algebra constructor, using a [UniqueFactory](#) for handling different implementations of free algebras. Allow degree weights for free algebras in letterplace implementation.

EXAMPLES:

```
sage: F = FreeAlgebra(ZZ, 3, 'x,y,z')
sage: F.base_ring()
Integer Ring
sage: G = FreeAlgebra(F, 2, 'm,n'); G
Free Algebra on 2 generators (m, n) over
  Free Algebra on 3 generators (x, y, z) over Integer Ring
sage: G.base_ring()
Free Algebra on 3 generators (x, y, z) over Integer Ring
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(ZZ, Integer(3), 'x,y,z')
>>> F.base_ring()
Integer Ring
>>> G = FreeAlgebra(F, Integer(2), 'm,n'); G
Free Algebra on 2 generators (m, n) over
  Free Algebra on 3 generators (x, y, z) over Integer Ring
>>> G.base_ring()
Free Algebra on 3 generators (x, y, z) over Integer Ring
```

The above free algebra is based on a generic implementation. By [Issue #7797](#), there is a different implementation [FreeAlgebra_letterplace](#) based on Singular's letterplace rings. It is currently restricted to weighted homogeneous elements and is therefore not the default. But the arithmetic is much faster than in the generic implementation. Moreover, we can compute Groebner bases with degree bound for its two-sided ideals, and thus provide ideal containment tests:

```
sage: # needs sage.libs.singular
sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace'); F
Free Associative Unital Algebra on 3 generators (x, y, z) over Rational Field
sage: I = F*[x*y+y*z, x^2+x*y-y*x-y^2]*F
```

(continues on next page)

(continued from previous page)

```

sage: I.groebner_basis(degbound=4)
Twosided Ideal (x*y + y*z,
               x*x - y*x - y*y - y*z,
               y*y*y - y*y*z + y*z*y - y*z*z,
               y*y*x + y*y*z + y*z*x + y*z*z,
               y*y*z*y - y*y*z*z + y*z*z*y - y*z*z*z,
               y*z*y*y - y*z*y*z + y*z*z*y - y*z*z*z,
               y*y*z*x + y*y*z*z + y*z*z*x + y*z*z*z,
               y*z*y*x + y*z*y*z + y*z*z*x + y*z*z*z) of Free Associative Unital
Algebra on 3 generators (x, y, z) over Rational Field
sage: y*z*y*y*z*z + 2*y*z*y*z*z*x + y*z*y*z*z*z - y*z*z*y*z*x + y*z*z*z*z*x in I
True

```

```

>>> from sage.all import *
>>> # needs sage.libs.singular
>>> F = FreeAlgebra(QQ, implementation='letterplace', names=('x', 'y', 'z')); (x, y, z) = F._first_ngens(3); F
Free Associative Unital Algebra on 3 generators (x, y, z) over Rational Field
>>> I = F*[x*y+y*z, x**Integer(2)+x*y-y*x-y**Integer(2)]*F
>>> I.groebner_basis(degbound=Integer(4))
Twosided Ideal (x*y + y*z,
               x*x - y*x - y*y - y*z,
               y*y*y - y*y*z + y*z*y - y*z*z,
               y*y*x + y*y*z + y*z*x + y*z*z,
               y*y*z*y - y*y*z*z + y*z*z*y - y*z*z*z,
               y*z*y*y - y*z*y*z + y*z*z*y - y*z*z*z,
               y*y*z*x + y*y*z*z + y*z*z*x + y*z*z*z,
               y*z*y*x + y*z*y*z + y*z*z*x + y*z*z*z) of Free Associative Unital
Algebra on 3 generators (x, y, z) over Rational Field
>>> y*z*y*y*z*z + Integer(2)*y*z*y*z*z*x + y*z*y*z*z*z - y*z*z*y*z*x + y*z*z*z*z*x in I
True

```

Positive integral degree weights for the letterplace implementation was introduced in [Issue #7797](#):

```

sage: # needs sage.libs.singular
sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace', degrees=[2,1,3])
sage: x.degree()
2
sage: y.degree()
1
sage: z.degree()
3
sage: I = F*[x*y-y*x, x^2+2*y*z, (x*y)^2-z^2]*F
sage: Q.<a,b,c> = F.quo(I)
sage: TestSuite(Q).run()
sage: a^2*b^2
c*c

```

```

>>> from sage.all import *
>>> # needs sage.libs.singular
>>> F = FreeAlgebra(QQ, implementation='letterplace', degrees=[Integer(2), Integer(1),

```

(continues on next page)

(continued from previous page)

```

↪Integer(3)], names=('x', 'y', 'z',)); (x, y, z,) = F._first_ngens(3)
>>> x.degree()
2
>>> y.degree()
1
>>> z.degree()
3
>>> I = F*[x*y-y*x, x**Integer(2)+Integer(2)*y*z, (x*y)**Integer(2)-z**Integer(2)]*F
>>> Q = F.quo(I, names=('a', 'b', 'c',)); (a, b, c,) = Q._first_ngens(3)
>>> TestSuite(Q).run()
>>> a**Integer(2)*b**Integer(2)
c*c

```

class sage.algebras.free_algebra.**AssociativeFunctor**(vars, degs=None)

Bases: [ConstructionFunctor](#)

A constructor for free associative algebras.

EXAMPLES:

```

sage: P = algebras.Free(ZZ, 2, 'x,y')
sage: x,y = P.gens()
sage: F = P.construction()[0]; F
Associative[x,y]

sage: A = GF(5)['a,b']
sage: a, b = A.gens()
sage: F(A)
Free Algebra on 2 generators (x, y) over Multivariate Polynomial Ring in a, b
↪over Finite Field of size 5

sage: f = A.hom([a+b,a-b],A)
sage: F(f)
Generic endomorphism of Free Algebra on 2 generators (x, y)
over Multivariate Polynomial Ring in a, b over Finite Field of size 5

sage: F(f)(a * F(A)(x))
(a+b)*x

```

```

>>> from sage.all import *
>>> P = algebras.Free(ZZ, Integer(2), 'x,y')
>>> x,y = P.gens()
>>> F = P.construction()[Integer(0)]; F
Associative[x,y]

>>> A = GF(Integer(5))['a,b']
>>> a, b = A.gens()
>>> F(A)
Free Algebra on 2 generators (x, y) over Multivariate Polynomial Ring in a, b
↪over Finite Field of size 5

>>> f = A.hom([a+b,a-b],A)
>>> F(f)

```

(continues on next page)

(continued from previous page)

```
Generic endomorphism of Free Algebra on 2 generators (x, y)
over Multivariate Polynomial Ring in a, b over Finite Field of size 5

>>> F(f) (a * F(A) (x))
(a+b)*x
```

merge (*other*)

Merge self with another construction functor, or return None.

EXAMPLES:

```
sage: from sage.algebras.free_algebra import AssociativeFunctor
sage: F = AssociativeFunctor(['x','y'])
sage: G = AssociativeFunctor(['t'])
sage: F.merge(G)
Associative[x,y,t]
sage: F.merge(F)
Associative[x,y]
```

```
>>> from sage.all import *
>>> from sage.algebras.free_algebra import AssociativeFunctor
>>> F = AssociativeFunctor(['x','y'])
>>> G = AssociativeFunctor(['t'])
>>> F.merge(G)
Associative[x,y,t]
>>> F.merge(F)
Associative[x,y]
```

With degrees:

```
sage: F = AssociativeFunctor(['x','y'], (2,3))
sage: G = AssociativeFunctor(['t'], (4,))
sage: H = AssociativeFunctor(['z','y'], (5,3))
sage: F.merge(G)
Associative[x,y,t] with degrees (2, 3, 4)
sage: F.merge(H)
Associative[x,y,z] with degrees (2, 3, 5)
```

```
>>> from sage.all import *
>>> F = AssociativeFunctor(['x','y'], (Integer(2),Integer(3)))
>>> G = AssociativeFunctor(['t'], (Integer(4),))
>>> H = AssociativeFunctor(['z','y'], (Integer(5),Integer(3)))
>>> F.merge(G)
Associative[x,y,t] with degrees (2, 3, 4)
>>> F.merge(H)
Associative[x,y,z] with degrees (2, 3, 5)
```

Now some actual use cases:

```
sage: R = algebras.Free(ZZ, 3, 'x,y,z')
sage: x,y,z = R.gens()
sage: 1/2 * x
```

(continues on next page)

(continued from previous page)

```

1/2*x
sage: parent(1/2 * x)
Free Algebra on 3 generators (x, y, z) over Rational Field

sage: S = algebras.Free(QQ, 2, 'z,t')
sage: z,t = S.gens()
sage: x + t
t + x
sage: parent(x + t)
Free Algebra on 4 generators (z, t, x, y) over Rational Field

```

```

>>> from sage.all import *
>>> R = algebras.Free(ZZ, Integer(3), 'x,y,z')
>>> x,y,z = R.gens()
>>> Integer(1)/Integer(2) * x
1/2*x
>>> parent(Integer(1)/Integer(2) * x)
Free Algebra on 3 generators (x, y, z) over Rational Field

>>> S = algebras.Free(QQ, Integer(2), 'z,t')
>>> z,t = S.gens()
>>> x + t
t + x
>>> parent(x + t)
Free Algebra on 4 generators (z, t, x, y) over Rational Field

```

rank = 9

class sage.algebras.free_algebra.**FreeAlgebraFactory**

Bases: [UniqueFactory](#)

A constructor of free algebras.

See [free_algebra](#) for examples and corner cases.

EXAMPLES:

```

sage: FreeAlgebra(GF(5),3,'x')
Free Algebra on 3 generators (x0, x1, x2) over Finite Field of size 5
sage: F.<x,y,z> = FreeAlgebra(GF(5),3)
sage: (x+y+z)^2
x^2 + x*y + x*z + y*x + y^2 + y*z + z*x + z*y + z^2
sage: FreeAlgebra(GF(5),3, 'xx, zba, Y')
Free Algebra on 3 generators (xx, zba, Y) over Finite Field of size 5
sage: FreeAlgebra(GF(5),3, 'abc')
Free Algebra on 3 generators (a, b, c) over Finite Field of size 5
sage: FreeAlgebra(GF(5),1, 'z')
Free Algebra on 1 generator (z,) over Finite Field of size 5
sage: FreeAlgebra(GF(5),1, ['alpha'])
Free Algebra on 1 generator (alpha,) over Finite Field of size 5
sage: FreeAlgebra(FreeAlgebra(ZZ,1,'a'), 2, 'x')
Free Algebra on 2 generators (x0, x1) over
Free Algebra on 1 generator (a,) over Integer Ring

```

```

>>> from sage.all import *
>>> FreeAlgebra(GF(Integer(5)), Integer(3), 'x')
Free Algebra on 3 generators (x0, x1, x2) over Finite Field of size 5
>>> F = FreeAlgebra(GF(Integer(5)), Integer(3), names=('x', 'y', 'z',)); (x, y, z,
↪) = F._first_ngens(3)
>>> (x+y+z)**Integer(2)
x^2 + x*y + x*z + y*x + y^2 + y*z + z*x + z*y + z^2
>>> FreeAlgebra(GF(Integer(5)), Integer(3), 'xx, zba, Y')
Free Algebra on 3 generators (xx, zba, Y) over Finite Field of size 5
>>> FreeAlgebra(GF(Integer(5)), Integer(3), 'abc')
Free Algebra on 3 generators (a, b, c) over Finite Field of size 5
>>> FreeAlgebra(GF(Integer(5)), Integer(1), 'z')
Free Algebra on 1 generator (z,) over Finite Field of size 5
>>> FreeAlgebra(GF(Integer(5)), Integer(1), ['alpha'])
Free Algebra on 1 generator (alpha,) over Finite Field of size 5
>>> FreeAlgebra(FreeAlgebra(ZZ, Integer(1), 'a'), Integer(2), 'x')
Free Algebra on 2 generators (x0, x1) over
Free Algebra on 1 generator (a,) over Integer Ring

```

Free algebras are globally unique:

```

sage: F = FreeAlgebra(ZZ, 3, 'x,y,z')
sage: G = FreeAlgebra(ZZ, 3, 'x,y,z')
sage: F is G
True
sage: F.<x,y,z> = FreeAlgebra(GF(5), 3) # indirect doctest
sage: F is loads(dumps(F))
True
sage: F is FreeAlgebra(GF(5), ['x','y','z'])
True
sage: copy(F) is F is loads(dumps(F))
True
sage: TestSuite(F).run()

```

```

>>> from sage.all import *
>>> F = FreeAlgebra(ZZ, Integer(3), 'x,y,z')
>>> G = FreeAlgebra(ZZ, Integer(3), 'x,y,z')
>>> F is G
True
>>> F = FreeAlgebra(GF(Integer(5)), Integer(3), names=('x', 'y', 'z',)); (x, y, z,
↪) = F._first_ngens(3) # indirect doctest
>>> F is loads(dumps(F))
True
>>> F is FreeAlgebra(GF(Integer(5)), ['x','y','z'])
True
>>> copy(F) is F is loads(dumps(F))
True
>>> TestSuite(F).run()

```

By [Issue #7797](#), we provide a different implementation of free algebras, based on Singular’s “letterplace rings”. Our letterplace wrapper allows for choosing positive integral degree weights for the generators of the free algebra. However, only (weighted) homogeneous elements are supported. Of course, isomorphic algebras in different implementations are not identical:

```

sage: # needs sage.libs.singular
sage: G = FreeAlgebra(GF(5), ['x', 'y', 'z'], implementation='letterplace')
sage: F == G
False
sage: G is FreeAlgebra(GF(5), ['x', 'y', 'z'], implementation='letterplace')
True
sage: copy(G) is G is loads(dumps(G))
True
sage: TestSuite(G).run()

```

```

>>> from sage.all import *
>>> # needs sage.libs.singular
>>> G = FreeAlgebra(GF(Integer(5)), ['x', 'y', 'z'], implementation='letterplace')
>>> F == G
False
>>> G is FreeAlgebra(GF(Integer(5)), ['x', 'y', 'z'], implementation='letterplace')
True
>>> copy(G) is G is loads(dumps(G))
True
>>> TestSuite(G).run()

```

```

sage: # needs sage.libs.singular
sage: H = FreeAlgebra(GF(5), ['x', 'y', 'z'], implementation='letterplace',
.....:                degrees=[1,2,3])
sage: F != H != G
True
sage: H is FreeAlgebra(GF(5), ['x', 'y', 'z'], implementation='letterplace',
.....:                degrees=[1,2,3])
True
sage: copy(H) is H is loads(dumps(H))
True
sage: TestSuite(H).run()

```

```

>>> from sage.all import *
>>> # needs sage.libs.singular
>>> H = FreeAlgebra(GF(Integer(5)), ['x', 'y', 'z'], implementation='letterplace',
...                degrees=[Integer(1), Integer(2), Integer(3)])
>>> F != H != G
True
>>> H is FreeAlgebra(GF(Integer(5)), ['x', 'y', 'z'], implementation='letterplace',
...                degrees=[Integer(1), Integer(2), Integer(3)])
True
>>> copy(H) is H is loads(dumps(H))
True
>>> TestSuite(H).run()

```

Free algebras commute with their base ring.

```

sage: K.<a,b> = FreeAlgebra(QQ,2)
sage: K.is_commutative()
False
sage: L.<c> = FreeAlgebra(K,1)

```

(continues on next page)

(continued from previous page)

```

sage: L.is_commutative()
False
sage: s = a*b^2 * c^3; s
a*b^2*c^3
sage: parent(s)
Free Algebra on 1 generator (c,) over
  Free Algebra on 2 generators (a, b) over Rational Field
sage: c^3 * a * b^2
a*b^2*c^3

```

```

>>> from sage.all import *
>>> K = FreeAlgebra(QQ,Integer(2), names=('a', 'b',)); (a, b,) = K._first_ngens(2)
>>> K.is_commutative()
False
>>> L = FreeAlgebra(K,Integer(1), names=('c',)); (c,) = L._first_ngens(1)
>>> L.is_commutative()
False
>>> s = a*b**Integer(2) * c**Integer(3); s
a*b^2*c^3
>>> parent(s)
Free Algebra on 1 generator (c,) over
  Free Algebra on 2 generators (a, b) over Rational Field
>>> c**Integer(3) * a * b**Integer(2)
a*b^2*c^3

```

create_key (*base_ring*, *arg1=None*, *arg2=None*, *sparse=None*, *order=None*, *names=None*, *name=None*, *implementation=None*, *degrees=None*)

Create the key under which a free algebra is stored.

create_object (*version*, *key*)

Construct the free algebra that belongs to a unique key.

NOTE:

Of course, that method should not be called directly, since it does not use the cache of free algebras.

class sage.algebras.free_algebra.**FreeAlgebra_generic** (*R*, *n*, *names*, *degrees=None*)

Bases: `CombinatorialFreeModule`

The free algebra on n generators over a base ring.

INPUT:

- R – a ring
- n – integer
- *names* – the generator names
- *degrees* – (optional) a tuple or list specifying the degrees of all the generators, if omitted, the algebra is not graded

EXAMPLES:

```

sage: F.<x,y,z> = FreeAlgebra(QQ, 3); F
Free Algebra on 3 generators (x, y, z) over Rational Field
sage: mul(F.gens())

```

(continues on next page)

(continued from previous page)

```

x*y*z
sage: mul([ F.gen(i%3) for i in range(12) ])
x*y*z*x*y*z*x*y*z*x*y*z
sage: mul([ F.gen(i%3) for i in range(12) ]) + mul([ F.gen(i%2) for i in _
↳range(12) ])
x*y*x*y*x*y*x*y*x*y*x*y + x*y*z*x*y*z*x*y*z*x*y*z
sage: (2 + x*z + x^2)^2 + (x - y)^2
4 + 5*x^2 - x*y + 4*x*z - y*x + y^2 + x^4 + x^3*z + x*z*x^2 + x*z*x*z

```

```

>>> from sage.all import *
>>> F = FreeAlgebra(QQ, Integer(3), names=('x', 'y', 'z')); (x, y, z,) = F._
↳first_ngens(3); F
Free Algebra on 3 generators (x, y, z) over Rational Field
>>> mul(F.gens())
x*y*z
>>> mul([ F.gen(i%Integer(3)) for i in range(Integer(12)) ])
x*y*z*x*y*z*x*y*z*x*y*z
>>> mul([ F.gen(i%Integer(3)) for i in range(Integer(12)) ]) + mul([ F.gen(i
↳%Integer(2)) for i in range(Integer(12)) ])
x*y*x*y*x*y*x*y*x*y*x*y + x*y*z*x*y*z*x*y*z*x*y*z
>>> (Integer(2) + x*z + x**Integer(2))**Integer(2) + (x - y)**Integer(2)
4 + 5*x^2 - x*y + 4*x*z - y*x + y^2 + x^4 + x^3*z + x*z*x^2 + x*z*x*z

```

Elementalias of *FreeAlgebraElement***algebra_generators()**

Return the algebra generators of self.

EXAMPLES:

```

sage: F = FreeAlgebra(ZZ, 3, 'x, y, z')
sage: F.algebra_generators()
Finite family {'x': x, 'y': y, 'z': z}

```

```

>>> from sage.all import *
>>> F = FreeAlgebra(ZZ, Integer(3), 'x, y, z')
>>> F.algebra_generators()
Finite family {'x': x, 'y': y, 'z': z}

```

construction()

Return the construction of self.

EXAMPLES:

```

sage: F, R = algebras.Free(QQ, 4, 'x, y, z, t').construction(); F
Associative[x, y, z, t]

```

```

>>> from sage.all import *
>>> F, R = algebras.Free(QQ, Integer(4), 'x, y, z, t').construction(); F
Associative[x, y, z, t]

```

degree_on_basis(*m*)

Return the degree of the basis element indexed by *m*.

EXAMPLES:

```
sage: A.<a, b> = FreeAlgebra(QQ, degrees=(1, -1))
sage: m = A.basis().keys()[42]
sage: m
a*b*a*b^2
sage: A.degree_on_basis(m)
-1
sage: (a*b*a*b^2).degree()
-1
```

```
>>> from sage.all import *
>>> A = FreeAlgebra(QQ, degrees=(Integer(1), -Integer(1)), names=('a', 'b',));
↳ (a, b,) = A._first_ngens(2)
>>> m = A.basis().keys()[Integer(42)]
>>> m
a*b*a*b^2
>>> A.degree_on_basis(m)
-1
>>> (a*b*a*b**Integer(2)).degree()
-1
```

g_algebra(*relations*, *names=None*, *order='degrevlex'*, *check=True*)

The *G*-Algebra derived from this algebra by relations.

By default it is assumed that any two variables commute.

Todo

- Coercion doesn't work yet, there is some cheating about assumptions
- The optional argument *check* controls checking the degeneracy conditions. Furthermore, the default values interfere with non-degeneracy conditions.

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: A.<x, y, z> = FreeAlgebra(QQ, 3)
sage: G = A.g_algebra({y*x: -x*y})
sage: (x, y, z) = G.gens()
sage: x*y
x*y
sage: y*x
-x*y
sage: z*x
x*z
sage: (x, y, z) = A.gens()
sage: G = A.g_algebra({y*x: -x*y + 1})
sage: (x, y, z) = G.gens()
sage: y*x
```

(continues on next page)

(continued from previous page)

```

-x*y + 1
sage: (x,y,z) = A.gens()
sage: G = A.g_algebra({y*x: -x*y + z})
sage: (x,y,z) = G.gens()
sage: y*x
-x*y + z

```

```

>>> from sage.all import *
>>> # needs sage.libs.singular
>>> A = FreeAlgebra(QQ,Integer(3), names=('x', 'y', 'z',)); (x, y, z,) = A._
↳first_ngens(3)
>>> G = A.g_algebra({y*x: -x*y})
>>> (x,y,z) = G.gens()
>>> x*y
x*y
>>> y*x
-x*y
>>> z*x
x*z
>>> (x,y,z) = A.gens()
>>> G = A.g_algebra({y*x: -x*y + Integer(1)})
>>> (x,y,z) = G.gens()
>>> y*x
-x*y + 1
>>> (x,y,z) = A.gens()
>>> G = A.g_algebra({y*x: -x*y + z})
>>> (x,y,z) = G.gens()
>>> y*x
-x*y + z

```

gen(*i*)

The *i*-th generator of the algebra.

EXAMPLES:

```

sage: F = FreeAlgebra(ZZ,3,'x,y,z')
sage: F.gen(0)
x

```

```

>>> from sage.all import *
>>> F = FreeAlgebra(ZZ,Integer(3),'x,y,z')
>>> F.gen(Integer(0))
x

```

gens()

Return the generators of self.

EXAMPLES:

```

sage: F = FreeAlgebra(ZZ,3,'x,y,z')
sage: F.gens()
(x, y, z)

```

```
>>> from sage.all import *
>>> F = FreeAlgebra(ZZ, Integer(3), 'x, y, z')
>>> F.gens()
(x, y, z)
```

is_field(*proof=True*)

Return True if this Free Algebra is a field.

This happens only if the base ring is a field and there are no generators

EXAMPLES:

```
sage: A = FreeAlgebra(QQ, 0, '')
sage: A.is_field()
True
sage: A = FreeAlgebra(QQ, 1, 'x')
sage: A.is_field()
False
```

```
>>> from sage.all import *
>>> A = FreeAlgebra(QQ, Integer(0), '')
>>> A.is_field()
True
>>> A = FreeAlgebra(QQ, Integer(1), 'x')
>>> A.is_field()
False
```

lie_polynomial(*w*)

Return the Lie polynomial associated to the Lyndon word w . If w is not Lyndon, then return the product of Lie polynomials of the Lyndon factorization of w .

Given a Lyndon word w , the Lie polynomial L_w is defined recursively by $L_w = [L_u, L_v]$, where $w = uv$ is the `standard factorization` of w , and $L_w = w$ when w is a single letter.

INPUT:

- w – a word or an element of the free monoid

EXAMPLES:

```
sage: F = FreeAlgebra(QQ, 3, 'x, y, z')
sage: M.<x, y, z> = FreeMonoid(3)
sage: F.lie_polynomial(x*y)
x*y - y*x
sage: F.lie_polynomial(y*x)
y*x
sage: F.lie_polynomial(x^2*y*x)
x^2*y*x - 2*x*y*x^2 + y*x^3
sage: F.lie_polynomial(y*z*x*z*x*z)
y*z*x*z*x*z - y*z*x*z^2*x - y*z^2*x^2*z + y*z^2*x*z*x
- z*y*x*z*x*z + z*y*x*z^2*x + z*y*z*x^2*z - z*y*z*x*z*x
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, Integer(3), 'x, y, z')
>>> M = FreeMonoid(Integer(3), names=('x', 'y', 'z')); (x, y, z,) = M._first_
```

(continues on next page)

(continued from previous page)

```

↪ngens(3)
>>> F.lie_polynomial(x*y)
x*y - y*x
>>> F.lie_polynomial(y*x)
y*x
>>> F.lie_polynomial(x**Integer(2)*y*x)
x^2*y*x - 2*x*y*x^2 + y*x^3
>>> F.lie_polynomial(y*z*x*z*x*z)
y*z*x*z*x*z - y*z*x*z^2*x - y*z^2*x^2*z + y*z^2*x*z*x
- z*y*x*z*x*z + z*y*x*z^2*x + z*y*z*x^2*z - z*y*z*x*z*x

```

monoid()

The free monoid of generators of the algebra.

EXAMPLES:

```

sage: F = FreeAlgebra(ZZ, 3, 'x, y, z')
sage: F.monoid()
Free monoid on 3 generators (x, y, z)

```

```

>>> from sage.all import *
>>> F = FreeAlgebra(ZZ, Integer(3), 'x, y, z')
>>> F.monoid()
Free monoid on 3 generators (x, y, z)

```

ngens()

The number of generators of the algebra.

EXAMPLES:

```

sage: F = FreeAlgebra(ZZ, 3, 'x, y, z')
sage: F.ngens()
3

```

```

>>> from sage.all import *
>>> F = FreeAlgebra(ZZ, Integer(3), 'x, y, z')
>>> F.ngens()
3

```

one_basis()

Return the index of the basis element 1.

EXAMPLES:

```

sage: F = FreeAlgebra(QQ, 2, 'x, y')
sage: F.one_basis()
1
sage: F.one_basis().parent()
Free monoid on 2 generators (x, y)

```

```

>>> from sage.all import *
>>> F = FreeAlgebra(QQ, Integer(2), 'x, y')

```

(continues on next page)

(continued from previous page)

```
>>> F.one_basis()
1
>>> F.one_basis().parent()
Free monoid on 2 generators (x, y)
```

pbw_basis()

Return the Poincaré-Birkhoff-Witt (PBW) basis of `self`.

EXAMPLES:

```
sage: F.<x,y> = FreeAlgebra(QQ, 2)
sage: F.poincare_birkhoff_witt_basis()
The Poincare-Birkhoff-Witt basis of Free Algebra on 2 generators (x, y) over
↳ Rational Field
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, Integer(2), names=('x', 'y')); (x, y) = F._first_
↳ ngens(2)
>>> F.poincare_birkhoff_witt_basis()
The Poincare-Birkhoff-Witt basis of Free Algebra on 2 generators (x, y) over
↳ Rational Field
```

pbw_element(elt)

Return the element `elt` in the Poincaré-Birkhoff-Witt basis.

EXAMPLES:

```
sage: F.<x,y> = FreeAlgebra(QQ, 2)
sage: F.pbw_element(x*y - y*x + 2)
2*PBW[1] + PBW[x*y]
sage: F.pbw_element(F.one())
PBW[1]
sage: F.pbw_element(x*y*x + x^3*y)
PBW[x*y]*PBW[x] + PBW[y]*PBW[x]^2 + PBW[x^3*y]
+ 3*PBW[x^2*y]*PBW[x] + 3*PBW[x*y]*PBW[x]^2 + PBW[y]*PBW[x]^3
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, Integer(2), names=('x', 'y')); (x, y) = F._first_
↳ ngens(2)
>>> F.pbw_element(x*y - y*x + Integer(2))
2*PBW[1] + PBW[x*y]
>>> F.pbw_element(F.one())
PBW[1]
>>> F.pbw_element(x*y*x + x**Integer(3)*y)
PBW[x*y]*PBW[x] + PBW[y]*PBW[x]^2 + PBW[x^3*y]
+ 3*PBW[x^2*y]*PBW[x] + 3*PBW[x*y]*PBW[x]^2 + PBW[y]*PBW[x]^3
```

poincare_birkhoff_witt_basis()

Return the Poincaré-Birkhoff-Witt (PBW) basis of `self`.

EXAMPLES:

```
sage: F.<x,y> = FreeAlgebra(QQ, 2)
sage: F.poincare_birkhoff_witt_basis()
The Poincare-Birkhoff-Witt basis of Free Algebra on 2 generators (x, y) over
↳Rational Field
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, Integer(2), names=('x', 'y',)); (x, y,) = F._first_
↳ngens(2)
>>> F.poincare_birkhoff_witt_basis()
The Poincare-Birkhoff-Witt basis of Free Algebra on 2 generators (x, y) over
↳Rational Field
```

product_on_basis(x, y)

Return the product of the basis elements indexed by x and y .

EXAMPLES:

```
sage: F = FreeAlgebra(ZZ, 3, 'x,y,z')
sage: I = F.basis().keys()
sage: x,y,z = I.gens()
sage: F.product_on_basis(x*y, z*y)
x*y*z*y
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(ZZ, Integer(3), 'x,y,z')
>>> I = F.basis().keys()
>>> x,y,z = I.gens()
>>> F.product_on_basis(x*y, z*y)
x*y*z*y
```

quo($mons, mats=None, names=None, **args$)

Return a quotient algebra.

The quotient algebra is defined via the action of a free algebra A on a (finitely generated) free module. The input for the quotient algebra is a list of monomials (in the underlying monoid for A) which form a free basis for the module of A , and a list of matrices, which give the action of the free generators of A on this monomial basis.

EXAMPLES:

Here is the quaternion algebra defined in terms of three generators:

```
sage: n = 3
sage: A = FreeAlgebra(QQ,n, 'i')
sage: F = A.monoid()
sage: i, j, k = F.gens()
sage: mons = [ F(1), i, j, k ]
sage: M = MatrixSpace(QQ,4)
sage: mats = [M([0,1,0,0, -1,0,0,0, 0,0,0,-1, 0,0,1,0]),
....:         M([0,0,1,0, 0,0,0,1, -1,0,0,0, 0,-1,0,0]),
....:         M([0,0,0,1, 0,0,-1,0, 0,1,0,0, -1,0,0,0]) ]
sage: H.<i,j,k> = A.quotient(mons, mats); H
Free algebra quotient on 3 generators ('i', 'j', 'k') and dimension 4
over Rational Field
```

```

>>> from sage.all import *
>>> n = Integer(3)
>>> A = FreeAlgebra(QQ,n,'i')
>>> F = A.monoid()
>>> i, j, k = F.gens()
>>> mons = [ F(Integer(1)), i, j, k ]
>>> M = MatrixSpace(QQ,Integer(4))
>>> mats = [M([Integer(0),Integer(1),Integer(0),Integer(0), -Integer(1),
↳Integer(0),Integer(0),Integer(0), Integer(0),Integer(0),Integer(0),-
↳Integer(1), Integer(0),Integer(0),Integer(1),Integer(0)]),
...      M([Integer(0),Integer(0),Integer(1),Integer(0), Integer(0),
↳Integer(0),Integer(0),Integer(1), -Integer(1),Integer(0),Integer(0),
↳Integer(0), Integer(0),-Integer(1),Integer(0),Integer(0)]),
...      M([Integer(0),Integer(0),Integer(0),Integer(1), Integer(0),
↳Integer(0),-Integer(1),Integer(0), Integer(0),Integer(1),Integer(0),
↳Integer(0), -Integer(1),Integer(0),Integer(0),Integer(0)]) ]
>>> H = A.quotient(mons, mats, names=('i', 'j', 'k')); (i, j, k) = H._first_
↳ngens(3); H
Free algebra quotient on 3 generators ('i', 'j', 'k') and dimension 4
over Rational Field

```

quotient (*mons*, *mats*=None, *names*=None, ***args*)

Return a quotient algebra.

The quotient algebra is defined via the action of a free algebra A on a (finitely generated) free module. The input for the quotient algebra is a list of monomials (in the underlying monoid for A) which form a free basis for the module of A , and a list of matrices, which give the action of the free generators of A on this monomial basis.

EXAMPLES:

Here is the quaternion algebra defined in terms of three generators:

```

sage: n = 3
sage: A = FreeAlgebra(QQ,n,'i')
sage: F = A.monoid()
sage: i, j, k = F.gens()
sage: mons = [ F(1), i, j, k ]
sage: M = MatrixSpace(QQ,4)
sage: mats = [M([0,1,0,0, -1,0,0,0, 0,0,0,-1, 0,0,1,0]),
....:      M([0,0,1,0, 0,0,0,1, -1,0,0,0, 0,-1,0,0]),
....:      M([0,0,0,1, 0,0,-1,0, 0,1,0,0, -1,0,0,0]) ]
sage: H.<i,j,k> = A.quotient(mons, mats); H
Free algebra quotient on 3 generators ('i', 'j', 'k') and dimension 4
over Rational Field

```

```

>>> from sage.all import *
>>> n = Integer(3)
>>> A = FreeAlgebra(QQ,n,'i')
>>> F = A.monoid()
>>> i, j, k = F.gens()
>>> mons = [ F(Integer(1)), i, j, k ]
>>> M = MatrixSpace(QQ,Integer(4))
>>> mats = [M([Integer(0),Integer(1),Integer(0),Integer(0), -Integer(1),

```

(continues on next page)

(continued from previous page)

```

↪Integer(0), Integer(0), Integer(0), Integer(0), Integer(0), Integer(0), -
↪Integer(1), Integer(0), Integer(0), Integer(1), Integer(0)]],
...      M([Integer(0), Integer(0), Integer(1), Integer(0), Integer(0),
↪Integer(0), Integer(0), Integer(1), -Integer(1), Integer(0), Integer(0),
↪Integer(0), Integer(0), -Integer(1), Integer(0), Integer(0)]),
...      M([Integer(0), Integer(0), Integer(0), Integer(1), Integer(0),
↪Integer(0), -Integer(1), Integer(0), Integer(0), Integer(1), Integer(0),
↪Integer(0), -Integer(1), Integer(0), Integer(0), Integer(0)]) ]
>>> H = A.quotient(mons, mats, names=('i', 'j', 'k')); (i, j, k) = H._first_
↪ngens(3); H
Free algebra quotient on 3 generators ('i', 'j', 'k') and dimension 4
over Rational Field

```

class sage.algebras.free_algebra.PBWBasisOfFreeAlgebra(*alg*)

Bases: CombinatorialFreeModule

The Poincaré-Birkhoff-Witt basis of the free algebra.

EXAMPLES:

```

sage: F.<x,y> = FreeAlgebra(QQ, 2)
sage: PBW = F.pbw_basis()
sage: px, py = PBW.gens()
sage: px * py
PBW[x*y] + PBW[y]*PBW[x]
sage: py * px
PBW[y]*PBW[x]
sage: px * py^3 * px - 2*px * py
-2*PBW[x*y] - 2*PBW[y]*PBW[x] + PBW[x*y^3]*PBW[x]
+ 3*PBW[y]*PBW[x*y^2]*PBW[x] + 3*PBW[y]^2*PBW[x*y]*PBW[x]
+ PBW[y]^3*PBW[x]^2

```

```

>>> from sage.all import *
>>> F = FreeAlgebra(QQ, Integer(2), names=('x', 'y')); (x, y) = F._first_
↪ngens(2)
>>> PBW = F.pbw_basis()
>>> px, py = PBW.gens()
>>> px * py
PBW[x*y] + PBW[y]*PBW[x]
>>> py * px
PBW[y]*PBW[x]
>>> px * py**Integer(3) * px - Integer(2)*px * py
-2*PBW[x*y] - 2*PBW[y]*PBW[x] + PBW[x*y^3]*PBW[x]
+ 3*PBW[y]*PBW[x*y^2]*PBW[x] + 3*PBW[y]^2*PBW[x*y]*PBW[x]
+ PBW[y]^3*PBW[x]^2

```

We can convert between the two bases:

```

sage: p = PBW(x*y - y*x + 2); p
2*PBW[1] + PBW[x*y]
sage: F(p)
2 + x*y - y*x
sage: f = F.pbw_element(x*y*x + x^3*y + x + 3)

```

(continues on next page)

(continued from previous page)

```

sage: F(PBW(f)) == f
True
sage: p = px*py + py^4*px^2
sage: F(p)
x*y + y^4*x^2
sage: PBW(F(p)) == p
True

```

```

>>> from sage.all import *
>>> p = PBW(x*y - y*x + Integer(2)); p
2*PBW[1] + PBW[x*y]
>>> F(p)
2 + x*y - y*x
>>> f = F.pbw_element(x*y*x + x**Integer(3)*y + x + Integer(3))
>>> F(PBW(f)) == f
True
>>> p = px*py + py**Integer(4)*px**Integer(2)
>>> F(p)
x*y + y^4*x^2
>>> PBW(F(p)) == p
True

```

Note that multiplication in the PBW basis agrees with multiplication as monomials:

```

sage: F(px * py^3 * px - 2*px * py) == x*y^3*x - 2*x*y
True

```

```

>>> from sage.all import *
>>> F(px * py**Integer(3) * px - Integer(2)*px * py) == x*y**Integer(3)*x -
↪ Integer(2)*x*y
True

```

We verify Examples 1 and 2 in [MR1989]:

```

sage: F.<x,y,z> = FreeAlgebra(QQ)
sage: PBW = F.pbw_basis()
sage: PBW(x*y*z)
PBW[x*y*z] + PBW[x*z*y] + PBW[y]*PBW[x*z] + PBW[y*z]*PBW[x]
+ PBW[z]*PBW[x*y] + PBW[z]*PBW[y]*PBW[x]
sage: PBW(x*y*y*x)
PBW[x*y^2]*PBW[x] + 2*PBW[y]*PBW[x*y]*PBW[x] + PBW[y]^2*PBW[x]^2

```

```

>>> from sage.all import *
>>> F = FreeAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z,) = F._first_ngens(3)
>>> PBW = F.pbw_basis()
>>> PBW(x*y*z)
PBW[x*y*z] + PBW[x*z*y] + PBW[y]*PBW[x*z] + PBW[y*z]*PBW[x]
+ PBW[z]*PBW[x*y] + PBW[z]*PBW[y]*PBW[x]
>>> PBW(x*y*y*x)
PBW[x*y^2]*PBW[x] + 2*PBW[y]*PBW[x*y]*PBW[x] + PBW[y]^2*PBW[x]^2

```

```

class Element

```


Bases: `IndexedFreeModuleElement`

expand()

Expand `self` in the monomials of the free algebra.

EXAMPLES:

```
sage: F = FreeAlgebra(QQ, 2, 'x,y')
sage: PBW = F.pbw_basis()
sage: x,y = F.monoid().gens()
sage: f = PBW(x^2*y) + PBW(x) + PBW(y^4*x)
sage: f.expand()
x + x^2*y - 2*x*y*x + y*x^2 + y^4*x
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, Integer(2), 'x,y')
>>> PBW = F.pbw_basis()
>>> x,y = F.monoid().gens()
>>> f = PBW(x**Integer(2)*y) + PBW(x) + PBW(y**Integer(4)*x)
>>> f.expand()
x + x^2*y - 2*x*y*x + y*x^2 + y^4*x
```

algebra_generators()

Return the generators of `self` as an algebra.

EXAMPLES:

```
sage: PBW = FreeAlgebra(QQ, 2, 'x,y').pbw_basis()
sage: gens = PBW.algebra_generators(); gens
(PBW[x], PBW[y])
sage: all(g.parent() is PBW for g in gens)
True
```

```
>>> from sage.all import *
>>> PBW = FreeAlgebra(QQ, Integer(2), 'x,y').pbw_basis()
>>> gens = PBW.algebra_generators(); gens
(PBW[x], PBW[y])
>>> all(g.parent() is PBW for g in gens)
True
```

expansion(t)

Return the expansion of the element `t` of the Poincaré-Birkhoff-Witt basis in the monomials of the free algebra.

EXAMPLES:

```
sage: F = FreeAlgebra(QQ, 2, 'x,y')
sage: PBW = F.pbw_basis()
sage: x,y = F.monoid().gens()
sage: PBW.expansion(PBW(x*y))
x*y - y*x
sage: PBW.expansion(PBW.one())
1
```

(continues on next page)

(continued from previous page)

```
sage: PBW.expansion(PBW(x*y*x) + 2*PBW(x) + 3)
3 + 2*x + x*y*x - y*x^2
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, Integer(2), 'x,y')
>>> PBW = F.pbw_basis()
>>> x,y = F.monoid().gens()
>>> PBW.expansion(PBW(x*y))
x*y - y*x
>>> PBW.expansion(PBW.one())
1
>>> PBW.expansion(PBW(x*y*x) + Integer(2)*PBW(x) + Integer(3))
3 + 2*x + x*y*x - y*x^2
```

free_algebra()

Return the associated free algebra of self.

EXAMPLES:

```
sage: PBW = FreeAlgebra(QQ, 2, 'x,y').pbw_basis()
sage: PBW.free_algebra()
Free Algebra on 2 generators (x, y) over Rational Field
```

```
>>> from sage.all import *
>>> PBW = FreeAlgebra(QQ, Integer(2), 'x,y').pbw_basis()
>>> PBW.free_algebra()
Free Algebra on 2 generators (x, y) over Rational Field
```

gen(i)

Return the i-th generator of self.

EXAMPLES:

```
sage: PBW = FreeAlgebra(QQ, 2, 'x,y').pbw_basis()
sage: PBW.gen(0)
PBW[x]
sage: PBW.gen(1)
PBW[y]
```

```
>>> from sage.all import *
>>> PBW = FreeAlgebra(QQ, Integer(2), 'x,y').pbw_basis()
>>> PBW.gen(Integer(0))
PBW[x]
>>> PBW.gen(Integer(1))
PBW[y]
```

gens()

Return the generators of self as an algebra.

EXAMPLES:

```
sage: PBW = FreeAlgebra(QQ, 2, 'x,y').pbw_basis()
sage: gens = PBW.algebra_generators(); gens
```

(continues on next page)

(continued from previous page)

```
(PBW[x], PBW[y])
sage: all(g.parent() is PBW for g in gens)
True
```

```
>>> from sage.all import *
>>> PBW = FreeAlgebra(QQ, Integer(2), 'x,y').pbw_basis()
>>> gens = PBW.algebra_generators(); gens
(PBW[x], PBW[y])
>>> all(g.parent() is PBW for g in gens)
True
```

one_basis()

Return the index of the basis element for 1.

EXAMPLES:

```
sage: PBW = FreeAlgebra(QQ, 2, 'x,y').pbw_basis()
sage: PBW.one_basis()
1
sage: PBW.one_basis().parent()
Free monoid on 2 generators (x, y)
```

```
>>> from sage.all import *
>>> PBW = FreeAlgebra(QQ, Integer(2), 'x,y').pbw_basis()
>>> PBW.one_basis()
1
>>> PBW.one_basis().parent()
Free monoid on 2 generators (x, y)
```

product(u, v)

Return the product of two elements u and v.

EXAMPLES:

```
sage: F = FreeAlgebra(QQ, 2, 'x,y')
sage: PBW = F.pbw_basis()
sage: x, y = PBW.gens()
sage: PBW.product(x, y)
PBW[x*y] + PBW[y]*PBW[x]
sage: PBW.product(y, x)
PBW[y]*PBW[x]
sage: PBW.product(y^2*x, x*y*x)
PBW[y]^2*PBW[x^2*y]*PBW[x] + 2*PBW[y]^2*PBW[x*y]*PBW[x]^2 + PBW[y]^3*PBW[x]^3
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, Integer(2), 'x,y')
>>> PBW = F.pbw_basis()
>>> x, y = PBW.gens()
>>> PBW.product(x, y)
PBW[x*y] + PBW[y]*PBW[x]
>>> PBW.product(y, x)
PBW[y]*PBW[x]
```

(continues on next page)

(continued from previous page)

```
>>> PBW.product(y**Integer(2)*x, x*y*x)
PBW[y]^2*PBW[x^2*y]*PBW[x] + 2*PBW[y]^2*PBW[x*y]*PBW[x]^2 + PBW[y]^3*PBW[x]^3
```

```
sage.algebras.free_algebra.is_FreeAlgebra(x)
```

Return True if x is a free algebra; otherwise, return False.

EXAMPLES:

```
sage: from sage.algebras.free_algebra import is_FreeAlgebra
sage: is_FreeAlgebra(5)
doctest:warning...
DeprecationWarning: the function is_FreeAlgebra is deprecated;
use 'isinstance(..., (FreeAlgebra_generic, FreeAlgebra_letterplace))' instead
See https://github.com/sagemath/sage/issues/37896 for details.
False
sage: is_FreeAlgebra(ZZ)
False
sage: is_FreeAlgebra(FreeAlgebra(ZZ,100,'x'))
True
sage: is_FreeAlgebra(FreeAlgebra(ZZ,10,'x',implementation='letterplace'))
True
sage: is_FreeAlgebra(FreeAlgebra(ZZ,10,'x',implementation='letterplace',
....:                        degrees=list(range(1,11))))
True
```

```
>>> from sage.all import *
>>> from sage.algebras.free_algebra import is_FreeAlgebra
>>> is_FreeAlgebra(Integer(5))
doctest:warning...
DeprecationWarning: the function is_FreeAlgebra is deprecated;
use 'isinstance(..., (FreeAlgebra_generic, FreeAlgebra_letterplace))' instead
See https://github.com/sagemath/sage/issues/37896 for details.
False
>>> is_FreeAlgebra(ZZ)
False
>>> is_FreeAlgebra(FreeAlgebra(ZZ,Integer(100),'x'))
True
>>> is_FreeAlgebra(FreeAlgebra(ZZ,Integer(10),'x',implementation='letterplace'))
True
>>> is_FreeAlgebra(FreeAlgebra(ZZ,Integer(10),'x',implementation='letterplace',
...                        degrees=list(range(Integer(1),Integer(11)))))
True
```

3.2 Free algebra elements

AUTHORS:

- David Kohel (2005-09)

```
class sage.algebras.free_algebra_element.FreeAlgebraElement(A,x)
```

Bases: `IndexedFreeModuleElement`, `AlgebraElement`

A free algebra element.

is_unit()

Return True if self is invertible.

EXAMPLES:

```
sage: A.<x, y, z> = FreeAlgebra(ZZ)
sage: A(-1).is_unit()
True
sage: A(2).is_unit()
False
sage: A(1 + x).is_unit()
False
sage: A.<x, y> = FreeAlgebra(QQ, degrees=(1,-1))
sage: A(x * y).is_unit()
False
sage: A(2).is_unit()
True
```

```
>>> from sage.all import *
>>> A = FreeAlgebra(ZZ, names=('x', 'y', 'z')); (x, y, z) = A._first_
↳ngens(3)
>>> A(-Integer(1)).is_unit()
True
>>> A(Integer(2)).is_unit()
False
>>> A(Integer(1) + x).is_unit()
False
>>> A = FreeAlgebra(QQ, degrees=(Integer(1), -Integer(1)), names=('x', 'y'));
↳(x, y) = A._first_ngens(2)
>>> A(x * y).is_unit()
False
>>> A(Integer(2)).is_unit()
True
```

to_pbw_basis()

Return self in the Poincaré-Birkhoff-Witt (PBW) basis.

EXAMPLES:

```
sage: F.<x,y,z> = FreeAlgebra(ZZ, 3)
sage: p = x^2*y + 3*y*x + 2
sage: p.to_pbw_basis()
2*PBW[1] + 3*PBW[y]*PBW[x] + PBW[x^2*y]
+ 2*PBW[x*y]*PBW[x] + PBW[y]*PBW[x]^2
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(ZZ, Integer(3), names=('x', 'y', 'z')); (x, y, z) = F._
↳first_ngens(3)
>>> p = x**Integer(2)*y + Integer(3)*y*x + Integer(2)
>>> p.to_pbw_basis()
2*PBW[1] + 3*PBW[y]*PBW[x] + PBW[x^2*y]
+ 2*PBW[x*y]*PBW[x] + PBW[y]*PBW[x]^2
```

variables()

Return the variables used in `self`.

EXAMPLES:

```
sage: A.<x,y,z> = FreeAlgebra(ZZ,3)
sage: elt = x + x*y + x^3*y
sage: elt.variables()
[x, y]
sage: elt = x + x^2 - x^4
sage: elt.variables()
[x]
sage: elt = x + z*y + z*x
sage: elt.variables()
[x, y, z]
```

```
>>> from sage.all import *
>>> A = FreeAlgebra(ZZ,Integer(3), names=('x', 'y', 'z')); (x, y, z) = A._
↳first_ngens(3)
>>> elt = x + x*y + x**Integer(3)*y
>>> elt.variables()
[x, y]
>>> elt = x + x**Integer(2) - x**Integer(4)
>>> elt.variables()
[x]
>>> elt = x + z*y + z*x
>>> elt.variables()
[x, y, z]
```

3.3 Free associative unital algebras, implemented via Singular's letterplace rings

AUTHOR:

- Simon King (2011-03-21): [Issue #7797](#)

With this implementation, Groebner bases out to a degree bound and normal forms can be computed for twosided weighted homogeneous ideals of free algebras. For now, all computations are restricted to weighted homogeneous elements, i.e., other elements cannot be created by arithmetic operations.

EXAMPLES:

```
sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace')
sage: F
Free Associative Unital Algebra on 3 generators (x, y, z) over Rational Field
sage: I = F*[x*y+y*z,x^2+x*y-y*x-y^2]*F
sage: I
Twosided Ideal (x*y + y*z, x*x + x*y - y*x - y*y) of Free Associative Unital Algebra
↳on 3 generators (x, y, z) over Rational Field
sage: x*(x*I.0-I.1*y+I.0*y)-I.1*y*z
x*y*x*y + x*y*y*y - x*y*y*z + x*y*z*y + y*x*y*z + y*y*y*z
sage: x^2*I.0-x*I.1*y+x*I.0*y-I.1*y*z in I
True
```

```

>>> from sage.all import *
>>> F = FreeAlgebra(QQ, implementation='letterplace', names=('x', 'y', 'z')); (x, y, z) = F._first_ngens(3)
>>> F
Free Associative Unital Algebra on 3 generators (x, y, z) over Rational Field
>>> I = F*[x*y+y*z,x**Integer(2)+x*y-y*x-y**Integer(2)]*F
>>> I
Twosided Ideal (x*y + y*z, x*x + x*y - y*x - y*y) of Free Associative Unital Algebra on 3 generators (x, y, z) over Rational Field
>>> x*(x*I.gen(0)-I.gen(1)*y+I.gen(0)*y)-I.gen(1)*y*z
x*y*x*y + x*y*y*y - x*y*y*z + x*y*z*y + y*x*y*z + y*y*y*z
>>> x**Integer(2)*I.gen(0)-x*I.gen(1)*y+x*I.gen(0)*y-I.gen(1)*y*z in I
True

```

The preceding containment test is based on the computation of Groebner bases with degree bound:

```

sage: I.groebner_basis(degbound=4)
Twosided Ideal (x*y + y*z,
  x*x - y*x - y*y - y*z,
  y*y*y - y*y*z + y*z*y - y*z*z,
  y*y*x + y*y*z + y*z*x + y*z*z,
  y*y*z*y - y*y*z*z + y*z*z*y - y*z*z*z,
  y*z*y*y - y*z*y*z + y*z*z*y - y*z*z*z,
  y*y*z*x + y*y*z*z + y*z*z*x + y*z*z*z,
  y*z*y*x + y*z*y*z + y*z*z*x + y*z*z*z) of Free Associative Unital
Algebra on 3 generators (x, y, z) over Rational Field

```

```

>>> from sage.all import *
>>> I.groebner_basis(degbound=Integer(4))
Twosided Ideal (x*y + y*z,
  x*x - y*x - y*y - y*z,
  y*y*y - y*y*z + y*z*y - y*z*z,
  y*y*x + y*y*z + y*z*x + y*z*z,
  y*y*z*y - y*y*z*z + y*z*z*y - y*z*z*z,
  y*z*y*y - y*z*y*z + y*z*z*y - y*z*z*z,
  y*y*z*x + y*y*z*z + y*z*z*x + y*z*z*z,
  y*z*y*x + y*z*y*z + y*z*z*x + y*z*z*z) of Free Associative Unital
Algebra on 3 generators (x, y, z) over Rational Field

```

When reducing an element by I , the original generators are chosen:

```

sage: (y*z*y*y).reduce(I)
y*z*y*y

```

```

>>> from sage.all import *
>>> (y*z*y*y).reduce(I)
y*z*y*y

```

However, there is a method for computing the normal form of an element, which is the same as reduction by the Groebner basis out to the degree of that element:

```

sage: (y*z*y*y).normal_form(I)
y*z*y*z - y*z*z*y + y*z*z*z

```

(continues on next page)

(continued from previous page)

```
sage: (y*z*y*y).reduce(I.groebner_basis(4))
y*z*y*z - y*z*z*y + y*z*z*z
```

```
>>> from sage.all import *
>>> (y*z*y*y).normal_form(I)
y*z*y*z - y*z*z*y + y*z*z*z
>>> (y*z*y*y).reduce(I.groebner_basis(Integer(4)))
y*z*y*z - y*z*z*y + y*z*z*z
```

The default term order derives from the degree reverse lexicographic order on the commutative version of the free algebra:

```
sage: F.commutative_ring().term_order()
Degree reverse lexicographic term order
```

```
>>> from sage.all import *
>>> F.commutative_ring().term_order()
Degree reverse lexicographic term order
```

A different term order can be chosen, and of course may yield a different normal form:

```
sage: L.<a,b,c> = FreeAlgebra(QQ, implementation='letterplace', order='lex')
sage: L.commutative_ring().term_order()
Lexicographic term order
sage: J = L*[a*b+b*c,a^2+a*b-b*c-c^2]*L
sage: J.groebner_basis(4)
Twosided Ideal (2*b*c*b - b*c*c + c*c*b,
a*b + b*c,
-a*c*c + 2*b*c*a + 2*b*c*c + c*c*a,
a*c*c*b - 2*b*c*c*b + b*c*c*c,
a*a - 2*b*c - c*c,
a*c*c*a - 2*b*c*c*a - 4*b*c*c*c - c*c*c*c) of Free Associative Unital
Algebra on 3 generators (a, b, c) over Rational Field
sage: (b*c*b*b).normal_form(J)
1/2*b*c*c*b - 1/2*c*c*b*b
```

```
>>> from sage.all import *
>>> L = FreeAlgebra(QQ, implementation='letterplace', order='lex', names=('a', 'b', 'c'
↪ ',')); (a, b, c,) = L._first_ngens(3)
>>> L.commutative_ring().term_order()
Lexicographic term order
>>> J = L*[a*b+b*c,a**Integer(2)+a*b-b*c-c**Integer(2)]*L
>>> J.groebner_basis(Integer(4))
Twosided Ideal (2*b*c*b - b*c*c + c*c*b,
a*b + b*c,
-a*c*c + 2*b*c*a + 2*b*c*c + c*c*a,
a*c*c*b - 2*b*c*c*b + b*c*c*c,
a*a - 2*b*c - c*c,
a*c*c*a - 2*b*c*c*a - 4*b*c*c*c - c*c*c*c) of Free Associative Unital
Algebra on 3 generators (a, b, c) over Rational Field
>>> (b*c*b*b).normal_form(J)
1/2*b*c*c*b - 1/2*c*c*b*b
```


Here is an example with degree weights:

```
sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace', degrees=[1,2,3])
sage: (x*y+z).degree()
3
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, implementation='letterplace', degrees=[Integer(1),Integer(2),
↳Integer(3)], names=('x', 'y', 'z',)); (x, y, z,) = F._first_ngens(3)
>>> (x*y+z).degree()
3
```

Todo

The computation of Groebner bases only works for global term orderings, and all elements must be weighted homogeneous with respect to positive integral degree weights. It is ongoing work in Singular to lift these restrictions.

We support coercion from the letterplace wrapper to the corresponding generic implementation of a free algebra (*FreeAlgebra_generic*), but there is no coercion in the opposite direction, since the generic implementation also comprises non-homogeneous elements.

We also do not support coercion from a subalgebra, or between free algebras with different term orderings, yet.

```
class sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace
```

Bases: *Parent*

Finitely generated free algebra, with arithmetic restricted to weighted homogeneous elements.

Note

The restriction to weighted homogeneous elements should be lifted as soon as the restriction to homogeneous elements is lifted in Singular's "Letterplace algebras".

EXAMPLES:

```
sage: K.<z> = GF(25)
sage: F.<a,b,c> = FreeAlgebra(K, implementation='letterplace')
sage: F
Free Associative Unital Algebra on 3 generators (a, b, c) over Finite Field in z
↳of size 5^2
sage: P = F.commutative_ring()
sage: P
Multivariate Polynomial Ring in a, b, c over Finite Field in z of size 5^2
```

```
>>> from sage.all import *
>>> K = GF(Integer(25), names=('z',)); (z,) = K._first_ngens(1)
>>> F = FreeAlgebra(K, implementation='letterplace', names=('a', 'b', 'c',)); (a,
↳b, c,) = F._first_ngens(3)
>>> F
Free Associative Unital Algebra on 3 generators (a, b, c) over Finite Field in z
↳of size 5^2
>>> P = F.commutative_ring()
```

(continues on next page)

(continued from previous page)

```
>>> P
Multivariate Polynomial Ring in a, b, c over Finite Field in z of size 5^2
```

We can do arithmetic as usual, as long as we stay (weighted) homogeneous:

```
sage: (z*a+(z+1)*b+2*c)^2
(z + 3)*a*a + (2*z + 3)*a*b + (2*z)*a*c + (2*z + 3)*b*a + (3*z + 4)*b*b + (2*z +
↪2)*b*c + (2*z)*c*a + (2*z + 2)*c*b - c*c
sage: a+1
Traceback (most recent call last):
...
ArithmeticError: can only add elements of the same weighted degree
```

```
>>> from sage.all import *
>>> (z*a+(z+Integer(1))*b+Integer(2)*c)**Integer(2)
(z + 3)*a*a + (2*z + 3)*a*b + (2*z)*a*c + (2*z + 3)*b*a + (3*z + 4)*b*b + (2*z +
↪2)*b*c + (2*z)*c*a + (2*z + 2)*c*b - c*c
>>> a+Integer(1)
Traceback (most recent call last):
...
ArithmeticError: can only add elements of the same weighted degree
```

commutative_ring()

Return the commutative version of this free algebra.

Note

This commutative ring is used as a unique key of the free algebra.

EXAMPLES:

```
sage: K.<z> = GF(25)
sage: F.<a,b,c> = FreeAlgebra(K, implementation='letterplace')
sage: F
Free Associative Unital Algebra on 3 generators (a, b, c) over Finite Field
↪in z of size 5^2
sage: F.commutative_ring()
Multivariate Polynomial Ring in a, b, c over Finite Field in z of size 5^2
sage: FreeAlgebra(F.commutative_ring()) is F
True
```

```
>>> from sage.all import *
>>> K = GF(Integer(25), names=('z',)); (z,) = K._first_ngens(1)
>>> F = FreeAlgebra(K, implementation='letterplace', names=('a', 'b', 'c',));
↪(a, b, c,) = F._first_ngens(3)
>>> F
Free Associative Unital Algebra on 3 generators (a, b, c) over Finite Field
↪in z of size 5^2
>>> F.commutative_ring()
Multivariate Polynomial Ring in a, b, c over Finite Field in z of size 5^2
```

(continues on next page)

(continued from previous page)

```
>>> FreeAlgebra(F.commutative_ring()) is F
True
```

current_ring()

Return the commutative ring that is used to emulate the non-commutative multiplication out to the current degree.

EXAMPLES:

```
sage: F.<a,b,c> = FreeAlgebra(QQ, implementation='letterplace')
sage: F.current_ring()
Multivariate Polynomial Ring in a, b, c over Rational Field
sage: a*b
a*b
sage: F.current_ring()
Multivariate Polynomial Ring in a, b, c, a_1, b_1, c_1 over Rational Field
sage: F.set_degbound(3)
sage: F.current_ring()
Multivariate Polynomial Ring in a, b, c, a_1, b_1, c_1, a_2, b_2, c_2 over
↪Rational Field
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, implementation='letterplace', names=('a', 'b', 'c'));
↪ (a, b, c) = F._first_ngens(3)
>>> F.current_ring()
Multivariate Polynomial Ring in a, b, c over Rational Field
>>> a*b
a*b
>>> F.current_ring()
Multivariate Polynomial Ring in a, b, c, a_1, b_1, c_1 over Rational Field
>>> F.set_degbound(Integer(3))
>>> F.current_ring()
Multivariate Polynomial Ring in a, b, c, a_1, b_1, c_1, a_2, b_2, c_2 over
↪Rational Field
```

degbound()

Return the degree bound that is currently used.

Note

When multiplying two elements of this free algebra, the degree bound will be dynamically adapted. It can also be set by `set_degbound()`.

EXAMPLES:

In order to avoid we get a free algebras from the cache that was created in another doctest and has a different degree bound, we choose a base ring that does not appear in other tests:

```
sage: F.<x,y,z> = FreeAlgebra(ZZ, implementation='letterplace')
sage: F.degbound()
1
sage: x*y
```

(continues on next page)

(continued from previous page)

```
x*y
sage: F.degbound()
2
sage: F.set_degbound(4)
sage: F.degbound()
4
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(ZZ, implementation='letterplace', names=('x', 'y', 'z'));
↪ (x, y, z) = F._first_ngens(3)
>>> F.degbound()
1
>>> x*y
x*y
>>> F.degbound()
2
>>> F.set_degbound(Integer(4))
>>> F.degbound()
4
```

gen(*i*)Return the *i*-th generator.

INPUT:

- *i* – integer

OUTPUT: the generator with index *i*

EXAMPLES:

```
sage: F.<a,b,c> = FreeAlgebra(QQ, implementation='letterplace')
sage: F.1 is F.1 # indirect doctest
True
sage: F.gen(2)
c
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, implementation='letterplace', names=('a', 'b', 'c'));
↪ (a, b, c) = F._first_ngens(3)
>>> F.gen(1) is F.gen(1) # indirect doctest
True
>>> F.gen(Integer(2))
c
```

generator_degrees()**gens()**

Return the tuple of generators.

EXAMPLES:

```
sage: F.<a,b,c> = FreeAlgebra(QQ, implementation='letterplace')
sage: F.gens()
(a, b, c)
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, implementation='letterplace', names=('a', 'b', 'c'));
↪ (a, b, c) = F._first_ngens(3)
>>> F.gens()
(a, b, c)
```

ideal_monoid()

Return the monoid of ideals of this free algebra.

EXAMPLES:

```
sage: F.<x,y> = FreeAlgebra(GF(2), implementation='letterplace')
sage: F.ideal_monoid()
Monoid of ideals of Free Associative Unital Algebra on 2 generators (x, y)
↪ over Finite Field of size 2
sage: F.ideal_monoid() is F.ideal_monoid()
True
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(GF(Integer(2)), implementation='letterplace', names=('x',
↪ 'y',)); (x, y,) = F._first_ngens(2)
>>> F.ideal_monoid()
Monoid of ideals of Free Associative Unital Algebra on 2 generators (x, y)
↪ over Finite Field of size 2
>>> F.ideal_monoid() is F.ideal_monoid()
True
```

is_field(proof=True)

Tell whether this free algebra is a field.

Note

This would only be the case in the degenerate case of no generators. But such an example cannot be constructed in this implementation.

ngens()

Return the number of generators.

EXAMPLES:

```
sage: F.<a,b,c> = FreeAlgebra(QQ, implementation='letterplace')
sage: F.ngens()
3
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, implementation='letterplace', names=('a', 'b', 'c'));
↪ (a, b, c,) = F._first_ngens(3)
>>> F.ngens()
3
```

set_degbound(d)

Increase the degree bound that is currently in place.

Note

The degree bound cannot be decreased.

EXAMPLES:

In order to avoid we get a free algebras from the cache that was created in another doctest and has a different degree bound, we choose a base ring that does not appear in other tests:

```
sage: F.<x,y,z> = FreeAlgebra(GF(251), implementation='letterplace')
sage: F.degbound()
1
sage: x*y
x*y
sage: F.degbound()
2
sage: F.set_degbound(4)
sage: F.degbound()
4
sage: F.set_degbound(2)
sage: F.degbound()
4
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(GF(Integer(251)), implementation='letterplace', names=('x
↳ ', 'y', 'z',)); (x, y, z,) = F._first_ngens(3)
>>> F.degbound()
1
>>> x*y
x*y
>>> F.degbound()
2
>>> F.set_degbound(Integer(4))
>>> F.degbound()
4
>>> F.set_degbound(Integer(2))
>>> F.degbound()
4
```

term_order_of_block()

Return the term order that is used for the commutative version of this free algebra.

EXAMPLES:

```
sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace')
sage: F.term_order_of_block()
Degree reverse lexicographic term order
sage: L.<a,b,c> = FreeAlgebra(QQ, implementation='letterplace', order='lex')
sage: L.term_order_of_block()
Lexicographic term order
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, implementation='letterplace', names=('x', 'y', 'z',));
```

(continues on next page)

(continued from previous page)

```

↪ (x, y, z) = F._first_ngens(3)
>>> F.term_order_of_block()
Degree reverse lexicographic term order
>>> L = FreeAlgebra(QQ, implementation='letterplace', order='lex', names=('a',
↪ 'b', 'c',)); (a, b, c) = L._first_ngens(3)
>>> L.term_order_of_block()
Lexicographic term order

```

classsage.algebras.letterplace.free_algebra_letterplace.**FreeAlgebra_letterplace_libsingular**

Bases: object

Internally used wrapper around a Singular Letterplace polynomial ring.

3.4 Weighted homogeneous elements of free algebras, in letterplace implementation

AUTHOR:

- Simon King (2011-03-23): Github issue [Issue #7797](#)

class sage.algebras.letterplace.free_algebra_element_letterplace.**FreeAlgebraElement_letterplace**Bases: `AlgebraElement`

Weighted homogeneous elements of a free associative unital algebra (letterplace implementation).

EXAMPLES:

```

sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace')
sage: x+y
x + y
sage: x*y !=y*x
True
sage: I = F*[x*y+y*z,x^2+x*y-y*x-y^2]*F
sage: (y^3).reduce(I)
y*y*y
sage: (y^3).normal_form(I)
y*y*z - y*z*y + y*z*z

```

```

>>> from sage.all import *
>>> F = FreeAlgebra(QQ, implementation='letterplace', names=('x', 'y', 'z',)); (x,
↪ y, z) = F._first_ngens(3)
>>> x+y
x + y
>>> x*y !=y*x
True
>>> I = F*[x*y+y*z,x**Integer(2)+x*y-y*x-y**Integer(2)]*F
>>> (y**Integer(3)).reduce(I)
y*y*y
>>> (y**Integer(3)).normal_form(I)
y*y*z - y*z*y + y*z*z

```

Here is an example with nontrivial degree weights:

```
sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace', degrees=[2,1,3])
sage: I = F*[x*y-y*x, x^2+2*y*z, (x*y)^2-z^2]*F
sage: x.degree()
2
sage: y.degree()
1
sage: z.degree()
3
sage: (x*y)^3
x*y*x*y*x*y
sage: ((x*y)^3).normal_form(I)
z*z*y*x
sage: ((x*y)^3).degree()
9
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, implementation='letterplace', degrees=[Integer(2),
↳ Integer(1), Integer(3)], names=('x', 'y', 'z')); (x, y, z,) = F._first_ngens(3)
>>> I = F*[x*y-y*x, x**Integer(2)+Integer(2)*y*z, (x*y)**Integer(2)-
↳ z**Integer(2)]*F
>>> x.degree()
2
>>> y.degree()
1
>>> z.degree()
3
>>> (x*y)**Integer(3)
x*y*x*y*x*y
>>> ((x*y)**Integer(3)).normal_form(I)
z*z*y*x
>>> ((x*y)**Integer(3)).degree()
9
```

degree()

Return the degree of this element.

Note

Generators may have a positive integral degree weight. All elements must be weighted homogeneous.

EXAMPLES:

```
sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace')
sage: ((x+y+z)^3).degree()
3
sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace', degrees=[2,1,
↳ 3])
sage: ((x*y+z)^3).degree()
9
```



```

>>> from sage.all import *
>>> F = FreeAlgebra(QQ, implementation='letterplace', names=('x', 'y', 'z'));
↪ (x, y, z) = F._first_ngens(3)
>>> ((x+y+z)**Integer(3)).degree()
3
>>> F = FreeAlgebra(QQ, implementation='letterplace', degrees=[Integer(2),
↪ Integer(1), Integer(3)], names=('x', 'y', 'z')); (x, y, z) = F._first_
↪ ngens(3)
>>> ((x*y+z)**Integer(3)).degree()
9

```

lc()

The leading coefficient of this free algebra element, as element of the base ring.

EXAMPLES:

```

sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace')
sage: ((2*x+3*y-4*z)^2*(5*y+6*z)).lc()
20
sage: ((2*x+3*y-4*z)^2*(5*y+6*z)).lc().parent() is F.base()
True
sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace', degrees=[2,1,
↪ 3])
sage: ((2*x*y+z)^2).lc()
4

```

```

>>> from sage.all import *
>>> F = FreeAlgebra(QQ, implementation='letterplace', names=('x', 'y', 'z'));
↪ (x, y, z) = F._first_ngens(3)
>>> ((Integer(2)*x+Integer(3)*y-
↪ Integer(4)*z)**Integer(2)*(Integer(5)*y+Integer(6)*z)).lc()
20
>>> ((Integer(2)*x+Integer(3)*y-
↪ Integer(4)*z)**Integer(2)*(Integer(5)*y+Integer(6)*z)).lc().parent() is F.
↪ base()
True
>>> F = FreeAlgebra(QQ, implementation='letterplace', degrees=[Integer(2),
↪ Integer(1), Integer(3)], names=('x', 'y', 'z')); (x, y, z) = F._first_
↪ ngens(3)
>>> ((Integer(2)*x*y+z)**Integer(2)).lc()
4

```

letterplace_polynomial()

Return the commutative polynomial that is used internally to represent this free algebra element.

EXAMPLES:

```

sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace')
sage: ((x+y-z)^2).letterplace_polynomial()
x*x_1 + x*y_1 - x*z_1 + y*x_1 + y*y_1 - y*z_1 - z*x_1 - z*y_1 + z*z_1

```

```

>>> from sage.all import *
>>> F = FreeAlgebra(QQ, implementation='letterplace', names=('x', 'y', 'z'));

```

(continues on next page)

(continued from previous page)

```

↪ (x, y, z) = F._first_ngens(3)
>>> ((x+y-z)**Integer(2)).letterplace_polynomial()
x*x_1 + x*y_1 - x*z_1 + y*x_1 + y*y_1 - y*z_1 - z*x_1 - z*y_1 + z*z_1

```

If degree weights are used, the letterplace polynomial is homogenized by slack variables:

```

sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace', degrees=[2,1,
↪ 3])
sage: ((x*y+z)^2).letterplace_polynomial()
x*x_1*y_2*x_3*x_4*y_5 + x*x_1*y_2*z_3*x_4*x_5 + z*x_1*x_2*x_3*x_4*y_5_
↪ + z*x_1*x_2*z_3*x_4*x_5

```

```

>>> from sage.all import *
>>> F = FreeAlgebra(QQ, implementation='letterplace', degrees=[Integer(2),
↪ Integer(1),Integer(3)], names=('x', 'y', 'z',)); (x, y, z) = F._first_
↪ ngens(3)
>>> ((x*y+z)**Integer(2)).letterplace_polynomial()
x*x_1*y_2*x_3*x_4*y_5 + x*x_1*y_2*z_3*x_4*x_5 + z*x_1*x_2*x_3*x_4*y_5_
↪ + z*x_1*x_2*z_3*x_4*x_5

```

lm()

The leading monomial of this free algebra element.

EXAMPLES:

```

sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace')
sage: ((2*x+3*y-4*z)^2*(5*y+6*z)).lm()
x*x*y
sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace', degrees=[2,1,
↪ 3])
sage: ((2*x*y+z)^2).lm()
x*y*x*y

```

```

>>> from sage.all import *
>>> F = FreeAlgebra(QQ, implementation='letterplace', names=('x', 'y', 'z',));
↪ (x, y, z) = F._first_ngens(3)
>>> ((Integer(2)*x+Integer(3)*y-
↪ Integer(4)*z)**Integer(2)*(Integer(5)*y+Integer(6)*z)).lm()
x*x*y
>>> F = FreeAlgebra(QQ, implementation='letterplace', degrees=[Integer(2),
↪ Integer(1),Integer(3)], names=('x', 'y', 'z',)); (x, y, z) = F._first_
↪ ngens(3)
>>> ((Integer(2)*x*y+z)**Integer(2)).lm()
x*y*x*y

```

lm_divides(p)

Tell whether or not the leading monomial of `self` divides the leading monomial of another element.

Note

A free algebra element p divides another one q if there are free algebra elements s and t such that $spt = q$.

EXAMPLES:

```

sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace', degrees=[2,1,
↪3])
sage: ((2*x*y+z)^2*z).lm()
x*y*x*y*z
sage: (y*x*y-y^4).lm()
y*x*y
sage: (y*x*y-y^4).lm_divides((2*x*y+z)^2*z)
True

```

```

>>> from sage.all import *
>>> F = FreeAlgebra(QQ, implementation='letterplace', degrees=[Integer(2),
↪Integer(1),Integer(3)], names=('x', 'y', 'z',)); (x, y, z,) = F._first_
↪ngens(3)
>>> ((Integer(2)*x*y+z)**Integer(2)*z).lm()
x*y*x*y*z
>>> (y*x*y-y**Integer(4)).lm()
y*x*y
>>> (y*x*y-y**Integer(4)).lm_divides((Integer(2)*x*y+z)**Integer(2)*z)
True

```

lt()

The leading term (monomial times coefficient) of this free algebra element.

EXAMPLES:

```

sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace')
sage: ((2*x+3*y-4*z)^2*(5*y+6*z)).lt()
20*x*x*y
sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace', degrees=[2,1,
↪3])
sage: ((2*x*y+z)^2).lt()
4*x*y*x*y

```

```

>>> from sage.all import *
>>> F = FreeAlgebra(QQ, implementation='letterplace', names=('x', 'y', 'z',));
↪ (x, y, z,) = F._first_ngens(3)
>>> ((Integer(2)*x+Integer(3)*y-
↪Integer(4)*z)**Integer(2)*(Integer(5)*y+Integer(6)*z)).lt()
20*x*x*y
>>> F = FreeAlgebra(QQ, implementation='letterplace', degrees=[Integer(2),
↪Integer(1),Integer(3)], names=('x', 'y', 'z',)); (x, y, z,) = F._first_
↪ngens(3)
>>> ((Integer(2)*x*y+z)**Integer(2)).lt()
4*x*y*x*y

```

normal_form(I)

Return the normal form of this element with respect to a twosided weighted homogeneous ideal.

INPUT:

- I – a twosided homogeneous ideal of the parent F of this element, x

OUTPUT: the normal form of x wrt. I

Note

The normal form is computed by reduction with respect to a Groebnerbasis of I with degree bound $\deg(x)$.

EXAMPLES:

```
sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace')
sage: I = F*[x*y+y*z, x^2+x*y-y*x-y^2]*F
sage: (x^5).normal_form(I)
-y*z*z*z*x - y*z*z*z*y - y*z*z*z*z
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, implementation='letterplace', names=('x', 'y', 'z'));
↪ (x, y, z) = F._first_ngens(3)
>>> I = F*[x*y+y*z, x**Integer(2)+x*y-y*x-y**Integer(2)]*F
>>> (x**Integer(5)).normal_form(I)
-y*z*z*z*x - y*z*z*z*y - y*z*z*z*z
```

We verify two basic properties of normal forms: The difference of an element and its normal form is contained in the ideal, and if two elements of the free algebra differ by an element of the ideal then they have the same normal form:

```
sage: x^5 - (x^5).normal_form(I) in I
True
sage: (x^5+x*I.0*y*z-3*z^2*I.1*y).normal_form(I) == (x^5).normal_form(I)
True
```

```
>>> from sage.all import *
>>> x**Integer(5) - (x**Integer(5)).normal_form(I) in I
True
>>> (x**Integer(5)+x*I.gen(0)*y*z-Integer(3)*z**Integer(2)*I.gen(1)*y).normal_
↪ form(I) == (x**Integer(5)).normal_form(I)
True
```

Here is an example with non-trivial degree weights:

```
sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace', degrees=[1,2,
↪ 3])
sage: I = F*[x*y-y*x+z, y^2+2*x*z, (x*y)^2-z^2]*F
sage: ((x*y)^3).normal_form(I)
z*z*y*x - z*z*z
sage: (x*y)^3-((x*y)^3).normal_form(I) in I
True
sage: ((x*y)^3+2*z*I.0*z+y*I.1*z-x*I.2*y).normal_form(I) == ((x*y)^3).normal_
↪ form(I)
True
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, implementation='letterplace', degrees=[Integer(1),
↪ Integer(2), Integer(3)], names=('x', 'y', 'z')); (x, y, z) = F._first_
↪ ngens(3)
```

(continues on next page)

(continued from previous page)

```

>>> I = F*[x*y-y*x+z, y**Integer(2)+Integer(2)*x*z, (x*y)**Integer(2)-
↳ z**Integer(2)]*F
>>> ((x*y)**Integer(3)).normal_form(I)
z*z*y*x - z*z*z
>>> (x*y)**Integer(3)-((x*y)**Integer(3)).normal_form(I) in I
True
>>> ((x*y)**Integer(3)+Integer(2)*z*I.gen(0)*z+y*I.gen(1)*z-x*I.gen(2)*y).
↳ normal_form(I) == ((x*y)**Integer(3)).normal_form(I)
True

```

reduce(*G*)

Reduce this element by a list of elements or by a twosided weighted homogeneous ideal.

INPUT:

Either a list or tuple of weighted homogeneous elements of the free algebra, or an ideal of the free algebra, or an ideal in the commutative polynomial ring that is currently used to implement the multiplication in the free algebra.

OUTPUT: the twosided reduction of this element by the argument

Note

This may not be the normal form of this element, unless the argument is a twosided Groebner basis up to the degree of this element.

EXAMPLES:

```

sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace')
sage: I = F*[x*y+y*z, x^2+x*y-y*x-y^2]*F
sage: p = y^2*z*y^2+y*z*y*z*y

```

```

>>> from sage.all import *
>>> F = FreeAlgebra(QQ, implementation='letterplace', names=('x', 'y', 'z',));
↳ (x, y, z,) = F._first_ngens(3)
>>> I = F*[x*y+y*z, x**Integer(2)+x*y-y*x-y**Integer(2)]*F
>>> p = y**Integer(2)*z*y**Integer(2)+y*z*y*z*y

```

We compute the letterplace version of the Groebner basis of *I* with degree bound 4:

```

sage: G = F._reductor_(I.groebner_basis(4).gens(), 4)
sage: G.ring() is F.current_ring()
True

```

```

>>> from sage.all import *
>>> G = F._reductor_(I.groebner_basis(Integer(4)).gens(), Integer(4))
>>> G.ring() is F.current_ring()
True

```

Since the element *p* is of degree 5, it is no surprise that its reductions with respect to the original generators of *I* (of degree 2), or with respect to *G* (Groebner basis with degree bound 4), or with respect to the Groebner basis with degree bound 5 (which yields its normal form) are pairwise different:

```

sage: p.reduce(I)
y*y*z*y*y + y*z*y*z*y
sage: p.reduce(G)
y*y*z*z*y + y*z*y*z*y - y*z*z*y*y + y*z*z*z*y
sage: p.normal_form(I)
y*y*z*z*z + y*z*y*z*z - y*z*z*y*z + y*z*z*z*z
sage: p.reduce(I) != p.reduce(G) != p.normal_form(I) != p.reduce(I)
True

```

```

>>> from sage.all import *
>>> p.reduce(I)
y*y*z*y*y + y*z*y*z*y
>>> p.reduce(G)
y*y*z*z*y + y*z*y*z*y - y*z*z*y*y + y*z*z*z*y
>>> p.normal_form(I)
y*y*z*z*z + y*z*y*z*z - y*z*z*y*z + y*z*z*z*z
>>> p.reduce(I) != p.reduce(G) != p.normal_form(I) != p.reduce(I)
True

```

3.5 Homogeneous ideals of free algebras

For twosided ideals and when the base ring is a field, this implementation also provides Groebner bases and ideal containment tests.

EXAMPLES:

```

sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace')
sage: F
Free Associative Unital Algebra on 3 generators (x, y, z) over Rational Field
sage: I = F*[x*y+y*z, x^2+x*y-y*x-y^2]*F
sage: I
Twosided Ideal (x*y + y*z, x*x + x*y - y*x - y*y) of Free Associative Unital Algebra
on 3 generators (x, y, z) over Rational Field

```

```

>>> from sage.all import *
>>> F = FreeAlgebra(QQ, implementation='letterplace', names=('x', 'y', 'z')); (x, y, z) = F._first_ngens(3)
>>> F
Free Associative Unital Algebra on 3 generators (x, y, z) over Rational Field
>>> I = F*[x*y+y*z, x**Integer(2)+x*y-y*x-y**Integer(2)]*F
>>> I
Twosided Ideal (x*y + y*z, x*x + x*y - y*x - y*y) of Free Associative Unital Algebra
on 3 generators (x, y, z) over Rational Field

```

One can compute Groebner bases out to a finite degree, can compute normal forms and can test containment in the ideal:

```

sage: I.groebner_basis(degbound=3)
Twosided Ideal (x*y + y*z,
x*x - y*x - y*y - y*z,
y*y*y - y*y*z + y*z*y - y*z*z,
y*y*x + y*y*z + y*z*x + y*z*z) of Free Associative Unital Algebra
on 3 generators (x, y, z) over Rational Field

```

(continues on next page)

(continued from previous page)

```
sage: (x*y*z*y*x).normal_form(I)
y*z*z*y*z + y*z*z*z*x + y*z*z*z*z
sage: x*y*z*y*x - (x*y*z*y*x).normal_form(I) in I
True
```

```
>>> from sage.all import *
>>> I.groebner_basis(degbound=Integer(3))
Twosided Ideal (x*y + y*z,
               x*x - y*x - y*y - y*z,
               y*y*y - y*y*z + y*z*y - y*z*z,
               y*y*x + y*y*z + y*z*x + y*z*z) of Free Associative Unital Algebra
               on 3 generators (x, y, z) over Rational Field
>>> (x*y*z*y*x).normal_form(I)
y*z*z*y*z + y*z*z*z*x + y*z*z*z*z
>>> x*y*z*y*x - (x*y*z*y*x).normal_form(I) in I
True
```

AUTHOR:

- Simon King (2011-03-22): See [Issue #7797](#).

```
class sage.algebras.letterplace.letterplace_ideal.LetterplaceIdeal (ring, gens, coerce=True,
                                                                    side='twosided')
```

Bases: `Ideal_nc`

Graded homogeneous ideals in free algebras.

In the two-sided case over a field, one can compute Groebner bases up to a degree bound, normal forms of graded homogeneous elements of the free algebra, and ideal containment.

EXAMPLES:

```
sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace')
sage: I = F*[x*y+y*z, x^2+x*y-y*x-y^2]*F
sage: I
Twosided Ideal (x*y + y*z, x*x + x*y - y*x - y*y) of Free Associative Unital
↳ Algebra on 3 generators (x, y, z) over Rational Field
sage: I.groebner_basis(2)
Twosided Ideal (x*y + y*z, x*x - y*x - y*y - y*z) of Free Associative Unital
↳ Algebra on 3 generators (x, y, z) over Rational Field
sage: I.groebner_basis(4)
Twosided Ideal (x*y + y*z,
               x*x - y*x - y*y - y*z,
               y*y*y - y*y*z + y*z*y - y*z*z,
               y*y*x + y*y*z + y*z*x + y*z*z,
               y*y*z*y - y*y*z*z + y*z*z*y - y*z*z*z,
               y*z*y*y - y*z*y*z + y*z*z*y - y*z*z*z,
               y*y*z*x + y*y*z*z + y*z*z*x + y*z*z*z,
               y*z*y*x + y*z*y*z + y*z*z*x + y*z*z*z) of Free Associative Unital
               Algebra on 3 generators (x, y, z) over Rational Field
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, implementation='letterplace', names=('x', 'y', 'z')); (x,
↳ y, z,) = F._first_ngens(3)
```

(continues on next page)

(continued from previous page)

```

>>> I = F*[x*y+y*z,x**Integer(2)+x*y-y*x-y**Integer(2)]*F
>>> I
Twosided Ideal (x*y + y*z, x*x + x*y - y*x - y*y) of Free Associative Unital
↪Algebra on 3 generators (x, y, z) over Rational Field
>>> I.groebner_basis(Integer(2))
Twosided Ideal (x*y + y*z, x*x - y*x - y*y - y*z) of Free Associative Unital
↪Algebra on 3 generators (x, y, z) over Rational Field
>>> I.groebner_basis(Integer(4))
Twosided Ideal (x*y + y*z,
    x*x - y*x - y*y - y*z,
    y*y*y - y*y*z + y*z*y - y*z*z,
    y*y*x + y*y*z + y*z*x + y*z*z,
    y*y*z*y - y*y*z*z + y*z*z*y - y*z*z*z,
    y*z*y*y - y*z*y*z + y*z*z*y - y*z*z*z,
    y*y*z*x + y*y*z*z + y*z*z*x + y*z*z*z,
    y*z*y*x + y*z*y*z + y*z*z*x + y*z*z*z) of Free Associative Unital
Algebra on 3 generators (x, y, z) over Rational Field

```

Groebner bases are cached. If one has computed a Groebner basis out to a high degree then it will also be returned if a Groebner basis with a lower degree bound is requested:

```

sage: I.groebner_basis(2) is I.groebner_basis(4)
True

```

```

>>> from sage.all import *
>>> I.groebner_basis(Integer(2)) is I.groebner_basis(Integer(4))
True

```

Of course, the normal form of any element has to satisfy the following:

```

sage: x*y*z*y*x - (x*y*z*y*x).normal_form(I) in I
True

```

```

>>> from sage.all import *
>>> x*y*z*y*x - (x*y*z*y*x).normal_form(I) in I
True

```

Left and right ideals can be constructed, but only twosided ideals provide Groebner bases:

```

sage: JL = F*[x*y+y*z,x^2+x*y-y*x-y^2]; JL
Left Ideal (x*y + y*z, x*x + x*y - y*x - y*y) of Free Associative Unital Algebra
↪on 3 generators (x, y, z) over Rational Field
sage: JR = [x*y+y*z,x^2+x*y-y*x-y^2]*F; JR
Right Ideal (x*y + y*z, x*x + x*y - y*x - y*y) of Free Associative Unital Algebra
↪on 3 generators (x, y, z) over Rational Field
sage: JR.groebner_basis(2)
Traceback (most recent call last):
...
TypeError: This ideal is not two-sided. We can only compute two-sided Groebner
↪bases
sage: JL.groebner_basis(2)
Traceback (most recent call last):

```

(continues on next page)

(continued from previous page)

```
...
TypeError: This ideal is not two-sided. We can only compute two-sided Groebner_
↳bases
```

```
>>> from sage.all import *
>>> JL = F*[x*y+y*z,x**Integer(2)+x*y-y*x-y**Integer(2)]; JL
Left Ideal (x*y + y*z, x*x + x*y - y*x - y*y) of Free Associative Unital Algebra_
↳on 3 generators (x, y, z) over Rational Field
>>> JR = [x*y+y*z,x**Integer(2)+x*y-y*x-y**Integer(2)]*F; JR
Right Ideal (x*y + y*z, x*x + x*y - y*x - y*y) of Free Associative Unital Algebra_
↳on 3 generators (x, y, z) over Rational Field
>>> JR.groebner_basis(Integer(2))
Traceback (most recent call last):
...
TypeError: This ideal is not two-sided. We can only compute two-sided Groebner_
↳bases
>>> JL.groebner_basis(Integer(2))
Traceback (most recent call last):
...
TypeError: This ideal is not two-sided. We can only compute two-sided Groebner_
↳bases
```

Also, it is currently not possible to compute a Groebner basis when the base ring is not a field:

```
sage: FZ.<a,b,c> = FreeAlgebra(ZZ, implementation='letterplace')
sage: J = FZ*[a^3-b^3]*FZ
sage: J.groebner_basis(2)
Traceback (most recent call last):
...
TypeError: Currently, we can only compute Groebner bases if the ring of_
↳coefficients is a field
```

```
>>> from sage.all import *
>>> FZ = FreeAlgebra(ZZ, implementation='letterplace', names=('a', 'b', 'c'));_
↳(a, b, c,) = FZ._first_ngens(3)
>>> J = FZ*[a**Integer(3)-b**Integer(3)]*FZ
>>> J.groebner_basis(Integer(2))
Traceback (most recent call last):
...
TypeError: Currently, we can only compute Groebner bases if the ring of_
↳coefficients is a field
```

The letterplace implementation of free algebras also provides integral degree weights for the generators, and we can compute Groebner bases for twosided graded homogeneous ideals:

```
sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace', degrees=[1,2,3])
sage: I = F*[x*y+z-y*x,x*y*z-x^6+y^3]*F
sage: I.groebner_basis(Infinity)
Twosided Ideal (x*y - y*x + z,
x*x*x*x*x*x - y*x*z - y*y*y + z*z,
x*z*z - y*x*x*z + y*x*z*x + y*y*z + y*z*y + z*x*z + z*y*y - z*z*x,
x*x*x*x*x*z + x*x*x*x*z*x + x*x*x*z*x*x + x*x*z*x*x*x + x*z*x*x*x*x +
```

(continues on next page)

(continued from previous page)

```

y**x*z*y - y*y*x*z + y*z*z + z*x*x*x*x*x - z*z*y,
x*x*x*x*z*y*y + x*x*x*z*y*y*x - x*x*x*z*y*z - x*x*z*y*x*z + x*x*z*y*y*x*x +
x*x*z*y*y*y - x*x*z*y*z*x - x*z*y*x*x*z - x*z*y*x*z*x +
x*z*y*y*x*x*x + 2*x*z*y*y*y*x - 2*x*z*y*y*z - x*z*y*z*x*x -
x*z*y*z*y + y*x*z*x*x*x*x - 4*y*x*z*x*x*z - 4*y*x*z*x*z*x +
4*y*x*z*y*x*x*x + 3*y*x*z*y*y*x - 4*y*x*z*y*z + y*y*x*x*x*x*z +
y*y*x*x*x*z*x - 3*y*y*x*x*z*x*x - y*y*x*x*z*y +
5*y*y*x*z*x*x*x + 4*y*y*x*z*y*x - 4*y*y*y*x*x*z +
4*y*y*y*x*z*x + 3*y*y*y*y*z + 4*y*y*y*z*x*x + 6*y*y*y*z*y +
y*y*z*x*x*x*x + y*y*z*x*z + 7*y*y*z*y*x*x + 7*y*y*z*y*y -
7*y*y*z*z*x - y*z*x*x*x*z - y*z*x*x*z*x + 3*y*z*x*z*x*x +
y*z*x*z*y + y*z*y*x*x*x*x - 3*y*z*y*x*z + 7*y*z*y*y*x*x +
3*y*z*y*y*y - 3*y*z*y*z*x - 5*y*z*z*x*x*x - 4*y*z*z*y*x +
4*y*z*z*z - z*y*x*x*x*z - z*y*x*x*z*x - z*y*x*z*x*x -
z*y*x*z*y + z*y*y*x*x*x*x - 3*z*y*y*x*z + 3*z*y*y*y*x*x +
z*y*y*y*y - 3*z*y*y*z*x - z*y*z*x*x*x - 2*z*y*z*y*x +
2*z*y*z*z - z*z*x*x*x*x*x + 4*z*z*x*x*z + 4*z*z*x*z*x -
4*z*z*y*x*x*x - 3*z*z*y*y*x + 4*z*z*y*z + 4*z*z*z*x*x +
2*z*z*z*y)
of Free Associative Unital Algebra on 3 generators (x, y, z) over Rational Field

```

```

>>> from sage.all import *
>>> F = FreeAlgebra(QQ, implementation='letterplace', degrees=[Integer(1),
↳ Integer(2), Integer(3)], names=('x', 'y', 'z')); (x, y, z) = F._first_ngens(3)
>>> I = F*[x*y+z-y*x, x*y*z-x**Integer(6)+y**Integer(3)]*F
>>> I.groebner_basis(Infinity)
Twosided Ideal (x*y - y*x + z,
x*x*x*x*x*x - y*x*z - y*y*y + z*z,
x*z*z - y*x*x*z + y*x*z*x + y*y*z + y*z*y + z*x*z + z*y*y - z*z*x,
x*x*x*x*x*x*z + x*x*x*x*x*z*x + x*x*x*z*x*x + x*x*z*x*x*x + x*z*x*x*x*x +
y*x*z*y - y*y*x*z + y*z*z + z*x*x*x*x*x - z*z*y,
x*x*x*x*z*y*y + x*x*x*z*y*y*x - x*x*x*z*y*z - x*x*z*y*x*z + x*x*z*y*y*x*x +
x*x*z*y*y*y - x*x*z*y*z*x - x*z*y*x*x*z - x*z*y*x*z*x +
x*z*y*y*x*x*x + 2*x*z*y*y*y*x - 2*x*z*y*y*z - x*z*y*z*x*x -
x*z*y*z*y + y*x*z*x*x*x*x*x - 4*y*x*z*x*x*z - 4*y*x*z*x*z*x +
4*y*x*z*y*x*x*x + 3*y*x*z*y*y*x - 4*y*x*z*y*z + y*y*x*x*x*x*z +
y*y*x*x*x*z*x - 3*y*y*x*x*z*x*x - y*y*x*x*z*y +
5*y*y*x*z*x*x*x + 4*y*y*x*z*y*x - 4*y*y*y*x*x*z +
4*y*y*y*x*z*x + 3*y*y*y*y*z + 4*y*y*y*z*x*x + 6*y*y*y*z*y +
y*y*z*x*x*x*x + y*y*z*x*z + 7*y*y*z*y*x*x + 7*y*y*z*y*y -
7*y*y*z*z*x - y*z*x*x*x*x*z - y*z*x*x*z*x + 3*y*z*x*z*x*x +
y*z*x*z*y + y*z*y*x*x*x*x - 3*y*z*y*x*z + 7*y*z*y*y*x*x +
3*y*z*y*y*y - 3*y*z*y*z*x - 5*y*z*z*x*x*x - 4*y*z*z*y*x +
4*y*z*z*z - z*y*x*x*x*x - z*y*x*x*z*x - z*y*x*z*x*x -
z*y*x*z*y + z*y*y*x*x*x*x - 3*z*y*y*x*z + 3*z*y*y*y*x*x +
z*y*y*y*y - 3*z*y*y*z*x - z*y*z*x*x*x - 2*z*y*z*y*x +
2*z*y*z*z - z*z*x*x*x*x*x + 4*z*z*x*x*z + 4*z*z*x*z*x -
4*z*z*y*x*x*x - 3*z*z*y*y*x + 4*z*z*y*z + 4*z*z*z*x*x +
2*z*z*z*y)
of Free Associative Unital Algebra on 3 generators (x, y, z) over Rational Field

```

Again, we can compute normal forms:

```
sage: (z*I.0-I.1).normal_form(I)
0
sage: (z*I.0-x*y*z).normal_form(I)
-y*x*z + z*z
```

```
>>> from sage.all import *
>>> (z*I.gen(0)-I.gen(1)).normal_form(I)
0
>>> (z*I.gen(0)-x*y*z).normal_form(I)
-y*x*z + z*z
```

groebner_basis (*degbound=None*)

Twosided Groebner basis with degree bound.

INPUT:

- *degbound* – (optional) integer or *Infinity*; if it is provided, a Groebner basis at least out to that degree is returned. By default, the current degree bound of the underlying ring is used.

ASSUMPTIONS:

Currently, we can only compute Groebner bases for twosided ideals, and the ring of coefficients must be a field. A *TypeError* is raised if one of these conditions is violated.

Note

- The result is cached. The same Groebner basis is returned if a smaller degree bound than the known one is requested.
- If the degree bound *Infinity* is requested, it is attempted to compute a complete Groebner basis. But we cannot guarantee that the computation will terminate, since not all twosided homogeneous ideals of a free algebra have a finite Groebner basis.

EXAMPLES:

```
sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace')
sage: I = F*[x*y+y*z,x^2+x*y-y*x-y^2]*F
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, implementation='letterplace', names=('x', 'y', 'z'));
↪ (x, y, z) = F._first_ngens(3)
>>> I = F*[x*y+y*z,x**Integer(2)+x*y-y*x-y**Integer(2)]*F
```

Since *F* was cached and since its degree bound cannot be decreased, it may happen that, as a side effect of other tests, it already has a degree bound bigger than 3. So, we cannot test against the output of *I.groebner_basis()*:

```
sage: F.set_degbound(3)
sage: I.groebner_basis() # not tested
Twosided Ideal (y*y*y - y*y*z + y*z*y - y*z*z, y*y*x + y*y*z + y*z*x + y*z*z,
↪ x*y + y*z, x*x - y*x - y*y - y*z) of Free Associative Unital Algebra on 3
↪ generators (x, y, z) over Rational Field
sage: I.groebner_basis(4)
```

(continues on next page)

(continued from previous page)

```

Twosided Ideal (x*y + y*z,
  x*x - y*x - y*y - y*z,
  y*y*y - y*y*z + y*z*y - y*z*z,
  y*y*x + y*y*z + y*z*x + y*z*z,
  y*y*z*y - y*y*z*z + y*z*z*y - y*z*z*z,
  y*z*y*y - y*z*y*z + y*z*z*y - y*z*z*z,
  y*y*z*x + y*y*z*z + y*z*z*x + y*z*z*z,
  y*z*y*x + y*z*y*z + y*z*z*x + y*z*z*z) of Free Associative
  Unital Algebra on 3 generators (x, y, z) over Rational Field
sage: I.groebner_basis(2) is I.groebner_basis(4)
True
sage: G = I.groebner_basis(4)
sage: G.groebner_basis(3) is G
True

```

```

>>> from sage.all import *
>>> F.set_degbound(Integer(3))
>>> I.groebner_basis() # not tested
Twosided Ideal (y*y*y - y*y*z + y*z*y - y*z*z, y*y*x + y*y*z + y*z*x + y*z*z,
  x*y + y*z, x*x - y*x - y*y - y*z) of Free Associative Unital Algebra on 3
  generators (x, y, z) over Rational Field
>>> I.groebner_basis(Integer(4))
Twosided Ideal (x*y + y*z,
  x*x - y*x - y*y - y*z,
  y*y*y - y*y*z + y*z*y - y*z*z,
  y*y*x + y*y*z + y*z*x + y*z*z,
  y*y*z*y - y*y*z*z + y*z*z*y - y*z*z*z,
  y*z*y*y - y*z*y*z + y*z*z*y - y*z*z*z,
  y*y*z*x + y*y*z*z + y*z*z*x + y*z*z*z,
  y*z*y*x + y*z*y*z + y*z*z*x + y*z*z*z) of Free Associative
  Unital Algebra on 3 generators (x, y, z) over Rational Field
>>> I.groebner_basis(Integer(2)) is I.groebner_basis(Integer(4))
True
>>> G = I.groebner_basis(Integer(4))
>>> G.groebner_basis(Integer(3)) is G
True

```

If a finite complete Groebner basis exists, we can compute it as follows:

```

sage: I = F*[x*y-y*x, x*z-z*x, y*z-z*y, x^2*y-z^3, x*y^2+z*x^2]*F
sage: I.groebner_basis(Infinity)
Twosided Ideal (-y*z + z*y,
  -x*z + z*x,
  -x*y + y*x,
  x*x*z + x*y*y,
  x*x*y - z*z*z,
  x*x*x*z + y*z*z*z,
  x*z*z*z*z + y*y*z*z*z) of Free Associative Unital Algebra
  on 3 generators (x, y, z) over Rational Field

```

```

>>> from sage.all import *
>>> I = F*[x*y-y*x, x*z-z*x, y*z-z*y, x**Integer(2)*y-z**Integer(3),

```

(continues on next page)

(continued from previous page)

```

↪x*y**Integer(2)+z*x**Integer(2)]*F
>>> I.groebner_basis(Infinity)
Twosided Ideal (-y*z + z*y,
  -x*z + z*x,
  -x*y + y*x,
  x*x*z + x*y*y,
  x*x*y - z*z*z,
  x*x*x*z + y*z*z*z,
  x*z*z*z*z + y*y*z*z*z) of Free Associative Unital Algebra
on 3 generators (x, y, z) over Rational Field

```

Since the commutators of the generators are contained in the ideal, we can verify the above result by a computation in a polynomial ring in negative lexicographic order:

```

sage: P.<c,b,a> = PolynomialRing(QQ,order='neglex')
sage: J = P*[a^2*b-c^3,a*b^2+c*a^2]
sage: J.groebner_basis()
[b*a^2 - c^3, b^2*a + c*a^2, c*a^3 + c^3*b, c^3*b^2 + c^4*a]

```

```

>>> from sage.all import *
>>> P = PolynomialRing(QQ,order='neglex', names=('c', 'b', 'a')); (c, b, a) ↪
↪= P._first_ngens(3)
>>> J = P*[a**Integer(2)*b-c**Integer(3),a*b**Integer(2)+c*a**Integer(2)]
>>> J.groebner_basis()
[b*a^2 - c^3, b^2*a + c*a^2, c*a^3 + c^3*b, c^3*b^2 + c^4*a]

```

Apparently, the results are compatible, by sending a to x , b to y and c to z .

reduce(G)

Reduction of this ideal by another ideal, or normal form of an algebra element with respect to this ideal.

INPUT:

- G – a list or tuple of elements, an ideal, the ambient algebra, or a single element

OUTPUT:

- The normal form of G with respect to this ideal, if G is an element of the algebra.
- The reduction of this ideal by the elements resp. generators of G , if G is a list, tuple or ideal.
- The zero ideal, if G is the algebra containing this ideal.

EXAMPLES:

```

sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace')
sage: I = F*[x*y+y*z,x^2+x*y-y*x-y^2]*F
sage: I.reduce(F)
Twosided Ideal (0) of Free Associative Unital Algebra on 3 generators (x, y, ↪
↪z) over Rational Field
sage: I.reduce(x^3)
-y*z*x - y*z*y - y*z*z
sage: I.reduce([x*y])
Twosided Ideal (y*z, x*x - y*x - y*y) of Free Associative Unital Algebra on 3 ↪
↪generators (x, y, z) over Rational Field
sage: I.reduce(F*[x^2+x*y,y^2+y*z]*F)

```

(continues on next page)

(continued from previous page)

```
Twosided Ideal (x*y + y*z, -y*x + y*z) of Free Associative Unital Algebra on 3
generators (x, y, z) over Rational Field
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, implementation='letterplace', names=('x', 'y', 'z'));
↳ (x, y, z) = F._first_ngens(3)
>>> I = F*[x*y+y*z, x**Integer(2)+x*y-y*x-y**Integer(2)]*F
>>> I.reduce(F)
Twosided Ideal (0) of Free Associative Unital Algebra on 3 generators (x, y,
↳ z) over Rational Field
>>> I.reduce(x**Integer(3))
-y*z*x - y*z*y - y*z*z
>>> I.reduce([x*y])
Twosided Ideal (y*z, x*x - y*x - y*y) of Free Associative Unital Algebra on 3
↳ generators (x, y, z) over Rational Field
>>> I.reduce(F*[x**Integer(2)+x*y, y**Integer(2)+y*z]*F)
Twosided Ideal (x*y + y*z, -y*x + y*z) of Free Associative Unital Algebra on 3
↳ generators (x, y, z) over Rational Field
```

3.6 Finite dimensional free algebra quotients

REMARK:

This implementation only works for finite dimensional quotients, since a list of basis monomials and the multiplication matrices need to be explicitly provided.

The homogeneous part of a quotient of a free algebra over a field by a finitely generated homogeneous twosided ideal is available in a different implementation. See [free_algebra_letterplace](#) and [quotient_ring](#).

class `sage.algebras.free_algebra_quotient.FreeAlgebraQuotient` (*A*, *mons*, *mats*, *names*)

Bases: `UniqueRepresentation`, `Parent`

Return a quotient algebra defined via the action of a free algebra *A* on a (finitely generated) free module.

The input for the quotient algebra is a list of monomials (in the underlying monoid for *A*) which form a free basis for the module of *A*, and a list of matrices, which give the action of the free generators of *A* on this monomial basis.

EXAMPLES:

Quaternion algebra defined in terms of three generators:

```
sage: n = 3
sage: A = FreeAlgebra(QQ, n, 'i')
sage: F = A.monoid()
sage: i, j, k = F.gens()
sage: mons = [F(1), i, j, k]
sage: M = MatrixSpace(QQ, 4)
sage: mats = [M([0, 1, 0, 0, -1, 0, 0, 0, 0, 0, 0, -1, 0, 0, 1, 0]),
.....:         M([0, 0, 1, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, -1, 0, 0]),
.....:         M([0, 0, 0, 1, 0, 0, -1, 0, 0, 1, 0, 0, -1, 0, 0, 0])]
sage: H3.<i, j, k> = FreeAlgebraQuotient(A, mons, mats)
sage: x = 1 + i + j + k
sage: x
```

(continues on next page)

(continued from previous page)

```

1 + i + j + k
sage: x**128
-170141183460469231731687303715884105728
+ 170141183460469231731687303715884105728*i
+ 170141183460469231731687303715884105728*j
+ 170141183460469231731687303715884105728*k

```

```

>>> from sage.all import *
>>> n = Integer(3)
>>> A = FreeAlgebra(QQ,n,'i')
>>> F = A.monoid()
>>> i, j, k = F.gens()
>>> mons = [F(Integer(1)), i, j, k]
>>> M = MatrixSpace(QQ,Integer(4))
>>> mats = [M([Integer(0),Integer(1),Integer(0),Integer(0), -Integer(1),
↳Integer(0),Integer(0),Integer(0), Integer(0),Integer(0),Integer(0),-Integer(1),
↳Integer(0),Integer(0),Integer(1),Integer(0)]),
...      M([Integer(0),Integer(0),Integer(1),Integer(0), Integer(0),Integer(0),
↳Integer(0),Integer(1), -Integer(1),Integer(0),Integer(0),Integer(0), Integer(0),
↳-Integer(1),Integer(0),Integer(0)]),
...      M([Integer(0),Integer(0),Integer(0),Integer(1), Integer(0),Integer(0),
↳-Integer(1),Integer(0), Integer(0),Integer(1),Integer(0),Integer(0), -
↳Integer(1),Integer(0),Integer(0),Integer(0)] )]
>>> H3 = FreeAlgebraQuotient(A,mons,mats, names=('i', 'j', 'k',)); (i, j, k,) =
↳H3._first_ngens(3)
>>> x = Integer(1) + i + j + k
>>> x
1 + i + j + k
>>> x**Integer(128)
-170141183460469231731687303715884105728
+ 170141183460469231731687303715884105728*i
+ 170141183460469231731687303715884105728*j
+ 170141183460469231731687303715884105728*k

```

Same algebra defined in terms of two generators, with some penalty on already slow arithmetic.

```

sage: n = 2
sage: A = FreeAlgebra(QQ,n,'x')
sage: F = A.monoid()
sage: i, j = F.gens()
sage: mons = [ F(1), i, j, i*j ]
sage: r = len(mons)
sage: M = MatrixSpace(QQ,r)
sage: mats = [M([0,1,0,0, -1,0,0,0, 0,0,0,-1, 0,0,1,0]),
...      M([0,0,1,0, 0,0,0,1, -1,0,0,0, 0,-1,0,0]) ]
sage: H2.<i,j> = A.quotient(mons,mats)
sage: k = i*j
sage: x = 1 + i + j + k
sage: x
1 + i + j + i*j
sage: x**128
-170141183460469231731687303715884105728

```

(continues on next page)

(continued from previous page)

```
+ 170141183460469231731687303715884105728*i
+ 170141183460469231731687303715884105728*j
+ 170141183460469231731687303715884105728*i*j
```

```
>>> from sage.all import *
>>> n = Integer(2)
>>> A = FreeAlgebra(QQ,n,'x')
>>> F = A.monoid()
>>> i, j = F.gens()
>>> mons = [ F(Integer(1)), i, j, i*j ]
>>> r = len(mons)
>>> M = MatrixSpace(QQ,r)
>>> mats = [M([Integer(0),Integer(1),Integer(0),Integer(0), -Integer(1),
↳Integer(0),Integer(0),Integer(0), Integer(0),Integer(0),Integer(0),-Integer(1),
↳Integer(0),Integer(0),Integer(1),Integer(0)]),
...      M([Integer(0),Integer(0),Integer(1),Integer(0), Integer(0),Integer(0),
↳Integer(0),Integer(1), -Integer(1),Integer(0),Integer(0),Integer(0), Integer(0),
↳-Integer(1),Integer(0),Integer(0)]) ]
>>> H2 = A.quotient(mons,mats, names=('i', 'j',)); (i, j,) = H2._first_ngens(2)
>>> k = i*j
>>> x = Integer(1) + i + j + k
>>> x
1 + i + j + i*j
>>> x**Integer(128)
-170141183460469231731687303715884105728
+ 170141183460469231731687303715884105728*i
+ 170141183460469231731687303715884105728*j
+ 170141183460469231731687303715884105728*i*j
```

Elementalias of *FreeAlgebraQuotientElement***dimension()**

Return the rank of the algebra (as a free module).

EXAMPLES:

```
sage: sage.algebras.free_algebra_quotient.hamilton_quatalg(QQ)[0].dimension()
4
```

```
>>> from sage.all import *
>>> sage.algebras.free_algebra_quotient.hamilton_quatalg(QQ)[Integer(0)].
↳dimension()
4
```

free_algebra()

Return the free algebra generating the algebra.

EXAMPLES:

```
sage: sage.algebras.free_algebra_quotient.hamilton_quatalg(QQ)[0].free_
↳algebra()
Free Algebra on 3 generators (i0, i1, i2) over Rational Field
```



```
>>> from sage.all import *
>>> sage.algebras.free_algebra_quotient.hamilton_quatalg(QQ)[Integer(0)].free_
↪algebra()
Free Algebra on 3 generators (i0, i1, i2) over Rational Field
```

gen(i)

Return the i-th generator of the algebra.

EXAMPLES:

```
sage: H, (i,j,k) = sage.algebras.free_algebra_quotient.hamilton_quatalg(QQ)
sage: H.gen(0)
i
sage: H.gen(2)
k
```

```
>>> from sage.all import *
>>> H, (i,j,k) = sage.algebras.free_algebra_quotient.hamilton_quatalg(QQ)
>>> H.gen(Integer(0))
i
>>> H.gen(Integer(2))
k
```

An `IndexError` is raised if an invalid generator is requested:

```
sage: H.gen(3)
Traceback (most recent call last):
...
IndexError: argument i (= 3) must be between 0 and 2
```

```
>>> from sage.all import *
>>> H.gen(Integer(3))
Traceback (most recent call last):
...
IndexError: argument i (= 3) must be between 0 and 2
```

Negative indexing into the generators is not supported:

```
sage: H.gen(-1)
Traceback (most recent call last):
...
IndexError: argument i (= -1) must be between 0 and 2
```

```
>>> from sage.all import *
>>> H.gen(-Integer(1))
Traceback (most recent call last):
...
IndexError: argument i (= -1) must be between 0 and 2
```

gens()

Return the tuple of generators of self.

EXAMPLES:

```
sage: H, (i,j,k) = sage.algebras.free_algebra_quotient.hamilton_quatalg(QQ)
sage: H.gens()
(i, j, k)
```

```
>>> from sage.all import *
>>> H, (i,j,k) = sage.algebras.free_algebra_quotient.hamilton_quatalg(QQ)
>>> H.gens()
(i, j, k)
```

matrix_action()

Return the matrix action used to define the algebra.

EXAMPLES:

```
sage: sage.algebras.free_algebra_quotient.hamilton_quatalg(QQ)[0].matrix_
↪action()
(
[ 0  1  0  0] [ 0  0  1  0] [ 0  0  0  1]
[-1  0  0  0] [ 0  0  0  1] [ 0  0 -1  0]
[ 0  0  0 -1] [-1  0  0  0] [ 0  1  0  0]
[ 0  0  1  0], [ 0 -1  0  0], [-1  0  0  0]
)
```

```
>>> from sage.all import *
>>> sage.algebras.free_algebra_quotient.hamilton_quatalg(QQ)[Integer(0)].
↪matrix_action()
(
[ 0  1  0  0] [ 0  0  1  0] [ 0  0  0  1]
[-1  0  0  0] [ 0  0  0  1] [ 0  0 -1  0]
[ 0  0  0 -1] [-1  0  0  0] [ 0  1  0  0]
[ 0  0  1  0], [ 0 -1  0  0], [-1  0  0  0]
)
```

module()

Return the free module of the algebra.

EXAMPLES:

```
sage: H = sage.algebras.free_algebra_quotient.hamilton_quatalg(QQ)[0]; H
Free algebra quotient on 3 generators ('i', 'j', 'k') and dimension 4 over_
↪Rational Field
sage: H.module()
Vector space of dimension 4 over Rational Field
```

```
>>> from sage.all import *
>>> H = sage.algebras.free_algebra_quotient.hamilton_quatalg(QQ)[Integer(0)];_
↪H
Free algebra quotient on 3 generators ('i', 'j', 'k') and dimension 4 over_
↪Rational Field
>>> H.module()
Vector space of dimension 4 over Rational Field
```

monoid()

Return the free monoid of generators of the algebra.

EXAMPLES:

```
sage: sage.algebras.free_algebra_quotient.hamilton_quatalg(QQ)[0].monoid()
Free monoid on 3 generators (i0, i1, i2)
```

```
>>> from sage.all import *
>>> sage.algebras.free_algebra_quotient.hamilton_quatalg(QQ)[Integer(0)].
↳monoid()
Free monoid on 3 generators (i0, i1, i2)
```

monomial_basis()

The free monoid of generators of the algebra as elements of a free monoid.

EXAMPLES:

```
sage: sage.algebras.free_algebra_quotient.hamilton_quatalg(QQ)[0].monomial_
↳basis()
(1, i0, i1, i2)
```

```
>>> from sage.all import *
>>> sage.algebras.free_algebra_quotient.hamilton_quatalg(QQ)[Integer(0)].
↳monomial_basis()
(1, i0, i1, i2)
```

ngens()

Return the number of generators of the algebra.

EXAMPLES:

```
sage: sage.algebras.free_algebra_quotient.hamilton_quatalg(QQ)[0].ngens()
3
```

```
>>> from sage.all import *
>>> sage.algebras.free_algebra_quotient.hamilton_quatalg(QQ)[Integer(0)].
↳ngens()
3
```

rank()

Return the rank of the algebra (as a free module).

EXAMPLES:

```
sage: sage.algebras.free_algebra_quotient.hamilton_quatalg(QQ)[0].rank()
4
```

```
>>> from sage.all import *
>>> sage.algebras.free_algebra_quotient.hamilton_quatalg(QQ)[Integer(0)].
↳rank()
4
```

`sage.algebras.free_algebra_quotient.hamilton_quatalg(R)`

Hamilton quaternion algebra over the commutative ring R , constructed as a free algebra quotient.

INPUT:

- R – a commutative ring

OUTPUT:

- Q – quaternion algebra
- gens – generators for Q

EXAMPLES:

```
sage: H, (i,j,k) = sage.algebras.free_algebra_quotient.hamilton_quatalg(ZZ)
sage: H
Free algebra quotient on 3 generators ('i', 'j', 'k') and dimension 4
over Integer Ring
sage: i^2
-1
sage: i in H
True
```

```
>>> from sage.all import *
>>> H, (i,j,k) = sage.algebras.free_algebra_quotient.hamilton_quatalg(ZZ)
>>> H
Free algebra quotient on 3 generators ('i', 'j', 'k') and dimension 4
over Integer Ring
>>> i**Integer(2)
-1
>>> i in H
True
```

Note that there is another vastly more efficient model for quaternion algebras in Sage; the one here is mainly for testing purposes:

```
sage: R.<i,j,k> = QuaternionAlgebra(QQ,-1,-1) # much fast than the above
```

```
>>> from sage.all import *
>>> R = QuaternionAlgebra(QQ,-Integer(1),-Integer(1), names=('i', 'j', 'k')); (i,
↪ j, k,) = R._first_ngens(3) # much fast than the above
```

3.7 Free algebra quotient elements

AUTHORS:

- William Stein (2011-11-19): improved doctest coverage to 100%
- David Kohel (2005-09): initial version

class `sage.algebras.free_algebra_quotient_element.FreeAlgebraQuotientElement(A,x)`

Bases: `AlgebraElement`

Create the element x of the `FreeAlgebraQuotient A`.

EXAMPLES:

```
sage: H, (i,j,k) = sage.algebras.free_algebra_quotient.hamilton_quatalg(ZZ)
sage: sage.algebras.free_algebra_quotient.FreeAlgebraQuotientElement(H, i)
i
sage: a = sage.algebras.free_algebra_quotient.FreeAlgebraQuotientElement(H, 1); a
1
sage: a in H
True
```

```
>>> from sage.all import *
>>> H, (i,j,k) = sage.algebras.free_algebra_quotient.hamilton_quatalg(ZZ)
>>> sage.algebras.free_algebra_quotient.FreeAlgebraQuotientElement(H, i)
i
>>> a = sage.algebras.free_algebra_quotient.FreeAlgebraQuotientElement(H,
↳Integer(1)); a
1
>>> a in H
True
```

vector()

Return underlying vector representation of this element.

EXAMPLES:

```
sage: H, (i,j,k) = sage.algebras.free_algebra_quotient.hamilton_quatalg(QQ)
sage: ((2/3)*i - j).vector()
(0, 2/3, -1, 0)
```

```
>>> from sage.all import *
>>> H, (i,j,k) = sage.algebras.free_algebra_quotient.hamilton_quatalg(QQ)
>>> ((Integer(2)/Integer(3))*i - j).vector()
(0, 2/3, -1, 0)
```

`sage.algebras.free_algebra_quotient_element.is_FreeAlgebraQuotientElement(x)`

EXAMPLES:

```
sage: H, (i,j,k) = sage.algebras.free_algebra_quotient.hamilton_quatalg(QQ)
sage: sage.algebras.free_algebra_quotient_element.is_FreeAlgebraQuotientElement(i)
doctest:warning...
DeprecationWarning: The function is_FreeAlgebraQuotientElement is deprecated;
use 'isinstance(..., FreeAlgebraQuotientElement)' instead.
See https://github.com/sagemath/sage/issues/38184 for details.
True
```

```
>>> from sage.all import *
>>> H, (i,j,k) = sage.algebras.free_algebra_quotient.hamilton_quatalg(QQ)
>>> sage.algebras.free_algebra_quotient_element.is_FreeAlgebraQuotientElement(i)
doctest:warning...
DeprecationWarning: The function is_FreeAlgebraQuotientElement is deprecated;
use 'isinstance(..., FreeAlgebraQuotientElement)' instead.
See https://github.com/sagemath/sage/issues/38184 for details.
True
```

Of course this is testing the data type:

```
sage: sage.algebras.free_algebra_quotient_element.is_FreeAlgebraQuotientElement(1)
False
sage: sage.algebras.free_algebra_quotient_element.is_
↪FreeAlgebraQuotientElement(H(1))
True
```

```
>>> from sage.all import *
>>> sage.algebras.free_algebra_quotient_element.is_
↪FreeAlgebraQuotientElement(Integer(1))
False
>>> sage.algebras.free_algebra_quotient_element.is_
↪FreeAlgebraQuotientElement(H(Integer(1)))
True
```

3.8 Tensor Algebras

AUTHORS:

- Travis Scrimshaw (2014-01-24): Initial version

Todo

- Coerce to/from free algebra.

class sage.algebras.tensor_algebra.**BaseRingLift**

Bases: `Morphism`

Morphism $R \rightarrow T(M)$ which identifies the base ring R of a tensor algebra $T(M)$ with the 0-th graded part of $T(M)$.

class sage.algebras.tensor_algebra.**TensorAlgebra** (M , *prefix='T'*, *category=None*, ***options*)

Bases: `CombinatorialFreeModule`

The tensor algebra $T(M)$ of a module M .

Let $\{b_i\}_{i \in I}$ be a basis of the R -module M . Then the tensor algebra $T(M)$ of M is an associative R -algebra, with a basis consisting of all tensors of the form $b_{i_1} \otimes b_{i_2} \otimes \cdots \otimes b_{i_n}$ for nonnegative integers n and n -tuples $(i_1, i_2, \dots, i_n) \in I^n$. The product of $T(M)$ is given by

$$(b_{i_1} \otimes \cdots \otimes b_{i_m}) \cdot (b_{j_1} \otimes \cdots \otimes b_{j_n}) = b_{i_1} \otimes \cdots \otimes b_{i_m} \otimes b_{j_1} \otimes \cdots \otimes b_{j_n}.$$

As an algebra, it is generated by the basis vectors b_i of M . It is an $\mathbb{N}$ -graded R -algebra, with the degree of each b_i being 1.

It also has a Hopf algebra structure: The comultiplication is the unique algebra morphism $\delta : T(M) \rightarrow T(M) \otimes T(M)$ defined by:

$$\delta(b_i) = b_i \otimes 1 + 1 \otimes b_i$$

(where the $\otimes$ symbol here forms tensors in $T(M) \otimes T(M)$, not inside $T(M)$ itself). The counit is the unique algebra morphism $T(M) \rightarrow R$ sending each b_i to 0. Its antipode S satisfies

$$S(b_{i_1} \otimes \cdots \otimes b_{i_m}) = (-1)^m (b_{i_m} \otimes \cdots \otimes b_{i_1}).$$

This is a connected graded cocommutative Hopf algebra.

REFERENCES:

- [Wikipedia article Tensor_algebra](#)

See also

TensorAlgebra

EXAMPLES:

```
sage: C = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: TA = TensorAlgebra(C)
sage: TA.dimension()
+Infinity
sage: TA.base_ring()
Rational Field
sage: TA.algebra_generators()
Finite family {'a': B['a'], 'b': B['b'], 'c': B['c']}
```

```
>>> from sage.all import *
>>> C = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
>>> TA = TensorAlgebra(C)
>>> TA.dimension()
+Infinity
>>> TA.base_ring()
Rational Field
>>> TA.algebra_generators()
Finite family {'a': B['a'], 'b': B['b'], 'c': B['c']}
```

`algebra_generators()`

Return the generators of this algebra.

EXAMPLES:

```
sage: C = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: TA = TensorAlgebra(C)
sage: TA.algebra_generators()
Finite family {'a': B['a'], 'b': B['b'], 'c': B['c']}
sage: m = SymmetricFunctions(QQ).m()
sage: Tm = TensorAlgebra(m)
sage: Tm.algebra_generators()
Lazy family (generator(i))_{i in Partitions}
```

```
>>> from sage.all import *
>>> C = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
>>> TA = TensorAlgebra(C)
>>> TA.algebra_generators()
Finite family {'a': B['a'], 'b': B['b'], 'c': B['c']}
>>> m = SymmetricFunctions(QQ).m()
>>> Tm = TensorAlgebra(m)
>>> Tm.algebra_generators()
Lazy family (generator(i))_{i in Partitions}
```

antipode_on_basis(*m*)

Return the antipode of the simple tensor indexed by *m*.

EXAMPLES:

```
sage: C = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: TA = TensorAlgebra(C)
sage: s = TA(['a', 'b', 'c']).leading_support()
sage: TA.antipode_on_basis(s)
-B['c'] # B['b'] # B['a']
sage: t = TA(['a', 'b', 'b', 'b']).leading_support()
sage: TA.antipode_on_basis(t)
B['b'] # B['b'] # B['b'] # B['a']
```

```
>>> from sage.all import *
>>> C = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
>>> TA = TensorAlgebra(C)
>>> s = TA(['a', 'b', 'c']).leading_support()
>>> TA.antipode_on_basis(s)
-B['c'] # B['b'] # B['a']
>>> t = TA(['a', 'b', 'b', 'b']).leading_support()
>>> TA.antipode_on_basis(t)
B['b'] # B['b'] # B['b'] # B['a']
```

base_module()

Return the base module of self.

EXAMPLES:

```
sage: C = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: TA = TensorAlgebra(C)
sage: TA.base_module() is C
True
```

```
>>> from sage.all import *
>>> C = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
>>> TA = TensorAlgebra(C)
>>> TA.base_module() is C
True
```

construction()

Return the functorial construction of self.

EXAMPLES:

```
sage: C = CombinatorialFreeModule(ZZ, ['a', 'b', 'c'])
sage: TA = TensorAlgebra(C)
sage: f, M = TA.construction()
sage: M == C
True
sage: f(M) == TA
True
```



```

>>> from sage.all import *
>>> C = CombinatorialFreeModule(ZZ, ['a', 'b', 'c'])
>>> TA = TensorAlgebra(C)
>>> f, M = TA.construction()
>>> M == C
True
>>> f(M) == TA
True

```

`coproduct_on_basis(m)`

Return the coproduct of the simple tensor indexed by `m`.

EXAMPLES:

```

sage: C = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: TA = TensorAlgebra(C, tensor_symbol="(X)")
sage: TA.coproduct_on_basis(TA.one_basis())
1 # 1
sage: I = TA.indices()
sage: ca = TA.coproduct_on_basis(I.gen('a')); ca
1 # B['a'] + B['a'] # 1
sage: s = TA(['a', 'b', 'c']).leading_support()
sage: cp = TA.coproduct_on_basis(s); cp
1 # B['a'](X)B['b'](X)B['c'] + B['a'] # B['b'](X)B['c']
+ B['a'](X)B['b'] # B['c'] + B['a'](X)B['b'](X)B['c'] # 1
+ B['a'](X)B['c'] # B['b'] + B['b'] # B['a'](X)B['c']
+ B['b'](X)B['c'] # B['a'] + B['c'] # B['a'](X)B['b']

```

```

>>> from sage.all import *
>>> C = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
>>> TA = TensorAlgebra(C, tensor_symbol="(X)")
>>> TA.coproduct_on_basis(TA.one_basis())
1 # 1
>>> I = TA.indices()
>>> ca = TA.coproduct_on_basis(I.gen('a')); ca
1 # B['a'] + B['a'] # 1
>>> s = TA(['a', 'b', 'c']).leading_support()
>>> cp = TA.coproduct_on_basis(s); cp
1 # B['a'](X)B['b'](X)B['c'] + B['a'] # B['b'](X)B['c']
+ B['a'](X)B['b'] # B['c'] + B['a'](X)B['b'](X)B['c'] # 1
+ B['a'](X)B['c'] # B['b'] + B['b'] # B['a'](X)B['c']
+ B['b'](X)B['c'] # B['a'] + B['c'] # B['a'](X)B['b']

```

We check that $\Delta(a \otimes b \otimes c) = \Delta(a)\Delta(b)\Delta(c)$:

```

sage: cb = TA.coproduct_on_basis(I.gen('b'))
sage: cc = TA.coproduct_on_basis(I.gen('c'))
sage: cp == ca * cb * cc
True

```

```

>>> from sage.all import *
>>> cb = TA.coproduct_on_basis(I.gen('b'))
>>> cc = TA.coproduct_on_basis(I.gen('c'))

```

(continues on next page)

(continued from previous page)

```
>>> cp == ca * cb * cc
True
```

counit(*x*)Return the counit of *x*.

INPUT:

- *x* – an element of *self*

EXAMPLES:

```
sage: C = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: TA = TensorAlgebra(C)
sage: x = TA(['a', 'b', 'c'])
sage: TA.counit(x)
0
sage: TA.counit(x + 3)
3
```

```
>>> from sage.all import *
>>> C = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
>>> TA = TensorAlgebra(C)
>>> x = TA(['a', 'b', 'c'])
>>> TA.counit(x)
0
>>> TA.counit(x + Integer(3))
3
```

degree_on_basis(*m*)Return the degree of the simple tensor *m*, which is its length (thought of as an element in the free monoid).

EXAMPLES:

```
sage: C = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: TA = TensorAlgebra(C)
sage: s = TA(['a', 'b', 'c']).leading_support(); s
F['a']*F['b']*F['c']
sage: TA.degree_on_basis(s)
3
```

```
>>> from sage.all import *
>>> C = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
>>> TA = TensorAlgebra(C)
>>> s = TA(['a', 'b', 'c']).leading_support(); s
F['a']*F['b']*F['c']
>>> TA.degree_on_basis(s)
3
```

gens()

Return the generators of this algebra.

EXAMPLES:

```

sage: C = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: TA = TensorAlgebra(C)
sage: TA.algebra_generators()
Finite family {'a': B['a'], 'b': B['b'], 'c': B['c']}
sage: m = SymmetricFunctions(QQ).m()
sage: Tm = TensorAlgebra(m)
sage: Tm.algebra_generators()
Lazy family (generator(i))_{i in Partitions}

```

```

>>> from sage.all import *
>>> C = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
>>> TA = TensorAlgebra(C)
>>> TA.algebra_generators()
Finite family {'a': B['a'], 'b': B['b'], 'c': B['c']}
>>> m = SymmetricFunctions(QQ).m()
>>> Tm = TensorAlgebra(m)
>>> Tm.algebra_generators()
Lazy family (generator(i))_{i in Partitions}

```

one_basis()

Return the empty word, which indexes the 1 of this algebra.

EXAMPLES:

```

sage: C = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: TA = TensorAlgebra(C)
sage: TA.one_basis()
1
sage: TA.one_basis().parent()
Free monoid indexed by {'a', 'b', 'c'}
sage: m = SymmetricFunctions(QQ).m()
sage: Tm = TensorAlgebra(m)
sage: Tm.one_basis()
1
sage: Tm.one_basis().parent()
Free monoid indexed by Partitions

```

```

>>> from sage.all import *
>>> C = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
>>> TA = TensorAlgebra(C)
>>> TA.one_basis()
1
>>> TA.one_basis().parent()
Free monoid indexed by {'a', 'b', 'c'}
>>> m = SymmetricFunctions(QQ).m()
>>> Tm = TensorAlgebra(m)
>>> Tm.one_basis()
1
>>> Tm.one_basis().parent()
Free monoid indexed by Partitions

```

product_on_basis(a, b)

Return the product of the basis elements indexed by a and b, as per AlgebrasWithBasis.

```
ParentMethods.product_on_basis().
```

INPUT:

- a, b – basis indices

EXAMPLES:

```
sage: C = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: TA = TensorAlgebra(C)
sage: I = TA.indices()
sage: g = I.gens()
sage: TA.product_on_basis(g['a']*g['b'], g['a']*g['c'])
B['a'] # B['b'] # B['a'] # B['c']
```

```
>>> from sage.all import *
>>> C = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
>>> TA = TensorAlgebra(C)
>>> I = TA.indices()
>>> g = I.gens()
>>> TA.product_on_basis(g['a']*g['b'], g['a']*g['c'])
B['a'] # B['b'] # B['a'] # B['c']
```

```
class sage.algebras.tensor_algebra.TensorAlgebraFunctor(base)
```

Bases: `ConstructionFunctor`

The tensor algebra functor.

Let R be a unital ring. Let V_R and A_R be the categories of R -modules and R -algebras respectively. The functor $T : V_R \rightarrow A_R$ sends an R -module M to the tensor algebra $T(M)$. The functor T is left-adjoint to the forgetful functor $F : A_R \rightarrow V_R$.

INPUT:

- `base` – the base R

rank = 20

FINITE DIMENSIONAL ALGEBRAS

4.1 Finite-Dimensional Algebras

class sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra (*k*, *table*, *names*, *assume_associative*, *category*)

Bases: UniqueRepresentation, Parent

Create a finite-dimensional k -algebra from a multiplication table.

INPUT:

- k – a field
- *table* – list of matrices
- *names* – string (default: 'e'); names for the basis elements
- *assume_associative* – boolean (default: False); if True, then the category is set to category.Associative() and methods requiring associativity assume this
- *category* – (default: MagmaticAlgebras(k).FiniteDimensional().WithBasis()) the category to which this algebra belongs

The list *table* must have the following form: there exists a finite-dimensional k -algebra of degree n with basis $(e_1, \dots, e_n)$ such that the i -th element of *table* is the matrix of right multiplication by e_i with respect to the basis $(e_1, \dots, e_n)$.

EXAMPLES:

```
sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1, 0], [0, 1]]),
.....:                               Matrix([[0, 1], [0, 0]])]); A
Finite-dimensional algebra of degree 2 over Finite Field of size 3
sage: TestSuite(A).run()

sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1, 0, 0], [0, 1, 0], [0, 0, 0]]),
.....:                               Matrix([[0, 1, 0], [0, 0, 0], [0, 0, 0]]),
.....:                               Matrix([[0, 0, 0], [0, 0, 0], [0, 0, 1]])])
sage: B
Finite-dimensional algebra of degree 3 over Rational Field
```

```

>>> from sage.all import *
>>> A = FiniteDimensionalAlgebra(GF(Integer(3)), [Matrix([[Integer(1),
↳ Integer(0)], [Integer(0), Integer(1)]]),
...
Matrix([[Integer(0), Integer(1)],
↳ [Integer(0), Integer(0)]])]); A
Finite-dimensional algebra of degree 2 over Finite Field of size 3
>>> TestSuite(A).run()

>>> B = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0), Integer(0)],
↳ [Integer(0), Integer(1), Integer(0)], [Integer(0), Integer(0), Integer(0)]]),
...
Matrix([[Integer(0), Integer(1), Integer(0)],
↳ [Integer(0), Integer(0), Integer(0)], [Integer(0), Integer(0), Integer(0)]]),
...
Matrix([[Integer(0), Integer(0), Integer(0)],
↳ [Integer(0), Integer(0), Integer(1)]])]])
>>> B
Finite-dimensional algebra of degree 3 over Rational Field

```

Element

alias of *FiniteDimensionalAlgebraElement*

base_extend(*F*)

Return self base changed to the field *F*.

EXAMPLES:

```

sage: C = FiniteDimensionalAlgebra(GF(2), [Matrix([1])])
sage: k.<y> = GF(4) #
↳ needs sage.rings.finite_rings
sage: C.base_extend(k) #
↳ needs sage.rings.finite_rings
Finite-dimensional algebra of degree 1 over Finite Field in y of size 2^2

```

```

>>> from sage.all import *
>>> C = FiniteDimensionalAlgebra(GF(Integer(2)), [Matrix([Integer(1)])])
>>> k = GF(Integer(4), names=('y',)); (y,) = k._first_ngens(1) # needs sage.
↳ rings.finite_rings
>>> C.base_extend(k) #
↳ needs sage.rings.finite_rings
Finite-dimensional algebra of degree 1 over Finite Field in y of size 2^2

```

basis()

Return a list of the basis elements of self.

EXAMPLES:

```

sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1, 0], [0, 1]]),
...:
Matrix([[0, 1], [0, 0]])])
sage: A.basis()
Finite family {0: e0, 1: e1}

```

```

>>> from sage.all import *
>>> A = FiniteDimensionalAlgebra(GF(Integer(3)), [Matrix([[Integer(1),
↳ Integer(0)], [Integer(0), Integer(1)]]),

```

(continues on next page)

(continued from previous page)

```

...                               Matrix([[Integer(0), Integer(1)],
↪[Integer(0), Integer(0)]])])
>>> A.basis()
Finite family {0: e0, 1: e1}

```

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```

sage: A = FiniteDimensionalAlgebra(GF(7), [Matrix([[1, 0], [0, 1]]),
....:                               Matrix([[0, 1], [2, 3]])])
sage: A.cardinality()
49

sage: B = FiniteDimensionalAlgebra(RR, [Matrix([[1, 0], [0, 1]]),
....:                               Matrix([[0, 1], [2, 3]])])
sage: B.cardinality()
+Infinity

sage: C = FiniteDimensionalAlgebra(RR, [])
sage: C.cardinality()
1

```

```

>>> from sage.all import *
>>> A = FiniteDimensionalAlgebra(GF(Integer(7)), [Matrix([[Integer(1),
↪Integer(0)], [Integer(0), Integer(1)]]),
...                               Matrix([[Integer(0), Integer(1)],
↪[Integer(2), Integer(3)]])])
>>> A.cardinality()
49

>>> B = FiniteDimensionalAlgebra(RR, [Matrix([[Integer(1), Integer(0)],
↪[Integer(0), Integer(1)]]),
...                               Matrix([[Integer(0), Integer(1)],
↪[Integer(2), Integer(3)]])])
>>> B.cardinality()
+Infinity

>>> C = FiniteDimensionalAlgebra(RR, [])
>>> C.cardinality()
1

```

degree()

Return the number of generators of `self`, i.e., the degree of `self` over its base field.

EXAMPLES:

```

sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1, 0], [0, 1]]),
....:                               Matrix([[0, 1], [0, 0]])])
sage: A.ngens()
2

```

```
>>> from sage.all import *
>>> A = FiniteDimensionalAlgebra(GF(Integer(3)), [Matrix([[Integer(1),
↳ Integer(0)], [Integer(0), Integer(1)]]),
...
Matrix([[Integer(0), Integer(1)],
↳ [Integer(0), Integer(0)]]]])
>>> A.ngens()
2
```

from_base_ring(*x*)

gen(*i*)

Return the *i*-th basis element of self.

EXAMPLES:

```
sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1, 0], [0, 1]]),
...:
Matrix([[0, 1], [0, 0]])])
sage: A.gen(0)
e0
```

```
>>> from sage.all import *
>>> A = FiniteDimensionalAlgebra(GF(Integer(3)), [Matrix([[Integer(1),
↳ Integer(0)], [Integer(0), Integer(1)]]),
...
Matrix([[Integer(0), Integer(1)],
↳ [Integer(0), Integer(0)]]]])
>>> A.gen(Integer(0))
e0
```

ideal(*gens=None, given_by_matrix=False, side=None*)

Return the right ideal of self generated by gens.

INPUT:

- *A* – a *FiniteDimensionalAlgebra*
- *gens* – (default: None) either an element of *A* or a list of elements of *A*, given as vectors, matrices, or *FiniteDimensionalAlgebraElements*. If *given_by_matrix* is True, then *gens* should instead be a matrix whose rows form a basis of an ideal of *A*.
- *given_by_matrix* – boolean (default: False); if True, no checking is done
- *side* – ignored but necessary for coercions

The algebra *A* has to be in the category of associative algebras.

EXAMPLES:

```
sage: cat = Algebras(GF(3)).FiniteDimensional().WithBasis()
sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1, 0], [0, 1]]),
...:
Matrix([[0, 1], [0, 0]])],
...:
category=cat)
sage: A.ideal(A([1, 1]))
Ideal (e0 + e1) of
Finite-dimensional algebra of degree 2 over Finite Field of size 3
```



```

>>> from sage.all import *
>>> cat = Algebras(GF(Integer(3))).FiniteDimensional().WithBasis()
>>> A = FiniteDimensionalAlgebra(GF(Integer(3)), [Matrix([[Integer(1),
↳ Integer(0)], [Integer(0), Integer(1)]]),
...
Matrix([[Integer(0), Integer(1)],
↳ [Integer(0), Integer(0)]])],
...
category=cat)
>>> A.ideal(A([Integer(1), Integer(1)]))
Ideal (e0 + e1) of
Finite-dimensional algebra of degree 2 over Finite Field of size 3

```

is_associative()

Return True if self is associative.

EXAMPLES:

```

sage: A = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0], [0,1]]),
.....:
Matrix([[0,1], [-1,0]])])
sage: A.is_associative()
True

sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,1,0], [0,0,1]]),
.....:
Matrix([[0,1,0], [0,0,0], [0,0,0]]),
.....:
Matrix([[0,0,1], [0,0,0], [1,0,0]])])
sage: B.is_associative()
False

sage: e = B.basis()
sage: (e[1]*e[2])*e[2]==e[1]*(e[2]*e[2])
False

```

```

>>> from sage.all import *
>>> A = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0)],
↳ [Integer(0), Integer(1)]]),
...
Matrix([[Integer(0), Integer(1)], [-
↳ Integer(1), Integer(0)]])]
>>> A.is_associative()
True

>>> B = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0),
↳ Integer(0)], [Integer(0), Integer(1), Integer(0)], [Integer(0), Integer(0),
↳ Integer(1)]]),
...
Matrix([[Integer(0), Integer(1),
↳ Integer(0)], [Integer(0), Integer(0), Integer(0)], [Integer(0), Integer(0),
↳ Integer(0)]]),
...
Matrix([[Integer(0), Integer(0),
↳ Integer(1)], [Integer(0), Integer(0), Integer(0)], [Integer(1), Integer(0),
↳ Integer(0)]])]
>>> B.is_associative()
False

>>> e = B.basis()
>>> (e[Integer(1)]*e[Integer(2)]*e[Integer(2)]==e[Integer(1)]*(e[Inte-

```

(continues on next page)

(continued from previous page)

```
ger(2)]*e[Integer(2)])
False
```

is_commutative()

Return True if self is commutative.

EXAMPLES:

```
sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,1,0], [0,0,0]]),
....:                                   Matrix([[0,1,0], [0,0,0], [0,0,0]]),
....:                                   Matrix([[0,0,0], [0,0,0], [0,0,1]])])
sage: B.is_commutative()
True

sage: C = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,0,0], [0,0,0]]),
....:                                   Matrix([[0,1,0], [0,0,0], [0,0,0]]),
....:                                   Matrix([[0,0,0], [0,1,0], [0,0,1]])])
sage: C.is_commutative()
False
```

```
>>> from sage.all import *
>>> B = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1),Integer(0),
↪Integer(0)], [Integer(0),Integer(1),Integer(0)], [Integer(0),Integer(0),
↪Integer(0)]]),
...                                   Matrix([[Integer(0),Integer(1),
↪Integer(0)], [Integer(0),Integer(0),Integer(0)], [Integer(0),Integer(0),
↪Integer(0)]]),
...                                   Matrix([[Integer(0),Integer(0),
↪Integer(0)], [Integer(0),Integer(0),Integer(0)], [Integer(0),Integer(0),
↪Integer(1)]])])
>>> B.is_commutative()
True

>>> C = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1),Integer(0),
↪Integer(0)], [Integer(0),Integer(0),Integer(0)], [Integer(0),Integer(0),
↪Integer(0)]]),
...                                   Matrix([[Integer(0),Integer(1),
↪Integer(0)], [Integer(0),Integer(0),Integer(0)], [Integer(0),Integer(0),
↪Integer(0)]]),
...                                   Matrix([[Integer(0),Integer(0),
↪Integer(0)], [Integer(0),Integer(1),Integer(0)], [Integer(0),Integer(0),
↪Integer(1)]])])
>>> C.is_commutative()
False
```

is_finite()

Return True if the cardinality of self is finite.

EXAMPLES:

```
sage: A = FiniteDimensionalAlgebra(GF(7), [Matrix([[1, 0], [0, 1]]),
....:                                       Matrix([[0, 1], [2, 3]])])
sage: A.is_finite()
```

(continues on next page)

(continued from previous page)

```

True

sage: B = FiniteDimensionalAlgebra(RR, [Matrix([[1, 0], [0, 1]]),
....:                                   Matrix([[0, 1], [2, 3]])])
sage: B.is_finite()
False

sage: C = FiniteDimensionalAlgebra(RR, [])
sage: C.is_finite()
True

```

```

>>> from sage.all import *
>>> A = FiniteDimensionalAlgebra(GF(Integer(7)), [Matrix([[Integer(1),
↪ Integer(0)], [Integer(0), Integer(1)]]),
...                                             Matrix([[Integer(0), Integer(1)],
↪ [Integer(2), Integer(3)]]]])
>>> A.is_finite()
True

>>> B = FiniteDimensionalAlgebra(RR, [Matrix([[Integer(1), Integer(0)],
↪ [Integer(0), Integer(1)]]),
...                                   Matrix([[Integer(0), Integer(1)],
↪ [Integer(2), Integer(3)]]]])
>>> B.is_finite()
False

>>> C = FiniteDimensionalAlgebra(RR, [])
>>> C.is_finite()
True

```

is_unitary()

Return True if self has a two-sided multiplicative identity element.

Warning

This uses linear algebra; thus expect wrong results when the base ring is not a field.

EXAMPLES:

```

sage: A = FiniteDimensionalAlgebra(QQ, [])
sage: A.is_unitary()
True

sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1, 0], [0, 1]]),
....:                                   Matrix([[0, 1], [-1, 0]])])
sage: B.is_unitary()
True

sage: C = FiniteDimensionalAlgebra(QQ, [Matrix([[0, 0], [0, 0]]),
....:                                   Matrix([[0, 0], [0, 0]])])
sage: C.is_unitary()

```

(continues on next page)

(continued from previous page)

```

False

sage: D = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0], [0,1]]),
.....:                               Matrix([[1,0], [0,1]])])
sage: D.is_unitary()
False

sage: E = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0],[1,0]]),
.....:                               Matrix([[0,1],[0,1]])])
sage: E.is_unitary()
False

sage: F = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,1,0], [0,0,1]]),
.....:                               Matrix([[0,1,0], [0,0,0], [0,0,0]]),
.....:                               Matrix([[0,0,1], [0,0,0], [1,0,0]])])
sage: F.is_unitary()
True

sage: G = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,1,0], [0,0,1]]),
.....:                               Matrix([[0,1,0], [0,0,0], [0,0,0]]),
.....:                               Matrix([[0,1,0], [0,0,0], [1,0,0]])])
sage: G.is_unitary() # Unique right identity, but no left identity.
False

```

```

>>> from sage.all import *
>>> A = FiniteDimensionalAlgebra(QQ, [])
>>> A.is_unitary()
True

>>> B = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0)],
↪ [Integer(0), Integer(1)]]),
...                               Matrix([[Integer(0), Integer(1)], [-
↪ Integer(1), Integer(0)]]]])
>>> B.is_unitary()
True

>>> C = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(0), Integer(0)],
↪ [Integer(0), Integer(0)]]),
...                               Matrix([[Integer(0), Integer(0)], [-
↪ Integer(0), Integer(0)]]]])
>>> C.is_unitary()
False

>>> D = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0)],
↪ [Integer(0), Integer(1)]]),
...                               Matrix([[Integer(1), Integer(0)], [-
↪ Integer(0), Integer(1)]]]])
>>> D.is_unitary()
False

>>> E = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0)],
↪ [Integer(1), Integer(0)]]),

```

(continues on next page)

(continued from previous page)

```

...                               Matrix([[Integer(0), Integer(1)],
↪ [Integer(0), Integer(1)]])])
>>> E.is_unitary()
False

>>> F = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0),
↪ Integer(0)], [Integer(0), Integer(1), Integer(0)], [Integer(0), Integer(0),
↪ Integer(1)]]),
...                               Matrix([[Integer(0), Integer(1),
↪ Integer(0)], [Integer(0), Integer(0), Integer(0)], [Integer(0), Integer(0),
↪ Integer(0)]]),
...                               Matrix([[Integer(0), Integer(0),
↪ Integer(1)], [Integer(0), Integer(0), Integer(0)], [Integer(1), Integer(0),
↪ Integer(0)]])])
>>> F.is_unitary()
True

>>> G = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0),
↪ Integer(0)], [Integer(0), Integer(1), Integer(0)], [Integer(0), Integer(0),
↪ Integer(1)]]),
...                               Matrix([[Integer(0), Integer(1),
↪ Integer(0)], [Integer(0), Integer(0), Integer(0)], [Integer(0), Integer(0),
↪ Integer(0)]]),
...                               Matrix([[Integer(0), Integer(1),
↪ Integer(0)], [Integer(0), Integer(0), Integer(0)], [Integer(1), Integer(0),
↪ Integer(0)]])])
>>> G.is_unitary() # Unique right identity, but no left identity.
False

```

is_zero()

Return True if self is the zero ring.

EXAMPLES:

```

sage: A = FiniteDimensionalAlgebra(QQ, [])
sage: A.is_zero()
True

sage: B = FiniteDimensionalAlgebra(GF(7), [Matrix([0])])
sage: B.is_zero()
False

```

```

>>> from sage.all import *
>>> A = FiniteDimensionalAlgebra(QQ, [])
>>> A.is_zero()
True

>>> B = FiniteDimensionalAlgebra(GF(Integer(7)), [Matrix([Integer(0)])])
>>> B.is_zero()
False

```

left_table()

Return the list of matrices for left multiplication by the basis elements.

EXAMPLES:

```
sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0], [0,1]]),
....:                                Matrix([[0,1], [-1,0]])])
sage: T = B.left_table(); T
(
[1 0]  [ 0  1]
[0 1], [-1  0]
)
```

```
>>> from sage.all import *
>>> B = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1),Integer(0)],
↪[Integer(0),Integer(1)]]),
....:                                Matrix([[Integer(0),Integer(1)], [-
↪Integer(1),Integer(0)]]))])
>>> T = B.left_table(); T
(
[1 0]  [ 0  1]
[0 1], [-1  0]
)
```

We check immutability:

```
sage: T[0] = "vandalized by h4xx0r"
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
sage: T[1][0] = [13, 37]
Traceback (most recent call last):
...
ValueError: matrix is immutable; please change a copy instead
(i.e., use copy(M) to change a copy of M).
```

```
>>> from sage.all import *
>>> T[Integer(0)] = "vandalized by h4xx0r"
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
>>> T[Integer(1)][Integer(0)] = [Integer(13), Integer(37)]
Traceback (most recent call last):
...
ValueError: matrix is immutable; please change a copy instead
(i.e., use copy(M) to change a copy of M).
```

maximal_ideal()

Compute the maximal ideal of the local algebra `self`.

Note

self has to be in the category of unitary, commutative and associative algebras as in the examples below. It must moreover be local (have a unique maximal ideal).

OUTPUT:

- `FiniteDimensionalAlgebraIdeal`; the unique maximal ideal of `self`. If `self` is not a local algebra, a `ValueError` is raised.

EXAMPLES:

```
sage: cat = CommutativeAlgebras(GF(3)).FiniteDimensional().WithBasis()
sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1, 0], [0, 1]]),
....:                                     Matrix([[0, 1], [0, 0]])],
....:                                     category=cat)
sage: A.maximal_ideal()                                     #_
↳needs sage.rings.finite_rings
Ideal (0, e1) of
Finite-dimensional algebra of degree 2 over Finite Field of size 3

sage: cat = CommutativeAlgebras(QQ).FiniteDimensional().WithBasis()
sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1, 0, 0], [0, 1, 0], [0, 0, 0]]),
....:                                     Matrix([[0, 1, 0], [0, 0, 0], [0, 0, 0]]),
....:                                     Matrix([[0, 0, 0], [0, 0, 0], [0, 0, 1]])],
....:                                     category=cat)
sage: B.maximal_ideal()                                     #_
↳needs sage.libs.pari
Traceback (most recent call last):
...
ValueError: algebra is not local
```

```
>>> from sage.all import *
>>> cat = CommutativeAlgebras(GF(Integer(3))).FiniteDimensional().WithBasis()
>>> A = FiniteDimensionalAlgebra(GF(Integer(3)), [Matrix([[Integer(1),
↳Integer(0)], [Integer(0), Integer(1)]]),
...                                             Matrix([[Integer(0), Integer(1)],
↳[Integer(0), Integer(0)]]),
...                                             category=cat)
>>> A.maximal_ideal()                                     #_
↳needs sage.rings.finite_rings
Ideal (0, e1) of
Finite-dimensional algebra of degree 2 over Finite Field of size 3

>>> cat = CommutativeAlgebras(QQ).FiniteDimensional().WithBasis()
>>> B = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0),
↳Integer(0)], [Integer(0), Integer(1), Integer(0)], [Integer(0), Integer(0),
↳Integer(0)]]),
...                                     Matrix([[Integer(0), Integer(1),
↳Integer(0)], [Integer(0), Integer(0), Integer(0)], [Integer(0), Integer(0),
↳Integer(0)]]),
...                                     Matrix([[Integer(0), Integer(0),
↳Integer(0)], [Integer(0), Integer(0), Integer(0)], [Integer(0), Integer(0),
↳Integer(1)]]),
...                                     category=cat)
>>> B.maximal_ideal()                                     #_
↳needs sage.libs.pari
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
ValueError: algebra is not local
```

maximal_ideals()

Return a list consisting of all maximal ideals of `self`.

The algebra `self` has to be in the category of associative algebras.

EXAMPLES:

```
sage: cat = Algebras(GF(3)).FiniteDimensional().WithBasis()
sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1, 0], [0, 1]]),
....:                                     Matrix([[0, 1], [0, 0]])],
↪category=cat)
sage: A.maximal_ideals()                                     #_
↪needs sage.rings.finite_rings
[Ideal (e1) of Finite-dimensional algebra of degree 2 over Finite Field of_
↪size 3]

sage: cat = Algebras(QQ).FiniteDimensional().WithBasis()
sage: B = FiniteDimensionalAlgebra(QQ, [], category=cat)
sage: B.maximal_ideals()
[]
```

```
>>> from sage.all import *
>>> cat = Algebras(GF(Integer(3))).FiniteDimensional().WithBasis()
>>> A = FiniteDimensionalAlgebra(GF(Integer(3)), [Matrix([[Integer(1),_
↪Integer(0)], [Integer(0), Integer(1)]]),
...                                             Matrix([[Integer(0), Integer(1)],_
↪[Integer(0), Integer(0)]])], category=cat)
>>> A.maximal_ideals()                                     #_
↪needs sage.rings.finite_rings
[Ideal (e1) of Finite-dimensional algebra of degree 2 over Finite Field of_
↪size 3]

>>> cat = Algebras(QQ).FiniteDimensional().WithBasis()
>>> B = FiniteDimensionalAlgebra(QQ, [], category=cat)
>>> B.maximal_ideals()
[]
```

ngens()

Return the number of generators of `self`, i.e., the degree of `self` over its base field.

EXAMPLES:

```
sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1, 0], [0, 1]]),
....:                                     Matrix([[0, 1], [0, 0]])])
sage: A.ngens()
2
```

```
>>> from sage.all import *
>>> A = FiniteDimensionalAlgebra(GF(Integer(3)), [Matrix([[Integer(1),_
↪Integer(0)], [Integer(0), Integer(1)]]),
...                                             Matrix([[Integer(0), Integer(1)],_
↪[Integer(0), Integer(0)]])], category=cat)
```

(continues on next page)

(continued from previous page)

```

↪[Integer(0), Integer(0)]]])
>>> A.ngens()
2

```

one()Return the multiplicative identity element of `self`, if it exists.

EXAMPLES:

```

sage: A = FiniteDimensionalAlgebra(QQ, [])
sage: A.one()
0

sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0], [0,1]]),
....:                                     Matrix([[0,1], [-1,0]])])
sage: B.one()
e0

sage: C = FiniteDimensionalAlgebra(QQ, [Matrix([[0,0], [0,0]]),
....:                                     Matrix([[0,0], [0,0]])])
sage: C.one()
Traceback (most recent call last):
...
TypeError: algebra is not unitary

sage: D = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,1,0], [0,0,1]]),
....:                                     Matrix([[0,1,0], [0,0,0], [0,0,0]]),
....:                                     Matrix([[0,0,1], [0,0,0], [1,0,0]])])
sage: D.one()
e0

sage: E = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,1,0], [0,0,1]]),
....:                                     Matrix([[0,1,0], [0,0,0], [0,0,0]]),
....:                                     Matrix([[0,1,0], [0,0,0], [1,0,0]])])
sage: E.one()
Traceback (most recent call last):
...
TypeError: algebra is not unitary

```

```

>>> from sage.all import *
>>> A = FiniteDimensionalAlgebra(QQ, [])
>>> A.one()
0

>>> B = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0)],
↪[Integer(0), Integer(1)]]),
....:                                     Matrix([[Integer(0), Integer(1)], [-
↪Integer(1), Integer(0)]]]])
>>> B.one()
e0

>>> C = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(0), Integer(0)],
↪

```

(continues on next page)

(continued from previous page)

```

↪[Integer(0), Integer(0)]],
...
Matrix([[Integer(0), Integer(0)], ↪
↪[Integer(0), Integer(0)]])
>>> C.one()
Traceback (most recent call last):
...
TypeError: algebra is not unitary

>>> D = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0),
↪Integer(0)], [Integer(0), Integer(1), Integer(0)], [Integer(0), Integer(0),
↪Integer(1)]]),
...
Matrix([[Integer(0), Integer(1),
↪Integer(0)], [Integer(0), Integer(0), Integer(0)], [Integer(0), Integer(0),
↪Integer(0)]]),
...
Matrix([[Integer(0), Integer(0),
↪Integer(1)], [Integer(0), Integer(0), Integer(0)], [Integer(1), Integer(0),
↪Integer(0)]])])
>>> D.one()
e0

>>> E = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0),
↪Integer(0)], [Integer(0), Integer(1), Integer(0)], [Integer(0), Integer(0),
↪Integer(1)]]),
...
Matrix([[Integer(0), Integer(1),
↪Integer(0)], [Integer(0), Integer(0), Integer(0)], [Integer(0), Integer(0),
↪Integer(0)]]),
...
Matrix([[Integer(0), Integer(1),
↪Integer(0)], [Integer(0), Integer(0), Integer(0)], [Integer(1), Integer(0),
↪Integer(0)]])])
>>> E.one()
Traceback (most recent call last):
...
TypeError: algebra is not unitary

```

primary_decomposition()Return the primary decomposition of `self`.**Note**

`self` has to be in the category of unitary, commutative
and associative algebras as in the examples below.

OUTPUT:

- a list consisting of the quotient maps `self -> A`, with `A` running through the primary factors of `self`

EXAMPLES:

```

sage: cat = CommutativeAlgebras(GF(3)).FiniteDimensional().WithBasis()
sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1, 0], [0, 1]]),
.....:                                     Matrix([[0, 1], [0, 0]])], ↪

```

(continues on next page)

(continued from previous page)

```

↪category=cat)
sage: A.primary_decomposition() #_
↪needs sage.rings.finite_rings
[Morphism
  from Finite-dimensional algebra of degree 2 over Finite Field of size 3
    to Finite-dimensional algebra of degree 2 over Finite Field of size 3
  given by matrix [1 0]
                  [0 1]]

sage: cat = CommutativeAlgebras(QQ).FiniteDimensional().WithBasis()
sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,1,0], [0,0,0]]),
....:                                   Matrix([[0,1,0], [0,0,0], [0,0,0]]),
....:                                   Matrix([[0,0,0], [0,0,0], [0,0,1]])],
↪category=cat)
sage: B.primary_decomposition() #_
↪needs sage.libs.pari
[Morphism
  from Finite-dimensional algebra of degree 3 over Rational Field
    to Finite-dimensional algebra of degree 1 over Rational Field
  given by matrix [0]
                  [0]
                  [1],

Morphism
  from Finite-dimensional algebra of degree 3 over Rational Field
    to Finite-dimensional algebra of degree 2 over Rational Field
  given by matrix [1 0]
                  [0 1]
                  [0 0]]

```

```

>>> from sage.all import *
>>> cat = CommutativeAlgebras(GF(Integer(3))).FiniteDimensional().WithBasis()
>>> A = FiniteDimensionalAlgebra(GF(Integer(3)), [Matrix([[Integer(1),
↪Integer(0)], [Integer(0), Integer(1)]]),
...                                             Matrix([[Integer(0), Integer(1)],
↪[Integer(0), Integer(0)]]), category=cat)
>>> A.primary_decomposition() #_
↪needs sage.rings.finite_rings
[Morphism
  from Finite-dimensional algebra of degree 2 over Finite Field of size 3
    to Finite-dimensional algebra of degree 2 over Finite Field of size 3
  given by matrix [1 0]
                  [0 1]]

>>> cat = CommutativeAlgebras(QQ).FiniteDimensional().WithBasis()
>>> B = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0),
↪Integer(0)], [Integer(0), Integer(1), Integer(0)], [Integer(0), Integer(0),
↪Integer(0)]]),
...                                   Matrix([[Integer(0), Integer(1),
↪Integer(0)], [Integer(0), Integer(0), Integer(0)], [Integer(0), Integer(0),
↪Integer(0)]]),
...                                   Matrix([[Integer(0), Integer(0),
↪Integer(0)], [Integer(0), Integer(0), Integer(0)], [Integer(0), Integer(0),

```

(continues on next page)

(continued from previous page)

```

↪Integer(1)]]]], category=cat)
>>> B.primary_decomposition() #_
↪needs sage.libs.pari
[Morphism
  from Finite-dimensional algebra of degree 3 over Rational Field
    to Finite-dimensional algebra of degree 1 over Rational Field
  given by matrix [0]
                  [0]
                  [1],
Morphism
  from Finite-dimensional algebra of degree 3 over Rational Field
    to Finite-dimensional algebra of degree 2 over Rational Field
  given by matrix [1 0]
                  [0 1]
                  [0 0]]

```

quotient_map(ideal)

Return the quotient of self by ideal.

self has to be in the category of associative and unital algebras.

INPUT:

- ideal – a *FiniteDimensionalAlgebraIdeal*

OUTPUT:

- *FiniteDimensionalAlgebraMorphism*; the quotient homomorphism

EXAMPLES:

```

sage: cat = Algebras(GF(3)).FiniteDimensional().WithBasis()
sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1, 0], [0, 1]]),
.....:                                     Matrix([[0, 1], [0, 0]])],
.....:                                     category=cat)
sage: q0 = A.quotient_map(A.zero_ideal()); q0
Morphism
  from Finite-dimensional algebra of degree 2 over Finite Field of size 3
    to Finite-dimensional algebra of degree 2 over Finite Field of size 3
  given by matrix
  [1 0]
  [0 1]
sage: q1 = A.quotient_map(A.ideal(A.gen(1))); q1
Morphism
  from Finite-dimensional algebra of degree 2 over Finite Field of size 3
    to Finite-dimensional algebra of degree 1 over Finite Field of size 3
  given by matrix
  [1]
  [0]

```

```

>>> from sage.all import *
>>> cat = Algebras(GF(Integer(3))).FiniteDimensional().WithBasis()
>>> A = FiniteDimensionalAlgebra(GF(Integer(3)), [Matrix([[Integer(1),
↪Integer(0)], [Integer(0), Integer(1)]]),
...                                             Matrix([[Integer(0), Integer(1)],

```

(continues on next page)

(continued from previous page)

```

↪[Integer(0), Integer(0)]]],
...
category=cat)
>>> q0 = A.quotient_map(A.zero_ideal()); q0
Morphism
from Finite-dimensional algebra of degree 2 over Finite Field of size 3
to Finite-dimensional algebra of degree 2 over Finite Field of size 3
given by matrix
[1 0]
[0 1]
>>> q1 = A.quotient_map(A.ideal(A.gen(Integer(1)))); q1
Morphism
from Finite-dimensional algebra of degree 2 over Finite Field of size 3
to Finite-dimensional algebra of degree 1 over Finite Field of size 3
given by matrix
[1]
[0]

```

random_element (*args, **kwargs)

Return a random element of self.

Optional input parameters are propagated to the `random_element` method of the underlying `VectorSpace`.

EXAMPLES:

```

sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1, 0], [0, 1]]),
....:                                     Matrix([[0, 1], [0, 0]])])
sage: A.random_element() # random
e0 + 2*e1

sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,1,0], [0,0,0]]),
....:                                     Matrix([[0,1,0], [0,0,0], [0,0,0]]),
....:                                     Matrix([[0,0,0], [0,0,0], [0,0,1]])])
sage: B.random_element(num_bound=1000) # random
215/981*e0 + 709/953*e1 + 931/264*e2

```

```

>>> from sage.all import *
>>> A = FiniteDimensionalAlgebra(GF(Integer(3)), [Matrix([[Integer(1), ↪
↪Integer(0)], [Integer(0), Integer(1)]]),
...
Matrix([[Integer(0), Integer(1)], ↪
↪[Integer(0), Integer(0)]]]])
>>> A.random_element() # random
e0 + 2*e1

>>> B = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0),
↪Integer(0)], [Integer(0), Integer(1), Integer(0)], [Integer(0), Integer(0),
↪Integer(0)]]),
...
Matrix([[Integer(0), Integer(1),
↪Integer(0)], [Integer(0), Integer(0), Integer(0)], [Integer(0), Integer(0),
↪Integer(0)]]),
...
Matrix([[Integer(0), Integer(0),
↪Integer(0)], [Integer(0), Integer(0), Integer(0)], [Integer(0), Integer(0),
↪Integer(1)]]]])

```

(continues on next page)

(continued from previous page)

```
>>> B.random_element(num_bound=Integer(1000)) # random
215/981*e0 + 709/953*e1 + 931/264*e2
```

table()

Return the multiplication table of `self`, as a list of matrices for right multiplication by the basis elements.

EXAMPLES:

```
sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1, 0], [0, 1]]),
....:                                     Matrix([[0, 1], [0, 0]])])
sage: A.table()
(
 [1 0]  [0 1]
 [0 1], [0 0]
)
```

```
>>> from sage.all import *
>>> A = FiniteDimensionalAlgebra(GF(Integer(3)), [Matrix([[Integer(1),
↪Integer(0)], [Integer(0), Integer(1)]]),
....:                                     Matrix([[Integer(0), Integer(1)],
↪[Integer(0), Integer(0)]])])
>>> A.table()
(
 [1 0]  [0 1]
 [0 1], [0 0]
)
```

4.2 Elements of Finite Algebras

class `sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement`

Bases: `AlgebraElement`

Create an element of a *FiniteDimensionalAlgebra* using a multiplication table.

INPUT:

- `A` – a *FiniteDimensionalAlgebra* which will be the parent
- `elt` – vector, matrix or element of the base field (default: `None`)
- `check` – boolean (default: `True`); if `False` and `elt` is a matrix, assume that it is known to be the matrix of an element

If `elt` is a vector or a matrix consisting of a single row, it is interpreted as a vector of coordinates with respect to the given basis of `A`. If `elt` is a square matrix, it is interpreted as a multiplication matrix with respect to this basis.

EXAMPLES:

```
sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1, 0], [0, 1]]),
....:                                     Matrix([[0, 1], [0, 0]])])
sage: A(17)
2*e0
sage: A([1, 1])
e0 + e1
```

```

>>> from sage.all import *
>>> A = FiniteDimensionalAlgebra(GF(Integer(3)), [Matrix([[Integer(1), Integer(0)],
↳ [Integer(0), Integer(1)]]),
...
Matrix([[Integer(0), Integer(1)],
↳ [Integer(0), Integer(0)]]))
>>> A(Integer(17))
2*e0
>>> A([Integer(1), Integer(1)])
e0 + e1

```

`characteristic_polynomial()`

Return the characteristic polynomial of `self`.

Note

This function just returns the characteristic polynomial of the matrix of right multiplication by `self`. This may not be a very meaningful invariant if the algebra is not unitary and associative.

EXAMPLES:

```

sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,1,0], [0,0,0]]),
.....:
Matrix([[0,1,0], [0,0,0], [0,0,0]]),
.....:
Matrix([[0,0,0], [0,0,0], [0,0,1]])])
sage: B(0).characteristic_polynomial()
↳ #_
x^3
sage: b = B.random_element()
sage: f = b.characteristic_polynomial(); f
↳ #_
x^3 - 8*x^2 + 16*x
sage: f(b) == 0
↳ #_
↳ #_
True

```

```

>>> from sage.all import *
>>> B = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0),
↳ Integer(0)], [Integer(0), Integer(1), Integer(0)], [Integer(0), Integer(0),
↳ Integer(0)]]),
...
Matrix([[Integer(0), Integer(1),
↳ Integer(0)], [Integer(0), Integer(0), Integer(0)], [Integer(0), Integer(0),
↳ Integer(0)]]),
...
Matrix([[Integer(0), Integer(0),
↳ Integer(0)], [Integer(0), Integer(0), Integer(0)], [Integer(0), Integer(0),
↳ Integer(1)]])])
>>> B(Integer(0)).characteristic_polynomial()
↳ #_
↳ #_
x^3
>>> b = B.random_element()
>>> f = b.characteristic_polynomial(); f
↳ #_
↳ #_
x^3 - 8*x^2 + 16*x

```

(continues on next page)

(continued from previous page)

```
>>> f(b) == Integer(0)
↪      # needs sage.libs.pari
True
```

inverse()

Return the two-sided multiplicative inverse of `self`, if it exists.

This assumes that the algebra to which `self` belongs is associative.

Note

If an element of a finite-dimensional unitary associative algebra over a field admits a left inverse, then this is the unique left inverse, and it is also a right inverse.

EXAMPLES:

```
sage: C = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0], [0,1]]),
....:                                Matrix([[0,1], [-1,0]])])
sage: C([1,2]).inverse()
1/5*e0 - 2/5*e1
```

```
>>> from sage.all import *
>>> C = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1),Integer(0)],
↪[Integer(0),Integer(1)]]),
...                                Matrix([[Integer(0),Integer(1)], [-
↪Integer(1),Integer(0)]]]])
>>> C([Integer(1),Integer(2)]).inverse()
1/5*e0 - 2/5*e1
```

is_invertible()

Return True if `self` has a two-sided multiplicative inverse.

This assumes that the algebra to which `self` belongs is associative.

Note

If an element of a unitary finite-dimensional algebra over a field admits a left inverse, then this is the unique left inverse, and it is also a right inverse.

EXAMPLES:

```
sage: C = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0], [0,1]]),
....:                                Matrix([[0,1], [-1,0]])])
sage: C([1,2]).is_invertible()
True
sage: C(0).is_invertible()
False
```

```
>>> from sage.all import *
>>> C = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1),Integer(0)],
```

(continues on next page)

(continued from previous page)

```

↪[Integer(0), Integer(1)]],
...
Matrix([[Integer(0), Integer(1)], [-
↪Integer(1), Integer(0)]])
>>> C([Integer(1), Integer(2)]).is_invertible()
True
>>> C(Integer(0)).is_invertible()
False

```

is_nilpotent()

Return True if self is nilpotent.

EXAMPLES:

```

sage: C = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0], [0,1]]),
....:
Matrix([[0,1], [0,0]])])
sage: C([1,0]).is_nilpotent()
False
sage: C([0,1]).is_nilpotent()
True

sage: A = FiniteDimensionalAlgebra(QQ, [Matrix([0])])
sage: A([1]).is_nilpotent()
True

```

```

>>> from sage.all import *
>>> C = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0)],
↪[Integer(0), Integer(1)]],
...
Matrix([[Integer(0), Integer(1)],
↪[Integer(0), Integer(0)]])])
>>> C([Integer(1), Integer(0)]).is_nilpotent()
False
>>> C([Integer(0), Integer(1)]).is_nilpotent()
True

>>> A = FiniteDimensionalAlgebra(QQ, [Matrix([Integer(0)])])
>>> A([Integer(1)]).is_nilpotent()
True

```

is_zerodivisor()

Return True if self is a left or right zero-divisor.

EXAMPLES:

```

sage: C = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0], [0,1]]),
....:
Matrix([[0,1], [0,0]])])
sage: C([1,0]).is_zerodivisor()
False
sage: C([0,1]).is_zerodivisor()
True

```

```

>>> from sage.all import *
>>> C = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0)],

```

(continues on next page)

(continued from previous page)

```

↪[Integer(0), Integer(1)]],
...
Matrix([[Integer(0), Integer(1)],
↪[Integer(0), Integer(0)]])
>>> C([Integer(1), Integer(0)]).is_zerodivisor()
False
>>> C([Integer(0), Integer(1)]).is_zerodivisor()
True

```

left_matrix()

Return the matrix for multiplication by `self` from the left.

EXAMPLES:

```

sage: C = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,0,0], [0,0,0]]),
....:                                     Matrix([[0,1,0], [0,0,0], [0,0,0]]),
....:                                     Matrix([[0,0,0], [0,1,0], [0,0,1]])])
sage: C([1,2,0]).left_matrix()
[1 0 0]
[0 1 0]
[0 2 0]

```

```

>>> from sage.all import *
>>> C = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0),
↪Integer(0)], [Integer(0), Integer(0), Integer(0)], [Integer(0), Integer(0),
↪Integer(0)]]),
...
Matrix([[Integer(0), Integer(1),
↪Integer(0)], [Integer(0), Integer(0), Integer(0)], [Integer(0), Integer(0),
↪Integer(0)]]),
...
Matrix([[Integer(0), Integer(0),
↪Integer(0)], [Integer(0), Integer(1), Integer(0)], [Integer(0), Integer(0),
↪Integer(1)]])])
>>> C([Integer(1), Integer(2), Integer(0)]).left_matrix()
[1 0 0]
[0 1 0]
[0 2 0]

```

matrix()

Return the matrix for multiplication by `self` from the right.

EXAMPLES:

```

sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,1,0], [0,0,0]]),
....:                                     Matrix([[0,1,0], [0,0,0], [0,0,0]]),
....:                                     Matrix([[0,0,0], [0,0,0], [0,0,1]])])
sage: B(5).matrix()
[5 0 0]
[0 5 0]
[0 0 5]

```

```

>>> from sage.all import *
>>> B = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0),
↪Integer(0)], [Integer(0), Integer(1), Integer(0)], [Integer(0), Integer(0),

```

(continues on next page)

(continued from previous page)

```

↪Integer(0)])),
...
Matrix([[Integer(0), Integer(1),
↪Integer(0)], [Integer(0), Integer(0), Integer(0)], [Integer(0), Integer(0),
↪Integer(0)])),
...
Matrix([[Integer(0), Integer(0),
↪Integer(0)], [Integer(0), Integer(0), Integer(0)], [Integer(0), Integer(0),
↪Integer(1)]])])
>>> B(Integer(5)).matrix()
[5 0 0]
[0 5 0]
[0 0 5]

```

minimal_polynomial()Return the minimal polynomial of `self`.

EXAMPLES:

```

sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,1,0], [0,0,0]]),
....:
Matrix([[0,1,0], [0,0,0], [0,0,0]]),
....:
Matrix([[0,0,0], [0,0,0], [0,0,1]])])
sage: B(0).minimal_polynomial()
↪needs sage.libs.pari
x
sage: b = B.random_element()
sage: f = b.minimal_polynomial(); f # random
↪needs sage.libs.pari
x^3 + 1/2*x^2 - 7/16*x + 1/16
sage: f(b) == 0
↪needs sage.libs.pari
True

```

```

>>> from sage.all import *
>>> B = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0),
↪Integer(0)], [Integer(0), Integer(1), Integer(0)], [Integer(0), Integer(0),
↪Integer(0)])),
...
Matrix([[Integer(0), Integer(1),
↪Integer(0)], [Integer(0), Integer(0), Integer(0)], [Integer(0), Integer(0),
↪Integer(0)])),
...
Matrix([[Integer(0), Integer(0),
↪Integer(0)], [Integer(0), Integer(0), Integer(0)], [Integer(0), Integer(0),
↪Integer(1)]])])
>>> B(Integer(0)).minimal_polynomial()
↪ # needs sage.libs.pari
x
>>> b = B.random_element()
>>> f = b.minimal_polynomial(); f # random
↪needs sage.libs.pari
x^3 + 1/2*x^2 - 7/16*x + 1/16
>>> f(b) == Integer(0)
↪ # needs sage.libs.pari
True

```

monomial_coefficients (*copy=True*)

Return a dictionary whose keys are indices of basis elements in the support of `self` and whose values are the corresponding coefficients.

INPUT:

- `copy` – ignored

EXAMPLES:

```
sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0], [0,1]]),
....:                                     Matrix([[0,1], [-1,0]])])
sage: elt = B(Matrix([[1,1], [-1,1]]))
sage: elt.monomial_coefficients()
{0: 1, 1: 1}
```

```
>>> from sage.all import *
>>> B = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0)],
↪ [Integer(0), Integer(1)]]),
...                                     Matrix([[Integer(0), Integer(1)], [-
↪ Integer(1), Integer(0)]]])]
>>> elt = B(Matrix([[Integer(1), Integer(1)], [-Integer(1), Integer(1)]]))
>>> elt.monomial_coefficients()
{0: 1, 1: 1}
```

vector()

Return `self` as a vector.

EXAMPLES:

```
sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,1,0], [0,0,0]]),
....:                                     Matrix([[0,1,0], [0,0,0], [0,0,0]]),
....:                                     Matrix([[0,0,0], [0,0,0], [0,0,1]])])
sage: B(5).vector()
(5, 0, 5)
```

```
>>> from sage.all import *
>>> B = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0),
↪ Integer(0)], [Integer(0), Integer(1), Integer(0)], [Integer(0), Integer(0),
↪ Integer(0)]]),
...                                     Matrix([[Integer(0), Integer(1),
↪ Integer(0)], [Integer(0), Integer(0), Integer(0)], [Integer(0), Integer(0),
↪ Integer(0)]]),
...                                     Matrix([[Integer(0), Integer(0),
↪ Integer(0)], [Integer(0), Integer(0), Integer(0)], [Integer(0), Integer(0),
↪ Integer(1)]]])]
>>> B(Integer(5)).vector()
(5, 0, 5)
```

`sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.unpickle_FiniteDimensionalAlgebraElement`

Helper for unpickling of finite dimensional algebra elements.

4.3 Ideals of Finite Dimensional Algebras

It is necessary to use algebras in the category of associative algebras.

class sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_ideal.**FiniteDimensionalAlgebraIdeal**

Bases: `Ideal_generic`

An ideal of a `FiniteDimensionalAlgebra`.

INPUT:

- `A` – a finite-dimensional algebra
- `gens` – the generators of this ideal
- `given_by_matrix` – boolean (default: `False`); whether the basis matrix is given by `gens`

EXAMPLES:

```
sage: cat = CommutativeAlgebras(GF(3)).FiniteDimensional().WithBasis()
sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1, 0], [0, 1]]),
.....:                                     Matrix([[0, 1], [0, 0]])],
.....:                                     category=cat)
sage: A.ideal(A([0,1]))
Ideal (e1) of Finite-dimensional algebra of degree 2 over Finite Field of size 3
```

```
>>> from sage.all import *
>>> cat = CommutativeAlgebras(GF(Integer(3))).FiniteDimensional().WithBasis()
>>> A = FiniteDimensionalAlgebra(GF(Integer(3)), [Matrix([[Integer(1),
↳ Integer(0)], [Integer(0), Integer(1)]]),
...                                             Matrix([[Integer(0), Integer(1)],
↳ [Integer(0), Integer(0)]])],
...                                             category=cat)
>>> A.ideal(A([Integer(0), Integer(1)]))
Ideal (e1) of Finite-dimensional algebra of degree 2 over Finite Field of size 3
```

basis_matrix()

Return the echelonized matrix whose rows form a basis of `self`.

EXAMPLES:

```
sage: cat = CommutativeAlgebras(GF(3)).FiniteDimensional().WithBasis()
sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1, 0], [0, 1]]),
.....:                                     Matrix([[0, 1], [0, 0]])],
.....:                                     category=cat)
sage: I = A.ideal(A([1,1]))
sage: I.basis_matrix()
[1 0]
[0 1]
```

```
>>> from sage.all import *
>>> cat = CommutativeAlgebras(GF(Integer(3))).FiniteDimensional().WithBasis()
>>> A = FiniteDimensionalAlgebra(GF(Integer(3)), [Matrix([[Integer(1),
↳ Integer(0)], [Integer(0), Integer(1)]]),
...                                             Matrix([[Integer(0), Integer(1)],
↳ [Integer(0), Integer(0)]])],
...                                             category=cat)
```

(continues on next page)

(continued from previous page)

```

↪Integer(0)], [Integer(0), Integer(1)]],
...
Matrix([[Integer(0), Integer(1)],
↪[Integer(0), Integer(0)]]],
...
category=cat)
>>> I = A.ideal(A([Integer(1),Integer(1)]))
>>> I.basis_matrix()
[1 0]
[0 1]

```

vector_space()

Return self as a vector space.

EXAMPLES:

```

sage: cat = CommutativeAlgebras(GF(3)).FiniteDimensional().WithBasis()
sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1, 0], [0, 1]]),
....:                                     Matrix([[0, 1], [0, 0]]],
....:                                     category=cat)
sage: I = A.ideal(A([1,1]))
sage: I.vector_space()
Vector space of degree 2 and dimension 2 over Finite Field of size 3
Basis matrix:
[1 0]
[0 1]

```

```

>>> from sage.all import *
>>> cat = CommutativeAlgebras(GF(Integer(3))).FiniteDimensional().WithBasis()
>>> A = FiniteDimensionalAlgebra(GF(Integer(3)), [Matrix([[Integer(1),
↪Integer(0)], [Integer(0), Integer(1)]],
...
Matrix([[Integer(0), Integer(1)],
↪[Integer(0), Integer(0)]]],
...
category=cat)
>>> I = A.ideal(A([Integer(1),Integer(1)]))
>>> I.vector_space()
Vector space of degree 2 and dimension 2 over Finite Field of size 3
Basis matrix:
[1 0]
[0 1]

```

4.4 Morphisms Between Finite Algebras

```
class sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism.FiniteDimensionalAlgebraMorphism
```

Bases: `RingHomset_generic`

Set of morphisms between two finite-dimensional algebras.

zero()

Construct the zero morphism of self.

EXAMPLES:

```
sage: A = FiniteDimensionalAlgebra(QQ, [Matrix([1])])
sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1, 0], [0, 1]]),
.....:                               Matrix([[0, 1], [0, 0]])])
sage: H = Hom(A, B)
sage: H.zero()
Morphism from Finite-dimensional algebra of degree 1 over Rational Field to
Finite-dimensional algebra of degree 2 over Rational Field given by matrix
[0 0]
```

```
>>> from sage.all import *
>>> A = FiniteDimensionalAlgebra(QQ, [Matrix([Integer(1)])])
>>> B = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0)],
↪ [Integer(0), Integer(1)]]),
...                               Matrix([[Integer(0), Integer(1)],
↪ [Integer(0), Integer(0)]])])
>>> H = Hom(A, B)
>>> H.zero()
Morphism from Finite-dimensional algebra of degree 1 over Rational Field to
Finite-dimensional algebra of degree 2 over Rational Field given by matrix
[0 0]
```

class sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism.FiniteDimensionalAlgebraMorphism

Bases: RingHomomorphism_im_gens

Create a morphism between two *finite-dimensional algebras*.

INPUT:

- `parent` – the parent homset
- `f` – matrix of the underlying k -linear map
- `unitary` – boolean (default: `True`); if `True` and `check` is also `True`, raise a `ValueError` unless `A` and `B` are unitary and `f` respects unit elements
- `check` – boolean (default: `True`); check whether the given k -linear map really defines a (not necessarily unitary) k -algebra homomorphism

The algebras `A` and `B` must be defined over the same base field.

EXAMPLES:

```
sage: from sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_
↪ morphism import FiniteDimensionalAlgebraMorphism
sage: A = FiniteDimensionalAlgebra(QQ, [Matrix([[1, 0], [0, 1]]),
.....:                               Matrix([[0, 1], [0, 0]])])
sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([1])])
sage: H = Hom(A, B)
sage: f = H(Matrix([[1], [0]]))
sage: f.domain() is A
```

(continues on next page)

(continued from previous page)

```

True
sage: f.codomain() is B
True
sage: f(A.basis()[0])
e
sage: f(A.basis()[1])
0

```

```

>>> from sage.all import *
>>> from sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_
↳morphism import FiniteDimensionalAlgebraMorphism
>>> A = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0)]],
↳[Integer(0), Integer(1)])),
...
Matrix([[Integer(0), Integer(1)]],
↳[Integer(0), Integer(0)]])
>>> B = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1)]])])
>>> H = Hom(A, B)
>>> f = H(Matrix([[Integer(1)], [Integer(0)]]))
>>> f.domain() is A
True
>>> f.codomain() is B
True
>>> f(A.basis()[Integer(0)])
e
>>> f(A.basis()[Integer(1)])
0

```

Todo

An example illustrating unitary flag.

inverse_image(I)

Return the inverse image of I under self.

INPUT:

- I – FiniteDimensionalAlgebraIdeal, an ideal of self.codomain()

OUTPUT: FiniteDimensionalAlgebraIdeal, the inverse image of I under self

EXAMPLES:

```

sage: cat = CommutativeAlgebras(QQ).FiniteDimensional().WithBasis()
sage: A = FiniteDimensionalAlgebra(QQ, [Matrix([[1, 0], [0, 1]]),
....:                                     Matrix([[0, 1], [0, 0]]),
....:                                     category=cat)
sage: I = A.maximal_ideal()
↳needs sage.libs.pari
sage: q = A.quotient_map(I)
↳needs sage.libs.pari
sage: B = q.codomain()
↳needs sage.libs.pari

```

(continues on next page)

(continued from previous page)

```
sage: q.inverse_image(B.zero_ideal()) == I #_
↪needs sage.libs.pari
True
```

```
>>> from sage.all import *
>>> cat = CommutativeAlgebras(QQ).FiniteDimensional().WithBasis()
>>> A = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0)],
↪[Integer(0), Integer(1)]]),
...                               Matrix([[Integer(0), Integer(1)],
↪[Integer(0), Integer(0)]])]),
...                               category=cat)
>>> I = A.maximal_ideal() #_
↪needs sage.libs.pari
>>> q = A.quotient_map(I) #_
↪needs sage.libs.pari
>>> B = q.codomain() #_
↪needs sage.libs.pari
>>> q.inverse_image(B.zero_ideal()) == I #_
↪needs sage.libs.pari
True
```

matrix()

Return the matrix of self.

EXAMPLES:

```
sage: A = FiniteDimensionalAlgebra(QQ, [Matrix([[1, 0], [0, 1]]),
...:                               Matrix([[0, 1], [0, 0]])])
sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([1])])
sage: M = Matrix([[1], [0]])
sage: H = Hom(A, B)
sage: f = H(M)
sage: f.matrix() == M
True
```

```
>>> from sage.all import *
>>> A = FiniteDimensionalAlgebra(QQ, [Matrix([[Integer(1), Integer(0)],
↪[Integer(0), Integer(1)]]),
...                               Matrix([[Integer(0), Integer(1)],
↪[Integer(0), Integer(0)]])]))
>>> B = FiniteDimensionalAlgebra(QQ, [Matrix([Integer(1)])])
>>> M = Matrix([[Integer(1)], [Integer(0)]]))
>>> H = Hom(A, B)
>>> f = H(M)
>>> f.matrix() == M
True
```


NAMED ASSOCIATIVE ALGEBRAS

5.1 Affine nilTemperley Lieb Algebra of type A

class sage.algebras.affine_nil_temperley_lieb.**AffineNilTemperleyLiebTypeA**(*n*, *R=Integer*
Ring, prefix='a')

Bases: `CombinatorialFreeModule`

Construct the affine nilTemperley Lieb algebra of type $A_{n-1}^{(1)}$ as used in [Pos2005].

INPUT:

- *n* – positive integer

The affine nilTemperley Lieb algebra is generated by a_i for $i = 0, 1, \dots, n-1$ subject to the relations $a_i a_i = a_i a_{i+1} a_i = a_{i+1} a_i a_{i+1} = 0$ and $a_i a_j = a_j a_i$ for $i - j \not\equiv \pm 1$, where the indices are taken modulo n .

EXAMPLES:

```
sage: A = AffineNilTemperleyLiebTypeA(4)
sage: a = A.algebra_generators(); a
Finite family {0: a0, 1: a1, 2: a2, 3: a3}
sage: a[1]*a[2]*a[0] == a[1]*a[0]*a[2]
True
sage: a[0]*a[3]*a[0]
0
sage: A.an_element()
2*a0 + 1 + 3*a1 + a0*a1*a2*a3
```

```
>>> from sage.all import *
>>> A = AffineNilTemperleyLiebTypeA(Integer(4))
>>> a = A.algebra_generators(); a
Finite family {0: a0, 1: a1, 2: a2, 3: a3}
>>> a[Integer(1)]*a[Integer(2)]*a[Integer(0)] ==_
↪a[Integer(1)]*a[Integer(0)]*a[Integer(2)]
True
>>> a[Integer(0)]*a[Integer(3)]*a[Integer(0)]
0
>>> A.an_element()
2*a0 + 1 + 3*a1 + a0*a1*a2*a3
```

algebra_generator(*i*)

EXAMPLES:

```

sage: A = AffineNilTemperleyLiebTypeA(3)
sage: A.algebra_generator(1)
a1
sage: A = AffineNilTemperleyLiebTypeA(3, prefix = 't')
sage: A.algebra_generator(1)
t1

```

```

>>> from sage.all import *
>>> A = AffineNilTemperleyLiebTypeA(Integer(3))
>>> A.algebra_generator(Integer(1))
a1
>>> A = AffineNilTemperleyLiebTypeA(Integer(3), prefix = 't')
>>> A.algebra_generator(Integer(1))
t1

```

algebra_generators()

Return the generators a_i for $i = 0, 1, 2, \dots, n - 1$.

EXAMPLES:

```

sage: A = AffineNilTemperleyLiebTypeA(3)
sage: a = A.algebra_generators(); a
Finite family {0: a0, 1: a1, 2: a2}
sage: a[1]
a1

```

```

>>> from sage.all import *
>>> A = AffineNilTemperleyLiebTypeA(Integer(3))
>>> a = A.algebra_generators(); a
Finite family {0: a0, 1: a1, 2: a2}
>>> a[Integer(1)]
a1

```

has_no_braid_relation(w, i)

Assuming that w contains no relations of the form s_i^2 or $s_i s_{i+1} s_i$ or $s_i s_{i-1} s_i$, tests whether ws_i contains terms of this form.

EXAMPLES:

```

sage: A = AffineNilTemperleyLiebTypeA(5)
sage: W = A.weyl_group()
sage: s=W.simple_reflections()
sage: A.has_no_braid_relation(s[2]*s[1]*s[0]*s[4]*s[3],0)
False
sage: A.has_no_braid_relation(s[2]*s[1]*s[0]*s[4]*s[3],2)
True
sage: A.has_no_braid_relation(s[4],2)
True

```

```

>>> from sage.all import *
>>> A = AffineNilTemperleyLiebTypeA(Integer(5))
>>> W = A.weyl_group()
>>> s=W.simple_reflections()

```

(continues on next page)

(continued from previous page)

```

>>> A.has_no_braid_
↪relation(s[Integer(2)]*s[Integer(1)]*s[Integer(0)]*s[Integer(4)]*s[Integer(3)],
↪Integer(0))
False
>>> A.has_no_braid_
↪relation(s[Integer(2)]*s[Integer(1)]*s[Integer(0)]*s[Integer(4)]*s[Integer(3)],
↪Integer(2))
True
>>> A.has_no_braid_relation(s[Integer(4)], Integer(2))
True

```

index_set()

EXAMPLES:

```

sage: A = AffineNilTemperleyLiebTypeA(3)
sage: A.index_set()
(0, 1, 2)

```

```

>>> from sage.all import *
>>> A = AffineNilTemperleyLiebTypeA(Integer(3))
>>> A.index_set()
(0, 1, 2)

```

one_basis()

Return the unit of the underlying Weyl group, which index the one of this algebra, as per `AlgebrasWithBasis.ParentMethods.one_basis()`.

EXAMPLES:

```

sage: A = AffineNilTemperleyLiebTypeA(3)
sage: A.one_basis()
[1 0 0]
[0 1 0]
[0 0 1]
sage: A.one_basis() == A.weyl_group().one()
True
sage: A.one()
1

```

```

>>> from sage.all import *
>>> A = AffineNilTemperleyLiebTypeA(Integer(3))
>>> A.one_basis()
[1 0 0]
[0 1 0]
[0 0 1]
>>> A.one_basis() == A.weyl_group().one()
True
>>> A.one()
1

```

product_on_basis(w, w1)

Return $a_w a_{w1}$, where w and $w1$ are in the Weyl group assuming that w does not contain any braid relations.

EXAMPLES:

```

sage: A = AffineNilTemperleyLiebTypeA(5)
sage: W = A.weyl_group()
sage: s = W.simple_reflections()
sage: [A.product_on_basis(s[1],x) for x in s]
[a1*a0, 0, a1*a2, a3*a1, a4*a1]

sage: a = A.algebra_generators()
sage: x = a[1] * a[2]
sage: x
a1*a2
sage: x * a[1]
0
sage: x * a[2]
0
sage: x * a[0]
a1*a2*a0

sage: [x * a[1] for x in a]
[a0*a1, 0, a2*a1, a3*a1, a4*a1]

sage: w = s[1]*s[2]*s[1]
sage: A.product_on_basis(w,s[1])
Traceback (most recent call last):
...
AssertionError

```

```

>>> from sage.all import *
>>> A = AffineNilTemperleyLiebTypeA(Integer(5))
>>> W = A.weyl_group()
>>> s = W.simple_reflections()
>>> [A.product_on_basis(s[Integer(1)],x) for x in s]
[a1*a0, 0, a1*a2, a3*a1, a4*a1]

>>> a = A.algebra_generators()
>>> x = a[Integer(1)] * a[Integer(2)]
>>> x
a1*a2
>>> x * a[Integer(1)]
0
>>> x * a[Integer(2)]
0
>>> x * a[Integer(0)]
a1*a2*a0

>>> [x * a[Integer(1)] for x in a]
[a0*a1, 0, a2*a1, a3*a1, a4*a1]

>>> w = s[Integer(1)]*s[Integer(2)]*s[Integer(1)]
>>> A.product_on_basis(w,s[Integer(1)])
Traceback (most recent call last):
...

```

(continues on next page)

(continued from previous page)

AssertionError

`weyl_group()`

EXAMPLES:

```
sage: A = AffineNilTemperleyLiebTypeA(3)
sage: A.weyl_group()
Weyl Group of type ['A', 2, 1] (as a matrix group acting on the root space)
```

```
>>> from sage.all import *
>>> A = AffineNilTemperleyLiebTypeA(Integer(3))
>>> A.weyl_group()
Weyl Group of type ['A', 2, 1] (as a matrix group acting on the root space)
```

5.2 Askey-Wilson Algebras

AUTHORS:

- Travis Scrimshaw (2018-08): initial version

```
class sage.algebras.askey_wilson.AlgebraMorphism(domain, on_generators, position=0,
                                                  codomain=None, category=None)
```

Bases: `ModuleMorphismByLinearity`

An algebra morphism of the Askey-Wilson algebra defined by the images of the generators.

```
class sage.algebras.askey_wilson.AskeyWilsonAlgebra(R, q)
```

Bases: `CombinatorialFreeModule`

The (universal) Askey-Wilson algebra.

Let R be a commutative ring. The *universal Askey-Wilson algebra* is an associative unital algebra Δ_q over $R[q, q^{-1}]$ given by the generators $A, B, C, \alpha, \beta, \gamma$ that satisfy the following relations:

$$\begin{aligned}(q - q^{-1})\alpha &= (q^2 - q^{-2})A + qBC - q^{-1}CB, \\ (q - q^{-1})\beta &= (q^2 - q^{-2})B + qCA - q^{-1}AC, \\ (q - q^{-1})\gamma &= (q^2 - q^{-2})C + qAB - q^{-1}BA.\end{aligned}$$

The universal Askey-Wilson contains a *Casimir element* Ω , and the elements $\alpha, \beta, \gamma, \Omega$ generate the center of Δ_q , which is isomorphic to the polynomial ring $(R[q, q^{-1}])[\alpha, \beta, \gamma, \Omega]$ (assuming q is not a root of unity). Furthermore, the relations imply that Δ_q has a basis given by monomials $A^i B^j C^k \alpha^r \beta^s \gamma^t$, where $i, j, k, r, s, t \in \mathbf{Z}_{\geq 0}$.

The universal Askey-Wilson algebra also admits a faithful action of $PSL_2(\mathbf{Z})$ given by the automorphisms ρ (`permutation_automorphism()`):

$$A \mapsto B \mapsto C \mapsto A, \quad \alpha \mapsto \beta \mapsto \gamma \mapsto \alpha.$$

and σ (`reflection_automorphism()`):

$$A \mapsto B \mapsto A, C \mapsto C + \frac{AB - BA}{q - q^{-1}}, \quad \alpha \mapsto \beta \mapsto \alpha, \gamma \mapsto \gamma.$$

Note that $\rho^3 = \sigma^2 = 1$ and

$$\sigma(C) = C - qAB - (1 + q^2)C + q\gamma = C - qAB - q^2C + q\gamma.$$

The Askey-Wilson $AW_q(a, b, c)$ algebra is a specialization of the universal Askey-Wilson algebra by $\alpha = a$, $\beta = b$, $\gamma = c$, where $a, b, c \in R$. $AW_q(a, b, c)$ was first introduced by [Zhedanov1991] to describe the Askey-Wilson polynomials. The Askey-Wilson algebra has a central extension of Δ_q .

INPUT:

- R – a commutative ring
- q – (optional) the parameter q ; must be invertible in R

If q is not specified, then R is taken to be the base ring of a Laurent polynomial ring with variable q . Otherwise the element q must be an element of R .

Note

No check is performed to ensure q is not a root of unity, which may lead to violations of the results in [Terwilliger2011].

EXAMPLES:

We create the universal Askey-Wilson algebra and check the defining relations:

```
sage: AW = algebras.AskeyWilson(QQ)
sage: AW.inject_variables()
Defining A, B, C, a, b, g
sage: q = AW.q()
sage: (q^2-q^-2)*A + q*B*C - q^-1*C*B == (q-q^-1)*a
True
sage: (q^2-q^-2)*B + q*C*A - q^-1*A*C == (q-q^-1)*b
True
sage: (q^2-q^-2)*C + q*A*B - q^-1*B*A == (q-q^-1)*g
True
```

```
>>> from sage.all import *
>>> AW = algebras.AskeyWilson(QQ)
>>> AW.inject_variables()
Defining A, B, C, a, b, g
>>> q = AW.q()
>>> (q**Integer(2)-q**-Integer(2))*A + q*B*C - q**-Integer(1)*C*B == (q-q**-
↳Integer(1))*a
True
>>> (q**Integer(2)-q**-Integer(2))*B + q*C*A - q**-Integer(1)*A*C == (q-q**-
↳Integer(1))*b
True
>>> (q**Integer(2)-q**-Integer(2))*C + q*A*B - q**-Integer(1)*B*A == (q-q**-
↳Integer(1))*g
True
```

Next, we perform some computations:

```
sage: C * A
(q^-2)*A*C + (q^-3-q)*B - (q^-2-1)*b
sage: B^2 * g^2 * A
q^4*A*B^2*g^2 - (q^-1-q^7)*B*C*g^2 + (1-q^4)*B*g^3
+ (1-2*q^4+q^8)*A*g^2 - (q-q^3-q^5+q^7)*a*g^2
```

(continues on next page)

(continued from previous page)

```

sage: (B^3 - A) * (C^2 + q*A*B)
q^7*A*B^4 + B^3*C^2 - (q^2-q^14)*B^3*C + (q-q^7)*B^3*g - q*A^2*B
+ (3*q^3-4*q^7+q^19)*A*B^2 - A*C^2 - (1-q^6-q^8+q^14)*B^2*a
- (q^2-3*q^6+3*q^14-q^22)*B*C
+ (q^2-1+q-3*q^3-q^5+2*q^7-q^9+q^13+q^15-q^19)*B*g
+ (2*q^2-1-6*q^3+5*q^7-2*q^19+q^23)*A
- (2-2*q^2-4*q^4+4*q^6+q^8-q^10+q^12-q^14+q^16-q^18-q^20+q^22)*a

```

```

>>> from sage.all import *
>>> C * A
(q^2-2)*A*C + (q^3-q)*B - (q^2-1)*b
>>> B**Integer(2) * g**Integer(2) * A
q^4*A*B^2*g^2 - (q^2-1-q^7)*B*C*g^2 + (1-q^4)*B*g^3
+ (1-2*q^4+q^8)*A*g^2 - (q-q^3-q^5+q^7)*a*g^2
>>> (B**Integer(3) - A) * (C**Integer(2) + q*A*B)
q^7*A*B^4 + B^3*C^2 - (q^2-q^14)*B^3*C + (q-q^7)*B^3*g - q*A^2*B
+ (3*q^3-4*q^7+q^19)*A*B^2 - A*C^2 - (1-q^6-q^8+q^14)*B^2*a
- (q^2-3*q^6+3*q^14-q^22)*B*C
+ (q^2-1+q-3*q^3-q^5+2*q^7-q^9+q^13+q^15-q^19)*B*g
+ (2*q^2-1-6*q^3+5*q^7-2*q^19+q^23)*A
- (2-2*q^2-4*q^4+4*q^6+q^8-q^10+q^12-q^14+q^16-q^18-q^20+q^22)*a

```

We check the elements α , β , and γ are in the center:

```

sage: all(x * gen == gen * x for gen in AW.algebra_generators() for x in [a,b,g])
True

```

```

>>> from sage.all import *
>>> all(x * gen == gen * x for gen in AW.algebra_generators() for x in [a,b,g])
True

```

We verify that the *Casimir element* is in the center:

```

sage: Omega = AW.casimir_element()
sage: all(x * Omega == Omega * x for x in [A,B,C])
True

sage: x = AW.an_element()
sage: O2 = Omega^2
sage: x * O2 == O2 * x
True

```

```

>>> from sage.all import *
>>> Omega = AW.casimir_element()
>>> all(x * Omega == Omega * x for x in [A,B,C])
True

>>> x = AW.an_element()
>>> O2 = Omega**Integer(2)
>>> x * O2 == O2 * x
True

```

We prove Lemma 2.1 in [Terwilliger2011]:

```
sage: (q^2-q^-2) * C == (q-q^-1) * g - (q*A*B - q^-1*B*A)
True
sage: (q-q^-1) * (q^2-q^-2) * a == (B^2*A - (q^2+q^-2)*B*A*B + A*B^2
.....:                               + (q^2-q^-2)^2*A + (q-q^-1)^2*B*g)
True
sage: (q-q^-1) * (q^2-q^-2) * b == (A^2*B - (q^2+q^-2)*A*B*A + B*A^2
.....:                               + (q^2-q^-2)^2*B + (q-q^-1)^2*A*g)
True
```

```
>>> from sage.all import *
>>> (q**Integer(2)-q**-Integer(2)) * C == (q-q**-Integer(1)) * g - (q*A*B - q**-
↳ Integer(1)*B*A)
True
>>> (q-q**-Integer(1)) * (q**Integer(2)-q**-Integer(2)) * a == (B**Integer(2)*A -
↳ (q**Integer(2)+q**-Integer(2))*B*A*B + A*B**Integer(2)
...                               + (q**Integer(2)-q**-Integer(2))**Integer(2)*A
↳ + (q-q**-Integer(1))**Integer(2)*B*g)
True
>>> (q-q**-Integer(1)) * (q**Integer(2)-q**-Integer(2)) * b == (A**Integer(2)*B -
↳ (q**Integer(2)+q**-Integer(2))*A*B*A + B*A**Integer(2)
...                               + (q**Integer(2)-q**-Integer(2))**Integer(2)*B
↳ + (q-q**-Integer(1))**Integer(2)*A*g)
True
```

We prove Theorem 2.2 in [Terwilliger2011]:

```
sage: q3 = q^-2 + 1 + q^2
sage: A^3*B - q3*A^2*B*A + q3*A*B*A^2 - B*A^3 == -(q^2-q^-2)^2 * (A*B - B*A)
True
sage: B^3*A - q3*B^2*A*B + q3*B*A*B^2 - A*B^3 == -(q^2-q^-2)^2 * (B*A - A*B)
True
sage: (A^2*B^2 - B^2*A^2 + (q^2+q^-2)*(B*A*B*A-A*B*A*B)
.....: == -(q^1-q^-1)^2 * (A*B - B*A) * g)
True
```

```
>>> from sage.all import *
>>> q3 = q**-Integer(2) + Integer(1) + q**Integer(2)
>>> A**Integer(3)*B - q3*A**Integer(2)*B*A + q3*A*B*A**Integer(2) -
↳ B*A**Integer(3) == -(q**Integer(2)-q**-Integer(2))**Integer(2) * (A*B - B*A)
True
>>> B**Integer(3)*A - q3*B**Integer(2)*A*B + q3*B*A*B**Integer(2) -
↳ A*B**Integer(3) == -(q**Integer(2)-q**-Integer(2))**Integer(2) * (B*A - A*B)
True
>>> (A**Integer(2)*B**Integer(2) - B**Integer(2)*A**Integer(2) +
↳ (q**Integer(2)+q**-Integer(2))*(B*A*B*A-A*B*A*B)
... == -(q**Integer(1)-q**-Integer(1))**Integer(2) * (A*B - B*A) * g)
True
```

We construct an Askey-Wilson algebra over $\mathbf{F}_5$ at $q = 2$:

```
sage: AW = algebras.AskeyWilson(GF(5), q=2)
```

(continues on next page)

(continued from previous page)

```

sage: A,B,C,a,b,g = AW.algebra_generators()
sage: q = AW.q()
sage: Omega = AW.casimir_element()

sage: B * A
4*A*B + 2*g
sage: C * A
4*A*C + 2*b
sage: C * B
4*B*C + 2*a
sage: Omega^2
A^2*B^2*C^2 + A^3*B*C + A*B^3*C + A*B*C^3 + A^4 + 4*A^3*a
+ 2*A^2*B^2 + A^2*B*b + 2*A^2*C^2 + 4*A^2*C*g + 4*A^2*a^2
+ 4*A*B^2*a + 4*A*C^2*a + B^4 + B^3*b + 2*B^2*C^2 + 4*B^2*C*g
+ 4*B^2*b^2 + B*C^2*b + C^4 + 4*C^3*g + 4*C^2*g^2 + 2*a*b*g

sage: (q^2-q^-2)*A + q*B*C - q^-1*C*B == (q-q^-1)*a
True
sage: (q^2-q^-2)*B + q*C*A - q^-1*A*C == (q-q^-1)*b
True
sage: (q^2-q^-2)*C + q*A*B - q^-1*B*A == (q-q^-1)*g
True
sage: all(x * Omega == Omega * x for x in [A,B,C])
True

```

```

>>> from sage.all import *
>>> AW = algebras.AskeyWilson(GF(Integer(5)), q=Integer(2))
>>> A,B,C,a,b,g = AW.algebra_generators()
>>> q = AW.q()
>>> Omega = AW.casimir_element()

>>> B * A
4*A*B + 2*g
>>> C * A
4*A*C + 2*b
>>> C * B
4*B*C + 2*a
>>> Omega**Integer(2)
A^2*B^2*C^2 + A^3*B*C + A*B^3*C + A*B*C^3 + A^4 + 4*A^3*a
+ 2*A^2*B^2 + A^2*B*b + 2*A^2*C^2 + 4*A^2*C*g + 4*A^2*a^2
+ 4*A*B^2*a + 4*A*C^2*a + B^4 + B^3*b + 2*B^2*C^2 + 4*B^2*C*g
+ 4*B^2*b^2 + B*C^2*b + C^4 + 4*C^3*g + 4*C^2*g^2 + 2*a*b*g

>>> (q**Integer(2)-q**-Integer(2))*A + q*B*C - q**-Integer(1)*C*B == (q-q**-
↪ Integer(1))*a
True
>>> (q**Integer(2)-q**-Integer(2))*B + q*C*A - q**-Integer(1)*A*C == (q-q**-
↪ Integer(1))*b
True
>>> (q**Integer(2)-q**-Integer(2))*C + q*A*B - q**-Integer(1)*B*A == (q-q**-
↪ Integer(1))*g
True

```

(continues on next page)

(continued from previous page)

```
>>> all(x * Omega == Omega * x for x in [A,B,C])
True
```

REFERENCES:

- [Terwilliger2011]

algebra_generators()

Return the algebra generators of `self`.

EXAMPLES:

```
sage: AW = algebras.AskeyWilson(QQ)
sage: G = AW.algebra_generators()
sage: G['A']
A
sage: G['a']
a
sage: list(G)
[A, B, C, a, b, g]
```

```
>>> from sage.all import *
>>> AW = algebras.AskeyWilson(QQ)
>>> G = AW.algebra_generators()
>>> G['A']
A
>>> G['a']
a
>>> list(G)
[A, B, C, a, b, g]
```

an_element()

Return an element of `self`.

EXAMPLES:

```
sage: AW = algebras.AskeyWilson(QQ)
sage: AW.an_element()
(q^-3+3+2*q+q^2)*a*b*g^3 + q*A*C^2*b + 3*q^2*B*a^2*g + A
```

```
>>> from sage.all import *
>>> AW = algebras.AskeyWilson(QQ)
>>> AW.an_element()
(q^-3+3+2*q+q^2)*a*b*g^3 + q*A*C^2*b + 3*q^2*B*a^2*g + A
```

casimir_element()

Return the Casimir element of `self`.

The Casimir element of the Askey-Wilson algebra Δ_q is

$$\Omega = qABC + q^2A^2 + q^{-2}B^2 + q^2C^2 - qA\alpha - q^{-1}B\beta - qC\gamma.$$

The center $Z(\Delta_q)$ is generated by α , β , γ , and Ω .

EXAMPLES:

```
sage: AW = algebras.AskeyWilson(QQ)
sage: AW.casimir_element()
q*A*B*C + q^2*A^2 - q*A*a + (q^-2)*B^2 - (q^-1)*B*b + q^2*C^2 - q*C*g
```

```
>>> from sage.all import *
>>> AW = algebras.AskeyWilson(QQ)
>>> AW.casimir_element()
q*A*B*C + q^2*A^2 - q*A*a + (q^-2)*B^2 - (q^-1)*B*b + q^2*C^2 - q*C*g
```

We check that the Casimir element is in the center:

```
sage: Omega = AW.casimir_element()
sage: all(Omega * gen == gen * Omega for gen in AW.algebra_generators())
True
```

```
>>> from sage.all import *
>>> Omega = AW.casimir_element()
>>> all(Omega * gen == gen * Omega for gen in AW.algebra_generators())
True
```

gens()

Return the generators of `self`.

EXAMPLES:

```
sage: AW = algebras.AskeyWilson(QQ)
sage: AW.gens()
(A, B, C, a, b, g)
```

```
>>> from sage.all import *
>>> AW = algebras.AskeyWilson(QQ)
>>> AW.gens()
(A, B, C, a, b, g)
```

loop_representation()

Return the map π from `self` to 2×2 matrices over $R[\lambda, \lambda^{-1}]$, where F is the fraction field of the base ring of `self`.

Let AW be the Askey-Wilson algebra over R , and let F be the fraction field of R . Let M be the space of 2×2 matrices over $F[\lambda, \lambda^{-1}]$. Consider the following elements of M :

$$\mathcal{A} = \begin{pmatrix} \lambda & 1 - \lambda^{-1} \\ 0 & \lambda^{-1} \end{pmatrix}, \quad \mathcal{B} = \begin{pmatrix} \lambda^{-1} & 0 \\ \lambda - 1 & \lambda \end{pmatrix}, \quad \mathcal{C} = \begin{pmatrix} 1 & \lambda - 1 \\ 1 - \lambda^{-1} & \lambda + \lambda^{-1} - 1 \end{pmatrix}.$$

From Lemma 3.11 of [Terwilliger2011], we define a representation $\pi : AW \rightarrow M$ by

$$A \mapsto q\mathcal{A} + q^{-1}\mathcal{A}^{-1}, \quad B \mapsto q\mathcal{B} + q^{-1}\mathcal{B}^{-1}, \quad C \mapsto q\mathcal{C} + q^{-1}\mathcal{C}^{-1},$$

$$\alpha, \beta, \gamma \mapsto \nu I,$$

where $\nu = (q^2 + q^{-2})(\lambda + \lambda^{-1}) + (\lambda + \lambda^{-1})^2$.

We call this representation the *loop representation* as it is a representation using the loop group $SL_2(F[\lambda, \lambda^{-1}])$.

EXAMPLES:

```

sage: AW = algebras.AskeyWilson(QQ)
sage: q = AW.q()
sage: pi = AW.loop_representation()
sage: A,B,C,a,b,g = [pi(gen) for gen in AW.algebra_generators()]
sage: A
[
      1/q*lambda^-1 + q*lambda ((-q^2 + 1)/q)*lambda^-1 + ((q^2 - 1)/q)
      0
      q*lambda^-1 + 1/
      q*lambda]
sage: B
[
      q*lambda^-1 + 1/q*lambda
      0]
[((-q^2 + 1)/q) + ((q^2 - 1)/q)*lambda
      1/q*lambda^-1 + q*lambda]
sage: C
[1/q*lambda^-1 + ((q^2 - 1)/q) + 1/q*lambda
      ((q^2 - 1)/q) + ((-q^2 + 1)/q)
      q*lambda^-1 + ((-q^2 + 1)/q) +
      q*lambda]
sage: a
[lambda^-2 + ((q^4 + 1)/q^2)*lambda^-1 + 2 + ((q^4 + 1)/q^2)*lambda + lambda^
      2
      0]
[
      0 lambda^-2 + ((q^4 + 1)/q^2)*lambda^-1 + 2 + ((q^4 + 1)/q^2)*lambda +
      lambda^2]
sage: a == b
True
sage: a == g
True

sage: AW.an_element()
(q^-3+3+2*q+q^2)*a*b*g^3 + q*A*C^2*b + 3*q^2*B*a^2*g + A
sage: x = pi(AW.an_element())
sage: y = (q^-3+3+2*q+q^2)*a*b*g^3 + q*A*C^2*b + 3*q^2*B*a^2*g + A
sage: x == y
True

```

```

>>> from sage.all import *
>>> AW = algebras.AskeyWilson(QQ)
>>> q = AW.q()
>>> pi = AW.loop_representation()
>>> A,B,C,a,b,g = [pi(gen) for gen in AW.algebra_generators()]
>>> A
[
      1/q*lambda^-1 + q*lambda ((-q^2 + 1)/q)*lambda^-1 + ((q^2 - 1)/q)
      0
      q*lambda^-1 + 1/
      q*lambda]
>>> B
[
      q*lambda^-1 + 1/q*lambda
      0]
[((-q^2 + 1)/q) + ((q^2 - 1)/q)*lambda
      1/q*lambda^-1 + q*lambda]
>>> C
[1/q*lambda^-1 + ((q^2 - 1)/q) + 1/q*lambda
      ((q^2 - 1)/q) + ((-q^2 + 1)/q)
      q*lambda]

```

(continues on next page)

(continued from previous page)

```

[ ((q^2 - 1)/q)*lambda^-1 + ((-q^2 + 1)/q)      q*lambda^-1 + ((-q^2 + 1)/q) +
↪ q*lambda]
>>> a
[lambda^-2 + ((q^4 + 1)/q^2)*lambda^-1 + 2 + ((q^4 + 1)/q^2)*lambda + lambda^
↪ 2
↪ 0]
[
↪ 0 lambda^-2 + ((q^4 + 1)/q^2)*lambda^-1 + 2 + ((q^4 + 1)/q^2)*lambda +
↪ lambda^2]
>>> a == b
True
>>> a == g
True

>>> AW.an_element()
(q^-3+3+2*q+q^2)*a*b*g^3 + q*A*C^2*b + 3*q^2*B*a^2*g + A
>>> x = pi(AW.an_element())
>>> y = (q**-
↪ Integer(3)+Integer(3)+Integer(2)*q+q**Integer(2))*a*b*g**Integer(3) +
↪ q*A*C**Integer(2)*b + Integer(3)*q**Integer(2)*B*a**Integer(2)*g + A
>>> x == y
True

```

We check the defining relations of the Askey-Wilson algebra:

```

sage: A + (q*B*C - q^-1*C*B) / (q^2 - q^-2) == a / (q + q^-1)
True
sage: B + (q*C*A - q^-1*A*C) / (q^2 - q^-2) == b / (q + q^-1)
True
sage: C + (q*A*B - q^-1*B*A) / (q^2 - q^-2) == g / (q + q^-1)
True

```

```

>>> from sage.all import *
>>> A + (q*B*C - q**-Integer(1)*C*B) / (q**Integer(2) - q**-Integer(2)) == a /
↪ (q + q**-Integer(1))
True
>>> B + (q*C*A - q**-Integer(1)*A*C) / (q**Integer(2) - q**-Integer(2)) == b /
↪ (q + q**-Integer(1))
True
>>> C + (q*A*B - q**-Integer(1)*B*A) / (q**Integer(2) - q**-Integer(2)) == g /
↪ (q + q**-Integer(1))
True

```

We check Lemma 3.12 in [Terwilliger2011]:

```

sage: M = pi.codomain()
sage: la = M.base_ring().gen()
sage: p = M([[0,-1],[1,1]])
sage: s = M([[0,1],[la,0]])
sage: rho = AW.rho()
sage: sigma = AW.sigma()
sage: all(p*pi(gen)*~p == pi(rho(gen)) for gen in AW.algebra_generators())

```

(continues on next page)

(continued from previous page)

```
True
sage: all(s*pi(gen)*~s == pi(sigma(gen)) for gen in AW.algebra_generators())
True
```

```
>>> from sage.all import *
>>> M = pi.codomain()
>>> la = M.base_ring().gen()
>>> p = M([[Integer(0), -Integer(1)], [Integer(1), Integer(1)]])
>>> s = M([[Integer(0), Integer(1)], [la, Integer(0)]])
>>> rho = AW.rho()
>>> sigma = AW.sigma()
>>> all(p*pi(gen)*~p == pi(rho(gen)) for gen in AW.algebra_generators())
True
>>> all(s*pi(gen)*~s == pi(sigma(gen)) for gen in AW.algebra_generators())
True
```

one_basis()

Return the index of the basis element 1 of self.

EXAMPLES:

```
sage: AW = algebras.AskeyWilson(QQ)
sage: AW.one_basis()
(0, 0, 0, 0, 0, 0)
```

```
>>> from sage.all import *
>>> AW = algebras.AskeyWilson(QQ)
>>> AW.one_basis()
(0, 0, 0, 0, 0, 0)
```

permutation_automorphism()

Return the permutation automorphism ρ of self.

We define the automorphism ρ by

$$A \mapsto B \mapsto C \mapsto A, \quad \alpha \mapsto \beta \mapsto \gamma \mapsto \alpha.$$

EXAMPLES:

```
sage: AW = algebras.AskeyWilson(QQ)
sage: rho = AW.permutation_automorphism()
sage: [rho(gen) for gen in AW.algebra_generators()]
[B, C, A, b, g, a]

sage: AW.an_element()
(q^-3+3+2*q+q^2)*a*b*g^3 + q*A*C^2*b + 3*q^2*B*a^2*g + A
sage: rho(AW.an_element())
(q^-3+3+2*q+q^2)*a^3*b*g + q^5*A^2*B*g + 3*q^2*C*a*b^2
- (q^-2-q^6)*A*C*g + (q-q^5)*A*g^2 - (q^-3-2*q+q^5)*B*g
+ (q^-2-1-q^2+q^4)*b*g + B

sage: r3 = rho * rho * rho
sage: [r3(gen) for gen in AW.algebra_generators()]
```

(continues on next page)

(continued from previous page)

```

[A, B, C, a, b, g]
sage: r3(AW.an_element()) == AW.an_element()
True

>>> from sage.all import *
>>> AW = algebras.AskeyWilson(QQ)
>>> rho = AW.permutation_automorphism()
>>> [rho(gen) for gen in AW.algebra_generators()]
[B, C, A, b, g, a]

>>> AW.an_element()
(q^-3+3+2*q+q^2)*a*b*g^3 + q*A*C^2*b + 3*q^2*B*a^2*g + A
>>> rho(AW.an_element())
(q^-3+3+2*q+q^2)*a^3*b*g + q^5*A^2*B*g + 3*q^2*C*a*b^2
- (q^-2-q^6)*A*C*g + (q-q^5)*A*g^2 - (q^-3-2*q+q^5)*B*g
+ (q^-2-1-q^2+q^4)*b*g + B

>>> r3 = rho * rho * rho
>>> [r3(gen) for gen in AW.algebra_generators()]
[A, B, C, a, b, g]
>>> r3(AW.an_element()) == AW.an_element()
True

```

pi()

Return the map π from `self` to 2×2 matrices over $R[\lambda, \lambda^{-1}]$, where F is the fraction field of the base ring of `self`.

Let AW be the Askey-Wilson algebra over R , and let F be the fraction field of R . Let M be the space of 2×2 matrices over $F[\lambda, \lambda^{-1}]$. Consider the following elements of M :

$$\mathcal{A} = \begin{pmatrix} \lambda & 1 - \lambda^{-1} \\ 0 & \lambda^{-1} \end{pmatrix}, \quad \mathcal{B} = \begin{pmatrix} \lambda^{-1} & 0 \\ \lambda - 1 & \lambda \end{pmatrix}, \quad \mathcal{C} = \begin{pmatrix} 1 & \lambda - 1 \\ 1 - \lambda^{-1} & \lambda + \lambda^{-1} - 1 \end{pmatrix}.$$

From Lemma 3.11 of [Terwilliger2011], we define a representation $\pi : AW \rightarrow M$ by

$$A \mapsto q\mathcal{A} + q^{-1}\mathcal{A}^{-1}, \quad B \mapsto q\mathcal{B} + q^{-1}\mathcal{B}^{-1}, \quad C \mapsto q\mathcal{C} + q^{-1}\mathcal{C}^{-1},$$

$$\alpha, \beta, \gamma \mapsto \nu I,$$

where $\nu = (q^2 + q^{-2})(\lambda + \lambda^{-1}) + (\lambda + \lambda^{-1})^2$.

We call this representation the *loop representation* as it is a representation using the loop group $SL_2(F[\lambda, \lambda^{-1}])$.

EXAMPLES:

```

sage: AW = algebras.AskeyWilson(QQ)
sage: q = AW.q()
sage: pi = AW.loop_representation()
sage: A,B,C,a,b,g = [pi(gen) for gen in AW.algebra_generators()]
sage: A
[
      1/q*lambda^-1 + q*lambda ((-q^2 + 1)/q)*lambda^-1 + ((q^2 - 1)/q)
]
[
      0
      q*lambda^-1 + 1/

```

(continues on next page)

(continued from previous page)

```

↪ q*lambda]
sage: B
[
    q*lambda^-1 + 1/q*lambda                                0]
[ ((-q^2 + 1)/q) + ((q^2 - 1)/q)*lambda                    1/q*lambda^-1 + q*lambda]
sage: C
[1/q*lambda^-1 + ((q^2 - 1)/q) + 1/q*lambda                ((q^2 - 1)/q) + ((-q^2 + 1)/
↪ q)*lambda]
[ ((q^2 - 1)/q)*lambda^-1 + ((-q^2 + 1)/q)                q*lambda^-1 + ((-q^2 + 1)/q) + ↪
↪ q*lambda]
sage: a
[lambda^-2 + ((q^4 + 1)/q^2)*lambda^-1 + 2 + ((q^4 + 1)/q^2)*lambda + lambda^
↪ 2
↪ 0]
[
↪ 0 lambda^-2 + ((q^4 + 1)/q^2)*lambda^-1 + 2 + ((q^4 + 1)/q^2)*lambda + ↪
↪ lambda^2]
sage: a == b
True
sage: a == g
True

sage: AW.an_element()
(q^-3+3+2*q+q^2)*a*b*g^3 + q*A*C^2*b + 3*q^2*B*a^2*g + A
sage: x = pi(AW.an_element())
sage: y = (q^-3+3+2*q+q^2)*a*b*g^3 + q*A*C^2*b + 3*q^2*B*a^2*g + A
sage: x == y
True

```

```

>>> from sage.all import *
>>> AW = algebras.AskeyWilson(QQ)
>>> q = AW.q()
>>> pi = AW.loop_representation()
>>> A,B,C,a,b,g = [pi(gen) for gen in AW.algebra_generators()]
>>> A
[
    1/q*lambda^-1 + q*lambda ((-q^2 + 1)/q)*lambda^-1 + ((q^2 - ↪
↪ 1)/q)]
[
                                0                                q*lambda^-1 + 1/
↪ q*lambda]
>>> B
[
    q*lambda^-1 + 1/q*lambda                                0]
[ ((-q^2 + 1)/q) + ((q^2 - 1)/q)*lambda                    1/q*lambda^-1 + q*lambda]
>>> C
[1/q*lambda^-1 + ((q^2 - 1)/q) + 1/q*lambda                ((q^2 - 1)/q) + ((-q^2 + 1)/
↪ q)*lambda]
[ ((q^2 - 1)/q)*lambda^-1 + ((-q^2 + 1)/q)                q*lambda^-1 + ((-q^2 + 1)/q) + ↪
↪ q*lambda]
>>> a
[lambda^-2 + ((q^4 + 1)/q^2)*lambda^-1 + 2 + ((q^4 + 1)/q^2)*lambda + lambda^
↪ 2
↪ 0]
[
↪ 0 lambda^-2 + ((q^4 + 1)/q^2)*lambda^-1 + 2 + ((q^4 + 1)/q^2)*lambda + ↪

```

(continues on next page)

(continued from previous page)

```

↪lambda^2]
>>> a == b
True
>>> a == g
True

>>> AW.an_element()
(q^-3+3+2*q+q^2)*a*b*g^3 + q*A*C^2*b + 3*q^2*B*a^2*g + A
>>> x = pi(AW.an_element())
>>> y = (q**Integer(3)+Integer(3)+Integer(2)*q+q**Integer(2))*a*b*g**Integer(3) +
↪q*A*C**Integer(2)*b + Integer(3)*q**Integer(2)*B*a**Integer(2)*g + A
>>> x == y
True

```

We check the defining relations of the Askey-Wilson algebra:

```

sage: A + (q*B*C - q^-1*C*B) / (q^2 - q^-2) == a / (q + q^-1)
True
sage: B + (q*C*A - q^-1*A*C) / (q^2 - q^-2) == b / (q + q^-1)
True
sage: C + (q*A*B - q^-1*B*A) / (q^2 - q^-2) == g / (q + q^-1)
True

```

```

>>> from sage.all import *
>>> A + (q*B*C - q**Integer(1)*C*B) / (q**Integer(2) - q**Integer(2)) == a /
↪ (q + q**Integer(1))
True
>>> B + (q*C*A - q**Integer(1)*A*C) / (q**Integer(2) - q**Integer(2)) == b /
↪ (q + q**Integer(1))
True
>>> C + (q*A*B - q**Integer(1)*B*A) / (q**Integer(2) - q**Integer(2)) == g /
↪ (q + q**Integer(1))
True

```

We check Lemma 3.12 in [Terwilliger2011]:

```

sage: M = pi.codomain()
sage: la = M.base_ring().gen()
sage: p = M([[0, -1], [1, 1]])
sage: s = M([[0, 1], [la, 0]])
sage: rho = AW.rho()
sage: sigma = AW.sigma()
sage: all(p*pi(gen)*~p == pi(rho(gen)) for gen in AW.algebra_generators())
True
sage: all(s*pi(gen)*~s == pi(sigma(gen)) for gen in AW.algebra_generators())
True

```

```

>>> from sage.all import *
>>> M = pi.codomain()
>>> la = M.base_ring().gen()
>>> p = M([[Integer(0), -Integer(1)], [Integer(1), Integer(1)]])

```

(continues on next page)

(continued from previous page)

```

>>> s = M([[Integer(0), Integer(1)], [1a, Integer(0)]])
>>> rho = AW.rho()
>>> sigma = AW.sigma()
>>> all(p*pi(gen)*~p == pi(rho(gen)) for gen in AW.algebra_generators())
True
>>> all(s*pi(gen)*~s == pi(sigma(gen)) for gen in AW.algebra_generators())
True

```

product_on_basis(x, y)

Return the product of the basis elements indexed by x and y.

INPUT:

- x, y – tuple of length 6

EXAMPLES:

```

sage: AW = algebras.AskeyWilson(QQ)
sage: AW.product_on_basis((0, 0, 0, 0, 0, 0), (3, 5, 2, 0, 12, 3))
A^3*B^5*C^2*b^12*g^3
sage: AW.product_on_basis((0, 0, 0, 5, 3, 5), (3, 5, 2, 0, 12, 3))
A^3*B^5*C^2*a^5*b^15*g^8
sage: AW.product_on_basis((7, 0, 0, 5, 3, 5), (0, 5, 2, 0, 12, 3))
A^7*B^5*C^2*a^5*b^15*g^8
sage: AW.product_on_basis((7, 3, 0, 5, 3, 5), (0, 2, 2, 0, 12, 3))
A^7*B^5*C^2*a^5*b^15*g^8
sage: AW.product_on_basis((0, 1, 0, 5, 3, 5), (2, 0, 0, 0, 5, 3))
q^4*A^2*B*a^5*b^8*g^8 - (q^-3-q^5)*A*C*a^5*b^8*g^8
+ (1-q^4)*A*a^5*b^8*g^9 - (q^-4-2+q^4)*B*a^5*b^8*g^8
+ (q^-3-q^-1-q+q^3)*a^5*b^9*g^8
sage: AW.product_on_basis((0, 2, 1, 0, 2, 0), (1, 1, 0, 2, 1, 0))
q^4*A*B^3*C*a^2*b^3 - (q^5-q^9)*A^2*B^2*a^2*b^3
+ (q^2-q^4)*A*B^2*a^3*b^3 + (q^-3-q)*B^4*a^2*b^3
- (q^-2-1)*B^3*a^2*b^4 - (q-q^9)*B^2*C^2*a^2*b^3
+ (1-q^4)*B^2*C*a^2*b^3*g + (q^-4+2-5*q^4+2*q^12)*A*B*C*a^2*b^3
- (q^-1+q-2*q^3-2*q^5+q^7+q^9)*A*B*a^2*b^3*g
- (q^-3-q^3-2*q^5+q^7+q^9)*B*C*a^3*b^3
+ (q^-2-1-q^2+q^4)*B*a^3*b^3*g
- (q^-3-2*q+2*q^9-q^13)*A^2*a^2*b^3
+ (2*q^-2-2-3*q^2+3*q^4+q^10-q^12)*A*a^3*b^3
+ (q^-7-2*q^-3+2*q^5-q^9)*B^2*a^2*b^3
- (q^-6-q^-4-q^-2+1-q^2+q^4+q^6-q^8)*B*a^2*b^4
- (q^-7-q^-3-2*q+2*q^5+q^9-q^13)*C^2*a^2*b^3
+ (q^-6-3-2*q^2+5*q^4-q^8+q^10-q^12)*C*a^2*b^3*g
- (q^-1-2*q+2*q^5-q^7)*a^4*b^3
- (q^-3-q^-1-2*q+2*q^3+q^5-q^7)*a^2*b^3*g^2

```

```

>>> from sage.all import *
>>> AW = algebras.AskeyWilson(QQ)
>>> AW.product_on_basis((Integer(0), Integer(0), Integer(0), Integer(0),
↪ Integer(0), Integer(0)), (Integer(3), Integer(5), Integer(2), Integer(0),
↪ Integer(12), Integer(3)))
A^3*B^5*C^2*b^12*g^3

```

(continues on next page)

(continued from previous page)

```

>>> AW.product_on_basis((Integer(0), Integer(0), Integer(0), Integer(5),
↪ Integer(3), Integer(5)), (Integer(3), Integer(5), Integer(2), Integer(0),
↪ Integer(12), Integer(3)))
A^3*B^5*C^2*a^5*b^15*g^8
>>> AW.product_on_basis((Integer(7), Integer(0), Integer(0), Integer(5),
↪ Integer(3), Integer(5)), (Integer(0), Integer(5), Integer(2), Integer(0),
↪ Integer(12), Integer(3)))
A^7*B^5*C^2*a^5*b^15*g^8
>>> AW.product_on_basis((Integer(7), Integer(3), Integer(0), Integer(5),
↪ Integer(3), Integer(5)), (Integer(0), Integer(2), Integer(2), Integer(0),
↪ Integer(12), Integer(3)))
A^7*B^5*C^2*a^5*b^15*g^8
>>> AW.product_on_basis((Integer(0), Integer(1), Integer(0), Integer(5),
↪ Integer(3), Integer(5)), (Integer(2), Integer(0), Integer(0), Integer(0),
↪ Integer(5), Integer(3)))
q^4*A^2*B*a^5*b^8*g^8 - (q^-3-q^5)*A*C*a^5*b^8*g^8
+ (1-q^4)*A*a^5*b^8*g^9 - (q^-4-2+q^4)*B*a^5*b^8*g^8
+ (q^-3-q^-1-q+q^3)*a^5*b^9*g^8
>>> AW.product_on_basis((Integer(0), Integer(2), Integer(1), Integer(0),
↪ Integer(2), Integer(0)), (Integer(1), Integer(1), Integer(0), Integer(2),
↪ Integer(1), Integer(0)))
q^4*A*B^3*C*a^2*b^3 - (q^5-q^9)*A^2*B^2*a^2*b^3
+ (q^2-q^4)*A*B^2*a^3*b^3 + (q^-3-q)*B^4*a^2*b^3
- (q^-2-1)*B^3*a^2*b^4 - (q-q^9)*B^2*C^2*a^2*b^3
+ (1-q^4)*B^2*C*a^2*b^3*g + (q^-4+2-5*q^4+2*q^12)*A*B*C*a^2*b^3
- (q^-1+q-2*q^3-2*q^5+q^7+q^9)*A*B*a^2*b^3*g
- (q^-3-q^3-2*q^5+q^7+q^9)*B*C*a^3*b^3
+ (q^-2-1-q^2+q^4)*B*a^3*b^3*g
- (q^-3-2*q+2*q^9-q^13)*A^2*a^2*b^3
+ (2*q^-2-2-3*q^2+3*q^4+q^10-q^12)*A*a^3*b^3
+ (q^-7-2*q^-3+2*q^5-q^9)*B^2*a^2*b^3
- (q^-6-q^-4-q^-2+1-q^2+q^4+q^6-q^8)*B*a^2*b^4
- (q^-7-q^-3-2*q+2*q^5+q^9-q^13)*C^2*a^2*b^3
+ (q^-6-3-2*q^2+5*q^4-q^8+q^10-q^12)*C*a^2*b^3*g
- (q^-1-2*q+2*q^5-q^7)*a^4*b^3
- (q^-3-q^-1-2*q+2*q^3+q^5-q^7)*a^2*b^3*g^2

```

q()Return the parameter q of self.**EXAMPLES:**

```

sage: AW = algebras.AskeyWilson(QQ)
sage: q = AW.q()
sage: q
q
sage: q.parent()
Univariate Laurent Polynomial Ring in q over Rational Field

```

```

>>> from sage.all import *
>>> AW = algebras.AskeyWilson(QQ)
>>> q = AW.q()

```

(continues on next page)

(continued from previous page)

```
>>> q
q
>>> q.parent()
Univariate Laurent Polynomial Ring in q over Rational Field
```

reflection_automorphism()Return the reflection automorphism σ of self.We define the automorphism σ by

$$A \mapsto B \mapsto A, \quad C \mapsto C + \frac{AB - BA}{q - q^{-1}} = C - qAB - (1 + q^2)C + q\gamma,$$

$$\alpha \mapsto \beta \mapsto \alpha, \gamma \mapsto \gamma.$$

EXAMPLES:

```
sage: AW = algebras.AskeyWilson(QQ)
sage: sigma = AW.reflection_automorphism()
sage: [sigma(gen) for gen in AW.algebra_generators()]
[B, A, -q*A*B - q^2*C + q*g, b, a, g]

sage: AW.an_element()
(q^-3+3+2*q+q^2)*a*b*g^3 + q*A*C^2*b + 3*q^2*B*a^2*g + A
sage: sigma(AW.an_element())
q^9*A^2*B^3*a + (q^10+q^14)*A*B^2*C*a - (q^7+q^9)*A*B^2*a*g
+ (q^-3+3+2*q+q^2)*a*b*g^3 + (q-3*q^9+q^13+q^17)*A^2*B*a
- (q^2-q^6-q^8+q^14)*A*B*a^2 + 3*q^2*A*b^2*g + (q^5-q^9)*B^3*a
- (q^6-q^8)*B^2*a*b + q^13*B*C^2*a - 2*q^10*B*C*a*g + q^7*B*a*g^2
+ (q^2-2*q^10+q^18)*A*C*a - (q-q^7-2*q^9+2*q^11-q^15+q^17)*A*a*g
- (q^3-q^7-q^9+q^13)*C*a^2 + (q^2-q^6-2*q^8+2*q^10)*a^2*g
+ (q-3*q^5+3*q^9-q^13)*B*a - (q^2-q^4-2*q^6+2*q^8+q^10-q^12)*a*b + B

sage: s2 = sigma * sigma
sage: [s2(gen) for gen in AW.algebra_generators()]
[A, B, C, a, b, g]
sage: s2(AW.an_element()) == AW.an_element()
True
```

```
>>> from sage.all import *
>>> AW = algebras.AskeyWilson(QQ)
>>> sigma = AW.reflection_automorphism()
>>> [sigma(gen) for gen in AW.algebra_generators()]
[B, A, -q*A*B - q^2*C + q*g, b, a, g]

>>> AW.an_element()
(q^-3+3+2*q+q^2)*a*b*g^3 + q*A*C^2*b + 3*q^2*B*a^2*g + A
>>> sigma(AW.an_element())
q^9*A^2*B^3*a + (q^10+q^14)*A*B^2*C*a - (q^7+q^9)*A*B^2*a*g
+ (q^-3+3+2*q+q^2)*a*b*g^3 + (q-3*q^9+q^13+q^17)*A^2*B*a
- (q^2-q^6-q^8+q^14)*A*B*a^2 + 3*q^2*A*b^2*g + (q^5-q^9)*B^3*a
- (q^6-q^8)*B^2*a*b + q^13*B*C^2*a - 2*q^10*B*C*a*g + q^7*B*a*g^2
+ (q^2-2*q^10+q^18)*A*C*a - (q-q^7-2*q^9+2*q^11-q^15+q^17)*A*a*g
```

(continues on next page)

(continued from previous page)

```

- (q^3-q^7-q^9+q^13)*C*a^2 + (q^2-q^6-2*q^8+2*q^10)*a^2*g
+ (q^3-q^5+3*q^9-q^13)*B*a - (q^2-q^4-2*q^6+2*q^8+q^10-q^12)*a*b + B

>>> s2 = sigma * sigma
>>> [s2(gen) for gen in AW.algebra_generators()]
[A, B, C, a, b, g]
>>> s2(AW.an_element()) == AW.an_element()
True

```

rho()Return the permutation automorphism ρ of self.We define the automorphism ρ by

$$A \mapsto B \mapsto C \mapsto A, \quad \alpha \mapsto \beta \mapsto \gamma \mapsto \alpha.$$

EXAMPLES:

```

sage: AW = algebras.AskeyWilson(QQ)
sage: rho = AW.permutation_automorphism()
sage: [rho(gen) for gen in AW.algebra_generators()]
[B, C, A, b, g, a]

sage: AW.an_element()
(q^-3+3+2*q+q^2)*a*b*g^3 + q*A*C^2*b + 3*q^2*B*a^2*g + A
sage: rho(AW.an_element())
(q^-3+3+2*q+q^2)*a^3*b*g + q^5*A^2*B*g + 3*q^2*C*a*b^2
- (q^-2-q^6)*A*C*g + (q-q^5)*A*g^2 - (q^-3-2*q+q^5)*B*g
+ (q^-2-1-q^2+q^4)*b*g + B

sage: r3 = rho * rho * rho
sage: [r3(gen) for gen in AW.algebra_generators()]
[A, B, C, a, b, g]
sage: r3(AW.an_element()) == AW.an_element()
True

```

```

>>> from sage.all import *
>>> AW = algebras.AskeyWilson(QQ)
>>> rho = AW.permutation_automorphism()
>>> [rho(gen) for gen in AW.algebra_generators()]
[B, C, A, b, g, a]

>>> AW.an_element()
(q^-3+3+2*q+q^2)*a*b*g^3 + q*A*C^2*b + 3*q^2*B*a^2*g + A
>>> rho(AW.an_element())
(q^-3+3+2*q+q^2)*a^3*b*g + q^5*A^2*B*g + 3*q^2*C*a*b^2
- (q^-2-q^6)*A*C*g + (q-q^5)*A*g^2 - (q^-3-2*q+q^5)*B*g
+ (q^-2-1-q^2+q^4)*b*g + B

>>> r3 = rho * rho * rho
>>> [r3(gen) for gen in AW.algebra_generators()]
[A, B, C, a, b, g]

```

(continues on next page)

(continued from previous page)

```
>>> r3(AW.an_element()) == AW.an_element()
True
```

sigma()

Return the reflection automorphism σ of self.

We define the automorphism σ by

$$A \mapsto B \mapsto A, \quad C \mapsto C + \frac{AB - BA}{q - q^{-1}} = C - qAB - (1 + q^2)C + q\gamma,$$

$$\alpha \mapsto \beta \mapsto \alpha, \gamma \mapsto \gamma.$$

EXAMPLES:

```
sage: AW = algebras.AskeyWilson(QQ)
sage: sigma = AW.reflection_automorphism()
sage: [sigma(gen) for gen in AW.algebra_generators()]
[B, A, -q*A*B - q^2*C + q*g, b, a, g]

sage: AW.an_element()
(q^-3+3+2*q+q^2)*a*b*g^3 + q*A*C^2*b + 3*q^2*B*a^2*g + A
sage: sigma(AW.an_element())
q^9*A^2*B^3*a + (q^10+q^14)*A*B^2*C*a - (q^7+q^9)*A*B^2*a*g
+ (q^-3+3+2*q+q^2)*a*b*g^3 + (q-3*q^9+q^13+q^17)*A^2*B*a
- (q^2-q^6-q^8+q^14)*A*B*a^2 + 3*q^2*A*b^2*g + (q^5-q^9)*B^3*a
- (q^6-q^8)*B^2*a*b + q^13*B*C^2*a - 2*q^10*B*C*a*g + q^7*B*a*g^2
+ (q^2-2*q^10+q^18)*A*C*a - (q-q^7-2*q^9+2*q^11-q^15+q^17)*A*a*g
- (q^3-q^7-q^9+q^13)*C*a^2 + (q^2-q^6-2*q^8+2*q^10)*a^2*g
+ (q-3*q^5+3*q^9-q^13)*B*a - (q^2-q^4-2*q^6+2*q^8+q^10-q^12)*a*b + B

sage: s2 = sigma * sigma
sage: [s2(gen) for gen in AW.algebra_generators()]
[A, B, C, a, b, g]
sage: s2(AW.an_element()) == AW.an_element()
True
```

```
>>> from sage.all import *
>>> AW = algebras.AskeyWilson(QQ)
>>> sigma = AW.reflection_automorphism()
>>> [sigma(gen) for gen in AW.algebra_generators()]
[B, A, -q*A*B - q^2*C + q*g, b, a, g]

>>> AW.an_element()
(q^-3+3+2*q+q^2)*a*b*g^3 + q*A*C^2*b + 3*q^2*B*a^2*g + A
>>> sigma(AW.an_element())
q^9*A^2*B^3*a + (q^10+q^14)*A*B^2*C*a - (q^7+q^9)*A*B^2*a*g
+ (q^-3+3+2*q+q^2)*a*b*g^3 + (q-3*q^9+q^13+q^17)*A^2*B*a
- (q^2-q^6-q^8+q^14)*A*B*a^2 + 3*q^2*A*b^2*g + (q^5-q^9)*B^3*a
- (q^6-q^8)*B^2*a*b + q^13*B*C^2*a - 2*q^10*B*C*a*g + q^7*B*a*g^2
+ (q^2-2*q^10+q^18)*A*C*a - (q-q^7-2*q^9+2*q^11-q^15+q^17)*A*a*g
- (q^3-q^7-q^9+q^13)*C*a^2 + (q^2-q^6-2*q^8+2*q^10)*a^2*g
+ (q-3*q^5+3*q^9-q^13)*B*a - (q^2-q^4-2*q^6+2*q^8+q^10-q^12)*a*b + B
```

(continues on next page)

(continued from previous page)

```

>>> s2 = sigma * sigma
>>> [s2(gen) for gen in AW.algebra_generators()]
[A, B, C, a, b, g]
>>> s2(AW.an_element()) == AW.an_element()
True

```

some_elements()

Return some elements of `self`.

EXAMPLES:

```

sage: AW = algebras.AskeyWilson(QQ)
sage: AW.some_elements()
(A, B, C, a, b, g, 1,
 (q-3+3+2*q+q2)*a*b*g3 + q*A*C2*b + 3*q2*B*a2*g + A,
 q*A*B*C + q2*A2 - q*A*a + (q-2)*B2 - (q-1)*B*b + q2*C2 - q*C*g)

```

```

>>> from sage.all import *
>>> AW = algebras.AskeyWilson(QQ)
>>> AW.some_elements()
(A, B, C, a, b, g, 1,
 (q-3+3+2*q+q2)*a*b*g3 + q*A*C2*b + 3*q2*B*a2*g + A,
 q*A*B*C + q2*A2 - q*A*a + (q-2)*B2 - (q-1)*B*b + q2*C2 - q*C*g)

```

5.3 Diagram and Partition Algebras

AUTHORS:

- Mike Hansen (2007): Initial version
- Stephen Doty, Aaron Lauve, George H. Seelinger (2012): Implementation of partition, Brauer, Temperley–Lieb, and ideal partition algebras
- Stephen Doty, Aaron Lauve, George H. Seelinger (2015): Implementation of `*Diagram` classes and other methods to improve diagram algebras.
- Mike Zabrocki (2018): Implementation of individual element diagram classes
- Aaron Lauve, Mike Zabrocki (2018): Implementation of orbit basis for Partition algebra.
- Travis Scrimshaw (2024): Implemented the Potts representation of the partition algebra.

class `sage.combinat.diagram_algebras.AbstractPartitionDiagram` (*parent*, *d*, *check=True*)

Bases: `AbstractSetPartition`

Abstract base class for partition diagrams.

This class represents a single partition diagram, that is used as a basis key for a diagram algebra element. A partition diagram should be a partition of the set $\{1, \dots, k, -1, \dots, -k\}$. Each such set partition is regarded as a graph on nodes $\{1, \dots, k, -1, \dots, -k\}$ arranged in two rows, with nodes $1, \dots, k$ in the top row from left to right and with nodes $-1, \dots, -k$ in the bottom row from left to right, and an edge connecting two nodes if and only if the nodes lie in the same subset of the set partition.

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: pd = da.AbstractPartitionDiagrams(2)
sage: pd1 = da.AbstractPartitionDiagram(pd, [[1,2],[-1,-2]])
sage: pd2 = da.AbstractPartitionDiagram(pd, [[1,2],[-1,-2]])
sage: pd1
{{-2, -1}, {1, 2}}
sage: pd1 == pd2
True
sage: pd1 == [[1,2],[-1,-2]]
True
sage: pd1 == ((-2,-1), (2,1))
True
sage: pd1 == SetPartition([[1,2],[-1,-2]])
True
sage: pd3 = da.AbstractPartitionDiagram(pd, [[1,-2],[-1,2]])
sage: pd1 == pd3
False
sage: pd4 = da.AbstractPartitionDiagram(pd, [[1,2],[3,4]])
Traceback (most recent call last):
...
ValueError: {{1, 2}, {3, 4}} does not represent two rows of vertices of order 2

```

```

>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> pd = da.AbstractPartitionDiagrams(Integer(2))
>>> pd1 = da.AbstractPartitionDiagram(pd, [[Integer(1),Integer(2)],[-Integer(1),-
↪Integer(2)]])
>>> pd2 = da.AbstractPartitionDiagram(pd, [[Integer(1),Integer(2)],[-Integer(1),-
↪Integer(2)]])
>>> pd1
{{-2, -1}, {1, 2}}
>>> pd1 == pd2
True
>>> pd1 == [[Integer(1),Integer(2)],[-Integer(1),-Integer(2)]]
True
>>> pd1 == ((-Integer(2),-Integer(1)), (Integer(2),Integer(1)))
True
>>> pd1 == SetPartition([[Integer(1),Integer(2)],[-Integer(1),-Integer(2)]])
True
>>> pd3 = da.AbstractPartitionDiagram(pd, [[Integer(1),-Integer(2)],[-Integer(1),
↪Integer(2)]])
>>> pd1 == pd3
False
>>> pd4 = da.AbstractPartitionDiagram(pd, [[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]])
Traceback (most recent call last):
...
ValueError: {{1, 2}, {3, 4}} does not represent two rows of vertices of order 2

```

base_diagram()

Return the underlying implementation of the diagram.

OUTPUT: tuple of tuples of integers

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.AbstractPartitionDiagrams(2)
sage: pd([[1,2],[-1,-2]]).base_diagram() == ((-2,-1),(1,2))
True
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> pd = da.AbstractPartitionDiagrams(Integer(2))
>>> pd([[Integer(1),Integer(2)],[-Integer(1),-Integer(2)]]).base_diagram() == \
↪ ((-Integer(2),-Integer(1)),(Integer(1),Integer(2)))
True
```

check()

Check the validity of the input for the diagram.

compose(other, check=True)

Compose self with other.

The composition of two diagrams X and Y is given by placing X on top of Y and removing all loops.

OUTPUT:

A tuple where the first entry is the composite diagram and the second entry is how many loop were removed.

Note

This is not really meant to be called directly, but it works to call it this way if desired.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.AbstractPartitionDiagrams(2)
sage: pd([[1,2],[-1,-2]]).compose(pd([[1,2],[-1,-2]]))
(((-2,-1), {1, 2}), 1)
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> pd = da.AbstractPartitionDiagrams(Integer(2))
>>> pd([[Integer(1),Integer(2)],[-Integer(1),-Integer(2)]]).
↪ compose(pd([[Integer(1),Integer(2)],[-Integer(1),-Integer(2)]]))
(((-2,-1), {1, 2}), 1)
```

count_blocks_of_size(n)

Count the number of blocks of a given size.

INPUT:

- n – positive integer

EXAMPLES:

```
sage: from sage.combinat.diagram_algebras import PartitionDiagram
sage: pd = PartitionDiagram([[1,-3,-5],[2,4],[3,-1,-2],[5],[-4]])
```

(continues on next page)

(continued from previous page)

```

sage: pd.count_blocks_of_size(1)
2
sage: pd.count_blocks_of_size(2)
1
sage: pd.count_blocks_of_size(3)
2

```

```

>>> from sage.all import *
>>> from sage.combinat.diagram_algebras import PartitionDiagram
>>> pd = PartitionDiagram([[Integer(1), -Integer(3), -Integer(5)], [Integer(2),
↪ Integer(4)], [Integer(3), -Integer(1), -Integer(2)], [Integer(5)], [-
↪ Integer(4)]]])
>>> pd.count_blocks_of_size(Integer(1))
2
>>> pd.count_blocks_of_size(Integer(2))
1
>>> pd.count_blocks_of_size(Integer(3))
2

```

diagram()

Return the underlying implementation of the diagram.

OUTPUT: tuple of tuples of integers

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: pd = da.AbstractPartitionDiagrams(2)
sage: pd([[1, 2], [-1, -2]]).base_diagram() == ((-2, -1), (1, 2))
True

```

```

>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> pd = da.AbstractPartitionDiagrams(Integer(2))
>>> pd([[Integer(1), Integer(2)], [-Integer(1), -Integer(2)]]).base_diagram() ==
↪ ((-Integer(2), -Integer(1)), (Integer(1), Integer(2)))
True

```

dual()

Return the dual diagram of `self` by flipping it top-to-bottom.

EXAMPLES:

```

sage: from sage.combinat.diagram_algebras import PartitionDiagram
sage: D = PartitionDiagram([[1, -1], [2, -2, -3], [3]])
sage: D.dual()
{{-3}, {-2, 2, 3}, {-1, 1}}

```

```

>>> from sage.all import *
>>> from sage.combinat.diagram_algebras import PartitionDiagram
>>> D = PartitionDiagram([[Integer(1), -Integer(1)], [Integer(2), -Integer(2), -
↪ Integer(3)], [Integer(3)]]])

```

(continues on next page)

(continued from previous page)

```
>>> D.dual()
{{-3}, {-2, 2, 3}, {-1, 1}}
```

is_planar()

Test if the diagram `self` is planar.

A diagram element is planar if the graph of the nodes is planar.

EXAMPLES:

```
sage: from sage.combinat.diagram_algebras import BrauerDiagram
sage: BrauerDiagram([[1,-2],[2,-1]]).is_planar()
False
sage: BrauerDiagram([[1,-1],[2,-2]]).is_planar()
True
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram_algebras import BrauerDiagram
>>> BrauerDiagram([[Integer(1),-Integer(2)],[Integer(2),-Integer(1)]]).is_
↳ planar()
False
>>> BrauerDiagram([[Integer(1),-Integer(1)],[Integer(2),-Integer(2)]]).is_
↳ planar()
True
```

order()

Return the maximum entry in the diagram element.

A diagram element will be a partition of the set $\{-1, -2, \dots, -k, 1, 2, \dots, k\}$. The order of the diagram element is the value k .

EXAMPLES:

```
sage: from sage.combinat.diagram_algebras import PartitionDiagram
sage: PartitionDiagram([[1,-1],[2,-2,-3],[3]]).order()
3
sage: PartitionDiagram([[1,-1]]).order()
1
sage: PartitionDiagram([[1,-3,-5],[2,4],[3,-1,-2],[5],[-4]]).order()
5
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram_algebras import PartitionDiagram
>>> PartitionDiagram([[Integer(1),-Integer(1)],[Integer(2),-Integer(2),-
↳ Integer(3)],[Integer(3)]]).order()
3
>>> PartitionDiagram([[Integer(1),-Integer(1)]]).order()
1
>>> PartitionDiagram([[Integer(1),-Integer(3),-Integer(5)],[Integer(2),
↳ Integer(4)],[Integer(3),-Integer(1),-Integer(2)],[Integer(5)],[
↳ Integer(4)]]).order()
5
```

propagating_number()

Return the propagating number of the diagram.

The propagating number is the number of blocks with both a positive and negative number.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.AbstractPartitionDiagrams(2)
sage: d1 = pd([[1, -2], [2, -1]])
sage: d1.propagating_number()
2
sage: d2 = pd([[1, 2], [-2, -1]])
sage: d2.propagating_number()
0
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> pd = da.AbstractPartitionDiagrams(Integer(2))
>>> d1 = pd([[Integer(1), -Integer(2)], [Integer(2), -Integer(1)]])
>>> d1.propagating_number()
2
>>> d2 = pd([[Integer(1), Integer(2)], [-Integer(2), -Integer(1)]])
>>> d2.propagating_number()
0
```

set_partition()

Return the underlying implementation of the diagram as a set of sets.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.AbstractPartitionDiagrams(2)
sage: X = pd([[1, 2], [-1, -2]]).set_partition(); X
{{-2, -1}, {1, 2}}
sage: X.parent()
Set partitions
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> pd = da.AbstractPartitionDiagrams(Integer(2))
>>> X = pd([[Integer(1), Integer(2)], [-Integer(1), -Integer(2)]])
↪ X.set_partition(); X
{{-2, -1}, {1, 2}}
>>> X.parent()
Set partitions
```

class sage.combinat.diagram_algebras.**AbstractPartitionDiagrams**(*order*, *category=None*)

Bases: `Parent`, `UniqueRepresentation`

This is an abstract base class for partition diagrams.

The primary use of this class is to serve as basis keys for diagram algebras, but diagrams also have properties in their own right. Furthermore, this class is meant to be extended to create more efficient contains methods.

INPUT:

- `order` – integer or integer $+1/2$; the order of the diagrams
- `category` – (default: `FiniteEnumeratedSets()`) the category

All concrete classes should implement attributes

- `_name` – the name of the class
- `_diagram_func` – an iterator function that takes the order as its only input

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.PartitionDiagrams(2)
sage: pd
Partition diagrams of order 2
sage: pd.an_element() in pd
True
sage: elm = pd([[1,2],[-1,-2]])
sage: elm in pd
True
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> pd = da.PartitionDiagrams(Integer(2))
>>> pd
Partition diagrams of order 2
>>> pd.an_element() in pd
True
>>> elm = pd([[Integer(1), Integer(2)], [-Integer(1), -Integer(2)]])
>>> elm in pd
True
```

Element

alias of *AbstractPartitionDiagram*

class `sage.combinat.diagram_algebras.BrauerAlgebra` (*k*, *q*, *base_ring*, *prefix*)

Bases: *SubPartitionAlgebra*, *UnitDiagramMixin*

A Brauer algebra.

The Brauer algebra of rank *k* is an algebra with basis indexed by the collection of set partitions of $\{1, \dots, k, -1, \dots, -k\}$ with block size 2.

This algebra is a subalgebra of the partition algebra. For more information, see *PartitionAlgebra*.

INPUT:

- *k* – rank of the algebra
- *q* – the deformation parameter *q*
- *base_ring* – (default: `None`) a ring containing *q*; if `None` then just takes the parent of *q*
- *prefix* – (default: `'B'`) a label for the basis elements

EXAMPLES:

We now define the Brauer algebra of rank 2 with parameter *x* over **Z**:

```

sage: R.<x> = ZZ[]
sage: B = BrauerAlgebra(2, x, R)
sage: B
Brauer Algebra of rank 2 with parameter x
over Univariate Polynomial Ring in x over Integer Ring
sage: B.basis()
Lazy family (Term map from Brauer diagrams of order 2 to Brauer Algebra
of rank 2 with parameter x over Univariate Polynomial Ring in x
over Integer Ring(i))_{i in Brauer diagrams of order 2}
sage: B.basis().keys()
Brauer diagrams of order 2
sage: B.basis().keys() ([-2, 1], [2, -1])
{{-2, 1}, {-1, 2}}
sage: b = B.basis().list(); b
[B{{-2, -1}, {1, 2}}, B{{-2, 1}, {-1, 2}}, B{{-2, 2}, {-1, 1}}]
sage: b[0]
B{{-2, -1}, {1, 2}}
sage: b[0]^2
x*B{{-2, -1}, {1, 2}}
sage: b[0]^5
x^4*B{{-2, -1}, {1, 2}}

```

```

>>> from sage.all import *
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> B = BrauerAlgebra(Integer(2), x, R)
>>> B
Brauer Algebra of rank 2 with parameter x
over Univariate Polynomial Ring in x over Integer Ring
>>> B.basis()
Lazy family (Term map from Brauer diagrams of order 2 to Brauer Algebra
of rank 2 with parameter x over Univariate Polynomial Ring in x
over Integer Ring(i))_{i in Brauer diagrams of order 2}
>>> B.basis().keys()
Brauer diagrams of order 2
>>> B.basis().keys() ([-Integer(2), Integer(1)], [Integer(2), -Integer(1)])
{{-2, 1}, {-1, 2}}
>>> b = B.basis().list(); b
[B{{-2, -1}, {1, 2}}, B{{-2, 1}, {-1, 2}}, B{{-2, 2}, {-1, 1}}]
>>> b[Integer(0)]
B{{-2, -1}, {1, 2}}
>>> b[Integer(0)]**Integer(2)
x*B{{-2, -1}, {1, 2}}
>>> b[Integer(0)]**Integer(5)
x^4*B{{-2, -1}, {1, 2}}

```

Note, also that since the symmetric group algebra is contained in the Brauer algebra, there is also a conversion between the two.

```

sage: R.<x> = ZZ[]
sage: B = BrauerAlgebra(2, x, R)
sage: S = SymmetricGroupAlgebra(R, 2)
sage: S([2, 1]) * B([1, -1], [2, -2])
B{{-2, 1}, {-1, 2}}

```



```
>>> from sage.all import *
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> B = BrauerAlgebra(Integer(2), x, R)
>>> S = SymmetricGroupAlgebra(R, Integer(2))
>>> S([Integer(2), Integer(1)]) * B([[Integer(1), -Integer(1)], [Integer(2), -
↳ Integer(2)]])
B{{-2, 1}, {-1, 2}}
```

jucys_murphy(*j*)

Return the *j*-th generalized Jucys-Murphy element of *self*.

The *j*-th Jucys-Murphy element of a Brauer algebra is simply the *j*-th Jucys-Murphy element of the symmetric group algebra with an extra $(z - 1)/2$ term, where *z* is the parameter of the Brauer algebra.

REFERENCES:

EXAMPLES:

```
sage: # needs sage.symbolic
sage: z = var('z')
sage: B = BrauerAlgebra(3, z)
sage: B.jucys_murphy(1)
(1/2*z-1/2)*B{{-3, 3}, {-2, 2}, {-1, 1}}
sage: B.jucys_murphy(3)
-B{{-3, -2}, {-1, 1}, {2, 3}} - B{{-3, -1}, {-2, 2}, {1, 3}}
+ B{{-3, 1}, {-2, 2}, {-1, 3}} + B{{-3, 2}, {-2, 3}, {-1, 1}}
+ (1/2*z-1/2)*B{{-3, 3}, {-2, 2}, {-1, 1}}
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> z = var('z')
>>> B = BrauerAlgebra(Integer(3), z)
>>> B.jucys_murphy(Integer(1))
(1/2*z-1/2)*B{{-3, 3}, {-2, 2}, {-1, 1}}
>>> B.jucys_murphy(Integer(3))
-B{{-3, -2}, {-1, 1}, {2, 3}} - B{{-3, -1}, {-2, 2}, {1, 3}}
+ B{{-3, 1}, {-2, 2}, {-1, 3}} + B{{-3, 2}, {-2, 3}, {-1, 1}}
+ (1/2*z-1/2)*B{{-3, 3}, {-2, 2}, {-1, 1}}
```

options = Current options for Brauer diagram - display: normal

class sage.combinat.diagram_algebras.**BrauerDiagram**(parent, *d*, check=True)

Bases: *AbstractPartitionDiagram*

A Brauer diagram.

A Brauer diagram for an integer *k* is a partition of the set $\{1, \dots, k, -1, \dots, -k\}$ with block size 2.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(2)
sage: bd1 = bd([[1, 2], [-1, -2]])
sage: bd2 = bd([[1, 2, -1, -2]])
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: all blocks of {{-2, -1, 1, 2}} must be of size 2
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> bd = da.BrauerDiagrams(Integer(2))
>>> bd1 = bd([[Integer(1), Integer(2)], [-Integer(1), -Integer(2)]])
>>> bd2 = bd([[Integer(1), Integer(2), -Integer(1), -Integer(2)]])
Traceback (most recent call last):
...
ValueError: all blocks of {{-2, -1, 1, 2}} must be of size 2
```

bijection_on_free_nodes (*two_line=False*)

Return the induced bijection - as a list of $(x, f(x))$ values - from the free nodes on the top at the Brauer diagram to the free nodes at the bottom of *self*.

OUTPUT:

If *two_line* is True, then the output is the induced bijection as a two-row list (inputs, outputs).

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(3)
sage: elm = bd([[1, 2], [-2, -3], [3, -1]])
sage: elm.bijection_on_free_nodes()
[[3, -1]]
sage: elm2 = bd([[1, -2], [2, -3], [3, -1]])
sage: elm2.bijection_on_free_nodes(two_line=True)
[[1, 2, 3], [-2, -3, -1]]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> bd = da.BrauerDiagrams(Integer(3))
>>> elm = bd([[Integer(1), Integer(2)], [-Integer(2), -Integer(3)], [Integer(3), -
↪ Integer(1)]])
>>> elm.bijection_on_free_nodes()
[[3, -1]]
>>> elm2 = bd([[Integer(1), -Integer(2)], [Integer(2), -Integer(3)], [Integer(3), -
↪ Integer(1)]])
>>> elm2.bijection_on_free_nodes(two_line=True)
[[1, 2, 3], [-2, -3, -1]]
```

check()

Check the validity of the input for *self*.

involution_permutation_triple (*curt=True*)

Return the involution permutation triple of *self*.

From Graham-Lehrer (see *BrauerDiagrams*), a Brauer diagram is a triple (D_1, D_2, π) , where:

- D_1 is a partition of the top nodes;
- D_2 is a partition of the bottom nodes;
- π is the induced permutation on the free nodes.

INPUT:

- `curt` – boolean (default: `True`); if `True`, then return bijection on free nodes as a one-line notation (standardized to look like a permutation), else, return the honest mapping, a list of pairs $(i, -j)$ describing the bijection on free nodes

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(3)
sage: elm = bd([[1, 2], [-2, -3], [3, -1]])
sage: elm.involution_permutation_triple()
((1, 2)], [(-3, -2)], [1])
sage: elm.involution_permutation_triple(curt=False)
((1, 2)], [(-3, -2)], [[3, -1]])
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> bd = da.BrauerDiagrams(Integer(3))
>>> elm = bd([[Integer(1), Integer(2)], [-Integer(2), -Integer(3)], [Integer(3), -
↪Integer(1)]])
>>> elm.involution_permutation_triple()
((1, 2)], [(-3, -2)], [1])
>>> elm.involution_permutation_triple(curt=False)
((1, 2)], [(-3, -2)], [[3, -1]])
```

`is_elementary_symmetric()`

Check if is elementary symmetric.

Let (D_1, D_2, π) be the Graham-Lehrer representation of the Brauer diagram d . We say d is *elementary symmetric* if $D_1 = D_2$ and π is the identity.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(3)
sage: elm = bd([[1, 2], [-1, -2], [3, -3]])
sage: elm.is_elementary_symmetric()
True
sage: elm2 = bd([[1, 2], [-1, -3], [3, -2]])
sage: elm2.is_elementary_symmetric()
False
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> bd = da.BrauerDiagrams(Integer(3))
>>> elm = bd([[Integer(1), Integer(2)], [-Integer(1), -Integer(2)], [Integer(3), -
↪Integer(3)]])
>>> elm.is_elementary_symmetric()
True
>>> elm2 = bd([[Integer(1), Integer(2)], [-Integer(1), -Integer(3)], [Integer(3), -
↪Integer(2)]])
>>> elm2.is_elementary_symmetric()
False
```

`options = Current options for Brauer diagram - display: normal`

`perm()`

Return the induced bijection on the free nodes of `self` in one-line notation, re-indexed and treated as a permutation.

See also

[`bijection\_on\_free\_nodes\(\)`](#)

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(3)
sage: elm = bd([[1,2],[-2,-3],[3,-1]])
sage: elm.perm()
[1]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> bd = da.BrauerDiagrams(Integer(3))
>>> elm = bd([[Integer(1),Integer(2)],[-Integer(2),-Integer(3)],[Integer(3),-
↪Integer(1)]])
>>> elm.perm()
[1]
```

class `sage.combinat.diagram_algebras.BrauerDiagrams` (*order*, *category=None*)

Bases: [`AbstractPartitionDiagrams`](#)

This class represents all Brauer diagrams of integer or integer $+1/2$ order. For more information on Brauer diagrams, see [`BrauerAlgebra`](#).

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(2); bd
Brauer diagrams of order 2
sage: bd.list()
[{{-2, -1}, {1, 2}}, {{-2, 1}, {-1, 2}}, {{-2, 2}, {-1, 1}}]

sage: bd = da.BrauerDiagrams(5/2); bd
Brauer diagrams of order 5/2
sage: bd.list()
[{{-3, 3}, {-2, -1}, {1, 2}},
 {{-3, 3}, {-2, 1}, {-1, 2}},
 {{-3, 3}, {-2, 2}, {-1, 1}}]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> bd = da.BrauerDiagrams(Integer(2)); bd
Brauer diagrams of order 2
>>> bd.list()
[{{-2, -1}, {1, 2}}, {{-2, 1}, {-1, 2}}, {{-2, 2}, {-1, 1}}]
```

(continues on next page)

(continued from previous page)

```

>>> bd = da.BrauerDiagrams(Integer(5)/Integer(2)); bd
Brauer diagrams of order 5/2
>>> bd.list()
[{{-3, 3}, {-2, -1}, {1, 2}},
 {{-3, 3}, {-2, 1}, {-1, 2}},
 {{-3, 3}, {-2, 2}, {-1, 1}}]

```

Elementalias of *BrauerDiagram***cardinality()**Return the cardinality of *self*.The number of Brauer diagrams of integer order k is $(2k - 1)!!$.

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(3)
sage: bd.cardinality()
15

sage: bd = da.BrauerDiagrams(7/2)
sage: bd.cardinality()
15

```

```

>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> bd = da.BrauerDiagrams(Integer(3))
>>> bd.cardinality()
15

>>> bd = da.BrauerDiagrams(Integer(7)/Integer(2))
>>> bd.cardinality()
15

```

from_involution_permutation_triple(*D1_D2_pi*)Construct a Brauer diagram of *self* from an involution permutation triple.

A Brauer diagram can be represented as a triple where the first entry is a list of arcs on the top row of the diagram, the second entry is a list of arcs on the bottom row of the diagram, and the third entry is a permutation on the remaining nodes. This triple is called the *involution permutation triple*. For more information, see [GL1996].

INPUT:

- *D1_D2_pi* – list or tuple where the first entry is a list of arcs on the top of the diagram, the second entry is a list of arcs on the bottom of the diagram, and the third entry is a permutation on the free nodes.

REFERENCES:

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(4)
sage: bd.from_involution_permutation_triple([[[1,2]], [[3,4]], [2,1]])
[{-4, -3}, {-2, 3}, {-1, 4}, {1, 2}]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> bd = da.BrauerDiagrams(Integer(4))
>>> bd.from_involution_permutation_triple([[[Integer(1), Integer(2)]],
↪ [[Integer(3), Integer(4)]], [Integer(2), Integer(1)]])
[{-4, -3}, {-2, 3}, {-1, 4}, {1, 2}]
```

options = Current options for Brauer diagram - display: normal

symmetric_diagrams (*l=None, perm=None*)

Return the list of Brauer diagrams with symmetric placement of *l* arcs, and with free nodes permuted according to *perm*.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(4)
sage: bd.symmetric_diagrams(l=1, perm=[2,1])
[{-4, -2}, {-3, 1}, {-1, 3}, {2, 4}},
[{-4, -3}, {-2, 1}, {-1, 2}, {3, 4}},
[{-4, -1}, {-3, 2}, {-2, 3}, {1, 4}},
[{-4, 2}, {-3, -1}, {-2, 4}, {1, 3}},
[{-4, 3}, {-3, 4}, {-2, -1}, {1, 2}},
[{-4, 1}, {-3, -2}, {-1, 4}, {2, 3}]]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> bd = da.BrauerDiagrams(Integer(4))
>>> bd.symmetric_diagrams(l=Integer(1), perm=[Integer(2), Integer(1)])
[{-4, -2}, {-3, 1}, {-1, 3}, {2, 4}},
[{-4, -3}, {-2, 1}, {-1, 2}, {3, 4}},
[{-4, -1}, {-3, 2}, {-2, 3}, {1, 4}},
[{-4, 2}, {-3, -1}, {-2, 4}, {1, 3}},
[{-4, 3}, {-3, 4}, {-2, -1}, {1, 2}},
[{-4, 1}, {-3, -2}, {-1, 4}, {2, 3}]]
```

class sage.combinat.diagram_algebras.**DiagramAlgebra** (*k, q, base_ring, prefix, diagrams, category=None*)

Bases: `CombinatorialFreeModule`

Abstract class for diagram algebras and is not designed to be used directly.

class **Element**

Bases: `IndexedFreeModuleElement`

An element of a diagram algebra.

This subclass provides a few additional methods for partition algebra elements. Most element methods are already implemented elsewhere.

diagram()

Return the underlying diagram of `self` if `self` is a basis element. Raises an error if `self` is not a basis element.

EXAMPLES:

```
sage: R.<x> = ZZ[]
sage: P = PartitionAlgebra(2, x, R)
sage: elt = 3*P([[1,2],[-2,-1]])
sage: elt.diagram()
{{-2, -1}, {1, 2}}
```

```
>>> from sage.all import *
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> P = PartitionAlgebra(Integer(2), x, R)
>>> elt = Integer(3)*P([[Integer(1),Integer(2)],[-Integer(2),-
↳Integer(1)]])
>>> elt.diagram()
{{-2, -1}, {1, 2}}
```

diagrams()

Return the diagrams in the support of `self`.

EXAMPLES:

```
sage: R.<x> = ZZ[]
sage: P = PartitionAlgebra(2, x, R)
sage: elt = 3*P([[1,2],[-2,-1]]) + P([[1,2],[-2], [-1]])
sage: sorted(elt.diagrams(), key=str)
[{{-2, -1}, {1, 2}}, {{-2}, {-1}, {1, 2}}]
```

```
>>> from sage.all import *
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> P = PartitionAlgebra(Integer(2), x, R)
>>> elt = Integer(3)*P([[Integer(1),Integer(2)],[-Integer(2),-
↳Integer(1)]]) + P([[Integer(1),Integer(2)],[-Integer(2)], [-
↳Integer(1)]])
>>> sorted(elt.diagrams(), key=str)
[{{-2, -1}, {1, 2}}, {{-2}, {-1}, {1, 2}}]
```

order()

Return the order of `self`.

The order of a partition algebra is defined as half of the number of nodes in the diagrams.

EXAMPLES:

```
sage: q = var('q') #_
↳needs sage.symbolic
sage: PA = PartitionAlgebra(2, q) #_
↳needs sage.symbolic
sage: PA.order() #_
↳needs sage.symbolic
2
```

```

>>> from sage.all import *
>>> q = var('q') #
↪needs sage.symbolic
>>> PA = PartitionAlgebra(Integer(2), q) #
↪ # needs sage.symbolic
>>> PA.order() #
↪needs sage.symbolic
2

```

set_partitions()

Return the collection of underlying set partitions indexing the basis elements of a given diagram algebra.

Todo

Is this really necessary? deprecate?

class sage.combinat.diagram_algebras.**DiagramBasis**(*k, q, base_ring, prefix, diagrams, category=None*)

Bases: *DiagramAlgebra*

Abstract base class for diagram algebras in the diagram basis.

product_on_basis(*d1, d2*)

Return the product $D_{d_1} D_{d_2}$ by two basis diagrams.

class sage.combinat.diagram_algebras.**HalfTemperleyLiebDiagrams**(*order, defects*)

Bases: *UniqueRepresentation, Parent*

Half diagrams for the Temperley-Lieb algebra cell modules.

class **Element**(*parent, d, check=True*)

Bases: *AbstractPartitionDiagram*

check()

Check the validity of the input of self.

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: htld = da.HalfTemperleyLiebDiagrams(7, 3)
sage: htld([[1,2], [3,4]]) # indirect doctest
{{1, 2}, {3, 4}}
sage: htld([[1,2], [-1, -2]]) # indirect doctest
Traceback (most recent call last):
...
ValueError: {{-2, -1}, {1, 2}} does not represent a half TL diagram of
↪order 7
sage: htld([[1,2,3], [4,5]]) # indirect doctest
Traceback (most recent call last):
...
ValueError: all blocks of {{1, 2, 3}, {4, 5}} must be of size 2
sage: htld([[1,2], [3,4], [5,6]]) # indirect doctest
Traceback (most recent call last):
...
ValueError: {{1, 2}, {3, 4}, {5, 6}} does not have 3 defects

```

(continues on next page)

(continued from previous page)

```
sage: htld([[1,3], [2,4]]) # indirect doctest
Traceback (most recent call last):
...
ValueError: {{1, 3}, {2, 4}} is not planar
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> htld = da.HalfTemperleyLiebDiagrams(Integer(7), Integer(3))
>>> htld([[Integer(1),Integer(2)], [Integer(3),Integer(4)]]) # indirect_
↳doctest
{{1, 2}, {3, 4}}
>>> htld([[Integer(1),Integer(2)], [-Integer(1), -Integer(2)]]) #_
↳indirect doctest
Traceback (most recent call last):
...
ValueError: {{-2, -1}, {1, 2}} does not represent a half TL diagram of_
↳order 7
>>> htld([[Integer(1),Integer(2),Integer(3)], [Integer(4),Integer(5)]])
↳# indirect doctest
Traceback (most recent call last):
...
ValueError: all blocks of {{1, 2, 3}, {4, 5}} must be of size 2
>>> htld([[Integer(1),Integer(2)], [Integer(3),Integer(4)], [Integer(5),
↳Integer(6)]]) # indirect doctest
Traceback (most recent call last):
...
ValueError: {{1, 2}, {3, 4}, {5, 6}} does not have 3 defects
>>> htld([[Integer(1),Integer(3)], [Integer(2),Integer(4)]]) # indirect_
↳doctest
Traceback (most recent call last):
...
ValueError: {{1, 3}, {2, 4}} is not planar
```

defects()

Return the defects of self.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: htld = da.HalfTemperleyLiebDiagrams(7, 3)
sage: d = htld([[1, 2], [4, 5]])
sage: d.defects()
frozenset({3, 6, 7})
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> htld = da.HalfTemperleyLiebDiagrams(Integer(7), Integer(3))
>>> d = htld([[Integer(1), Integer(2)], [Integer(4), Integer(5)]])
>>> d.defects()
frozenset({3, 6, 7})
```

cardinality()

Return the cardinality of self.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: htld = da.HalfTemperleyLiebDiagrams(7, 3)
sage: htld.cardinality()
14
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> htld = da.HalfTemperleyLiebDiagrams(Integer(7), Integer(3))
>>> htld.cardinality()
14
```

class sage.combinat.diagram_algebras.**IdealDiagram**(parent, d, check=True)

Bases: [AbstractPartitionDiagram](#)

The element class for a ideal diagram.

An ideal diagram for an integer k is a partition of the set $\{1, \dots, k, -1, \dots, -k\}$ where the propagating number is strictly smaller than the order.

EXAMPLES:

```
sage: from sage.combinat.diagram_algebras import IdealDiagrams as IDs
sage: IDs(2)
Ideal diagrams of order 2
sage: IDs(2).list()
[{{-2, -1, 1, 2}},
 {{-2, 1, 2}, {-1}},
 {{-2}, {-1, 1, 2}},
 {{-2, -1}, {1, 2}},
 {{-2}, {-1}, {1, 2}},
 {{-2, -1, 1}, {2}},
 {{-2, 1}, {-1}, {2}},
 {{-2, -1, 2}, {1}},
 {{-2, 2}, {-1}, {1}},
 {{-2}, {-1, 1}, {2}},
 {{-2}, {-1, 2}, {1}},
 {{-2, -1}, {1}, {2}},
 {{-2}, {-1}, {1}, {2}}]

sage: from sage.combinat.diagram_algebras import PartitionDiagrams as PDs
sage: PDs(4).cardinality() == factorial(4) + IDs(4).cardinality()
True
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram_algebras import IdealDiagrams as IDs
>>> IDs(Integer(2))
Ideal diagrams of order 2
>>> IDs(Integer(2)).list()
[{{-2, -1, 1, 2}},
 {{-2, 1, 2}, {-1}},
 {{-2}, {-1, 1, 2}},
 {{-2, -1}, {1, 2}},
```

(continues on next page)

(continued from previous page)

```

[{{-2}, {-1}, {1, 2}},
 {{-2, -1, 1}, {2}},
 {{-2, 1}, {-1}, {2}},
 {{-2, -1, 2}, {1}},
 {{-2, 2}, {-1}, {1}},
 {{-2}, {-1, 1}, {2}},
 {{-2}, {-1, 2}, {1}},
 {{-2, -1}, {1}, {2}},
 {{-2}, {-1}, {1}, {2}}]

>>> from sage.combinat.diagram_algebras import PartitionDiagrams as PDs
>>> PDs(Integer(4)).cardinality() == factorial(Integer(4)) + IDs(Integer(4)).
↳cardinality()
True

```

check()

Check the validity of the input for `self`.

class `sage.combinat.diagram_algebras.IdealDiagrams` (*order*, *category=None*)

Bases: *AbstractPartitionDiagrams*

All “ideal” diagrams of integer or integer +1/2 order.

If k is an integer then an ideal diagram of order k is a partition diagram of order k with propagating number less than k .

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: id = da.IdealDiagrams(3)
sage: id.an_element() in id
True
sage: id.cardinality() == len(id.list())
True
sage: da.IdealDiagrams(3/2).list()
[{{-2, -1, 1, 2}},
 {{-2, 1, 2}, {-1}},
 {{-2, -1, 2}, {1}},
 {{-2, 2}, {-1}, {1}}]

```

```

>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> id = da.IdealDiagrams(Integer(3))
>>> id.an_element() in id
True
>>> id.cardinality() == len(id.list())
True
>>> da.IdealDiagrams(Integer(3)/Integer(2)).list()
[{{-2, -1, 1, 2}},
 {{-2, 1, 2}, {-1}},
 {{-2, -1, 2}, {1}},
 {{-2, 2}, {-1}, {1}}]

```

Element

alias of *IdealDiagram*

class sage.combinat.diagram_algebras.**OrbitBasis** (alg)

Bases: *DiagramAlgebra*

The orbit basis of the partition algebra.

Let D_π represent the diagram basis element indexed by the partition π , then (see equations (2.14), (2.17) and (2.18) of [BH2017])

$$D_\pi = \sum_{\tau \geq \pi} O_\tau,$$

where the sum is over all partitions τ which are coarser than π and O_τ is the orbit basis element indexed by the partition τ .

If $\mu_{2k}(\pi, \tau)$ represents the Moebius function of the partition lattice, then

$$O_\pi = \sum_{\tau \geq \pi} \mu_{2k}(\pi, \tau) D_\tau.$$

If τ is a partition of ℓ blocks and the i -th block of τ is a union of b_i blocks of π , then

$$\mu_{2k}(\pi, \tau) = \prod_{i=1}^{\ell} (-1)^{b_i-1} (b_i - 1)!.$$

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: P2 = PartitionAlgebra(2, x, R)
sage: O2 = P2.orbit_basis(); O2
Orbit basis of Partition Algebra of rank 2 with parameter x over
Univariate Polynomial Ring in x over Rational Field
sage: oa = O2([[1], [-1], [2, -2]]); ob = O2([[-1, -2, 2], [1]]); oa, ob
(O{{-2, 2}}, {-1}, {1}}, O{{-2, -1, 2}, {1}})
sage: oa * ob
(x-2)*O{{-2, -1, 2}, {1}}
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> P2 = PartitionAlgebra(Integer(2), x, R)
>>> O2 = P2.orbit_basis(); O2
Orbit basis of Partition Algebra of rank 2 with parameter x over
Univariate Polynomial Ring in x over Rational Field
>>> oa = O2([[Integer(1)], [-Integer(1)], [Integer(2), -Integer(2)]]); ob = O2([[-
↪ Integer(1), -Integer(2), Integer(2)], [Integer(1)]]); oa, ob
(O{{-2, 2}, {-1}, {1}}, O{{-2, -1, 2}, {1}})
>>> oa * ob
(x-2)*O{{-2, -1, 2}, {1}}
```

We can convert between the two bases:

```
sage: pa = P2(oa); pa
2*P{{-2, -1, 1, 2}} - P{{-2, -1, 2}, {1}} - P{{-2, 1, 2}, {-1}}
+ P{{-2, 2}, {-1}, {1}} - P{{-2, 2}, {-1, 1}}
```

(continues on next page)

(continued from previous page)

```
sage: pa * ob
(-x+2)*P{{-2, -1, 1, 2}} + (x-2)*P{{-2, -1, 2}, {1}}
sage: _ == pa * P2(ob)
True
sage: O2(pa * ob)
(x-2)*O{{-2, -1, 2}, {1}}
```

```
>>> from sage.all import *
>>> pa = P2(oa); pa
2*P{{-2, -1, 1, 2}} - P{{-2, -1, 2}, {1}} - P{{-2, 1, 2}, {-1}}
+ P{{-2, 2}, {-1}, {1}} - P{{-2, 2}, {-1, 1}}
>>> pa * ob
(-x+2)*P{{-2, -1, 1, 2}} + (x-2)*P{{-2, -1, 2}, {1}}
>>> _ == pa * P2(ob)
True
>>> O2(pa * ob)
(x-2)*O{{-2, -1, 2}, {1}}
```

Note that the unit in the orbit basis is not a single diagram, in contrast to the natural diagram basis:

```
sage: P2.one()
P{{-2, 2}, {-1, 1}}
sage: O2.one()
O{{-2, -1, 1, 2}} + O{{-2, 2}, {-1, 1}}
sage: O2.one() == P2.one()
True
```

```
>>> from sage.all import *
>>> P2.one()
P{{-2, 2}, {-1, 1}}
>>> O2.one()
O{{-2, -1, 1, 2}} + O{{-2, 2}, {-1, 1}}
>>> O2.one() == P2.one()
True
```

class Element

Bases: *Element*

to_diagram_basis()

Expand self in the natural diagram basis of the partition algebra.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: P = PartitionAlgebra(2, x, R)
sage: O = P.orbit_basis()
sage: elt = O.an_element(); elt
3*O{{-2}, {-1, 1, 2}} + 2*O{{-2, -1, 1, 2}} + 2*O{{-2, 1, 2}, {-1}}
sage: elt.to_diagram_basis()
3*P{{-2}, {-1, 1, 2}} - 3*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
sage: pp = P.an_element(); pp
3*P{{-2}, {-1, 1, 2}} + 2*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
```

(continues on next page)

(continued from previous page)

```
sage: op = pp.to_orbit_basis(); op
3*O{{-2}, {-1, 1, 2}} + 7*O{{-2, -1, 1, 2}} + 2*O{{-2, 1, 2}, {-1}}
sage: pp == op.to_diagram_basis()
True
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> P = PartitionAlgebra(Integer(2), x, R)
>>> O = P.orbit_basis()
>>> elt = O.an_element(); elt
3*O{{-2}, {-1, 1, 2}} + 2*O{{-2, -1, 1, 2}} + 2*O{{-2, 1, 2}, {-1}}
>>> elt.to_diagram_basis()
3*P{{-2}, {-1, 1, 2}} - 3*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
>>> pp = P.an_element(); pp
3*P{{-2}, {-1, 1, 2}} + 2*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
>>> op = pp.to_orbit_basis(); op
3*O{{-2}, {-1, 1, 2}} + 7*O{{-2, -1, 1, 2}} + 2*O{{-2, 1, 2}, {-1}}
>>> pp == op.to_diagram_basis()
True
```

diagram_basis()

Return the associated partition algebra of `self` in the diagram basis.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: O2 = PartitionAlgebra(2, x, R).orbit_basis()
sage: P2 = O2.diagram_basis(); P2
Partition Algebra of rank 2 with parameter x over Univariate
Polynomial Ring in x over Rational Field
sage: o2 = O2.an_element(); o2
3*O{{-2}, {-1, 1, 2}} + 2*O{{-2, -1, 1, 2}} + 2*O{{-2, 1, 2}, {-1}}
sage: P2(o2)
3*P{{-2}, {-1, 1, 2}} - 3*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> O2 = PartitionAlgebra(Integer(2), x, R).orbit_basis()
>>> P2 = O2.diagram_basis(); P2
Partition Algebra of rank 2 with parameter x over Univariate
Polynomial Ring in x over Rational Field
>>> o2 = O2.an_element(); o2
3*O{{-2}, {-1, 1, 2}} + 2*O{{-2, -1, 1, 2}} + 2*O{{-2, 1, 2}, {-1}}
>>> P2(o2)
3*P{{-2}, {-1, 1, 2}} - 3*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
```

one()

Return the element 1 of the partition algebra in the orbit basis.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: P2 = PartitionAlgebra(2, x, R)
```

(continues on next page)

(continued from previous page)

```

sage: O2 = P2.orbit_basis()
sage: O2.one()
O{{-2, -1, 1, 2}} + O{{-2, 2}, {-1, 1}}

```

```

>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> P2 = PartitionAlgebra(Integer(2), x, R)
>>> O2 = P2.orbit_basis()
>>> O2.one()
O{{-2, -1, 1, 2}} + O{{-2, 2}, {-1, 1}}

```

product_on_basis(d1, d2)

Return the product $O_{d_1}O_{d_2}$ of two elements in the orbit basis self.

EXAMPLES:

```

sage: R.<x> = QQ[]
sage: OP = PartitionAlgebra(2, x, R).orbit_basis()
sage: SP = OP.basis().keys(); sp = SP([[ -2, -1, 1, 2]])
sage: OP.product_on_basis(sp, sp)
O{{-2, -1, 1, 2}}
sage: o1 = OP.one(); o2 = OP([]); o3 = OP.an_element()
sage: o2 == o1
False
sage: o1 * o1 == o1
True
sage: o3 * o1 == o1 * o3 and o3 * o1 == o3
True
sage: o4 = (3*OP([[ -2, -1, 1], [2]]) + 2*OP([[ -2, -1, 1, 2]])
....:      + 2*OP([[ -2, -1, 2], [1]]))
sage: o4 * o4
6*O{{-2, -1, 1}, {2}} + 4*O{{-2, -1, 1, 2}} + 4*O{{-2, -1, 2}, {1}}

```

```

>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> OP = PartitionAlgebra(Integer(2), x, R).orbit_basis()
>>> SP = OP.basis().keys(); sp = SP([[-Integer(2), -Integer(1), Integer(1),
↳ Integer(2)]])
>>> OP.product_on_basis(sp, sp)
O{{-2, -1, 1, 2}}
>>> o1 = OP.one(); o2 = OP([]); o3 = OP.an_element()
>>> o2 == o1
False
>>> o1 * o1 == o1
True
>>> o3 * o1 == o1 * o3 and o3 * o1 == o3
True
>>> o4 = (Integer(3)*OP([[-Integer(2), -Integer(1), Integer(1)],
↳ [Integer(2)]]) + Integer(2)*OP([[-Integer(2), -Integer(1), Integer(1),
↳ Integer(2)]])
...      + Integer(2)*OP([[-Integer(2), -Integer(1), Integer(2)],
↳ [Integer(1)]])

```

(continues on next page)

(continued from previous page)

```
>>> o4 * o4
6*O{{-2, -1, 1}, {2}} + 4*O{{-2, -1, 1, 2}} + 4*O{{-2, -1, 2}, {1}}
```

We compute Examples 4.5 in [BH2017]:

```
sage: R.<x> = QQ[]
sage: P = PartitionAlgebra(3,x); O = P.orbit_basis()
sage: O[[1,2,3],[-1,-2,-3]] * O[[1,2,3],[-1,-2,-3]]
(x-2)*O{{-3, -2, -1}, {1, 2, 3}} + (x-1)*O{{-3, -2, -1, 1, 2, 3}}

sage: P = PartitionAlgebra(4,x); O = P.orbit_basis()
sage: O[[1],[-1],[2,3],[4,-2],[-3,-4]] * O[[1],[2,-2],[3,4],[-1,-3],[-4]]
(x^2-11*x+30)*O{{-4}, {-3, -1}, {-2, 4}, {1}, {2, 3}}
+ (x^2-9*x+20)*O{{-4}, {-3, -1, 1}, {-2, 4}, {2, 3}}
+ (x^2-9*x+20)*O{{-4}, {-3, -1, 2, 3}, {-2, 4}, {1}}
+ (x^2-9*x+20)*O{{-4, 1}, {-3, -1}, {-2, 4}, {2, 3}}
+ (x^2-7*x+12)*O{{-4, 1}, {-3, -1, 2, 3}, {-2, 4}}
+ (x^2-9*x+20)*O{{-4, 2, 3}, {-3, -1}, {-2, 4}, {1}}
+ (x^2-7*x+12)*O{{-4, 2, 3}, {-3, -1, 1}, {-2, 4}}

sage: O[[1,-1],[2,-2],[3],[4,-3],[-4]] * O[[1,-2],[2],[3,-1],[4],[-3],[-4]]
(x-6)*O{{-4}, {-3}, {-2, 1}, {-1, 4}, {2}, {3}}
+ (x-5)*O{{-4}, {-3, 3}, {-2, 1}, {-1, 4}, {2}}
+ (x-5)*O{{-4, 3}, {-3}, {-2, 1}, {-1, 4}, {2}}

sage: P = PartitionAlgebra(6,x); O = P.orbit_basis()
sage: (O[[1,-2,-3],[2,4],[3,5,-6],[6],[-1],[-4,-5]]
..... * O[[1,-2],[2,3],[4],[5],[6,-4,-5,-6],[-1,-3]])
0

sage: (O[[1,-2],[2,-3],[3,5],[4,-5],[6,-4],[-1],[-6]]
..... * O[[1,-2],[2,-1],[3,-4],[4,-6],[5,-3],[6,-5]])
O{{-6, 6}, {-5}, {-4, 2}, {-3, 4}, {-2}, {-1, 1}, {3, 5}}
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> P = PartitionAlgebra(Integer(3),x); O = P.orbit_basis()
>>> O[[Integer(1),Integer(2),Integer(3)],[-Integer(1),-Integer(2),-
↳Integer(3)]] * O[[Integer(1),Integer(2),Integer(3)],[-Integer(1),-
↳Integer(2),-Integer(3)]]
(x-2)*O{{-3, -2, -1}, {1, 2, 3}} + (x-1)*O{{-3, -2, -1, 1, 2, 3}}

>>> P = PartitionAlgebra(Integer(4),x); O = P.orbit_basis()
>>> O[[Integer(1)],[-Integer(1)],[Integer(2),Integer(3)],[Integer(4),-
↳Integer(2)],[-Integer(3),-Integer(4)]] * O[[Integer(1)],[Integer(2),-
↳Integer(2)],[Integer(3),Integer(4)],[-Integer(1),-Integer(3)],[-Integer(4)]]
(x^2-11*x+30)*O{{-4}, {-3, -1}, {-2, 4}, {1}, {2, 3}}
+ (x^2-9*x+20)*O{{-4}, {-3, -1, 1}, {-2, 4}, {2, 3}}
+ (x^2-9*x+20)*O{{-4}, {-3, -1, 2, 3}, {-2, 4}, {1}}
+ (x^2-9*x+20)*O{{-4, 1}, {-3, -1}, {-2, 4}, {2, 3}}
+ (x^2-7*x+12)*O{{-4, 1}, {-3, -1, 2, 3}, {-2, 4}}
+ (x^2-9*x+20)*O{{-4, 2, 3}, {-3, -1}, {-2, 4}, {1}}
```

(continues on next page)

(continued from previous page)

```

+ (x^2-7*x+12)*O{{-4, 2, 3}, {-3, -1, 1}, {-2, 4}}

>>> O[[Integer(1),-Integer(1)],[Integer(2),-Integer(2)],[Integer(3)],
↪[Integer(4),-Integer(3)],[Integer(4)]] * O[[Integer(1),-Integer(2)],
↪[Integer(2)],[Integer(3),-Integer(1)],[Integer(4)],[Integer(3)],[Integer(4)]]
(x-6)*O{{-4}, {-3}, {-2, 1}, {-1, 4}, {2}, {3}}
+ (x-5)*O{{-4}, {-3, 3}, {-2, 1}, {-1, 4}, {2}}
+ (x-5)*O{{-4, 3}, {-3}, {-2, 1}, {-1, 4}, {2}}

>>> P = PartitionAlgebra(Integer(6),x); O = P.orbit_basis()
>>> (O[[Integer(1),-Integer(2),-Integer(3)],[Integer(2),Integer(4)],
↪[Integer(3),Integer(5),-Integer(6)],[Integer(6)],[Integer(1)],[Integer(4),
↪-Integer(5)]]
... * O[[Integer(1),-Integer(2)],[Integer(2),Integer(3)],[Integer(4)],
↪[Integer(5)],[Integer(6),-Integer(4),-Integer(5),-Integer(6)],[Integer(1),-
↪Integer(3)]]
0

>>> (O[[Integer(1),-Integer(2)],[Integer(2),-Integer(3)],[Integer(3),
↪Integer(5)],[Integer(4),-Integer(5)],[Integer(6),-Integer(4)],[Integer(1),
↪-Integer(6)]]
... * O[[Integer(1),-Integer(2)],[Integer(2),-Integer(1)],[Integer(3),-
↪Integer(4)],[Integer(4),-Integer(6)],[Integer(5),-Integer(3)],[Integer(6),-
↪Integer(5)]]
O{{-6, 6}, {-5}, {-4, 2}, {-3, 4}, {-2}, {-1, 1}, {3, 5}}

```

REFERENCES:

- [BH2017]

class sage.combinat.diagram_algebras.**PartitionAlgebra** (*k*, *q*, *base_ring*, *prefix*)

Bases: *DiagramBasis*, *UnitDiagramMixin*

A partition algebra.

A partition algebra of rank k over a given ground ring R is an algebra with (R -module) basis indexed by the collection of set partitions of $\{1, \dots, k, -1, \dots, -k\}$. Each such set partition can be represented by a graph on nodes $\{1, \dots, k, -1, \dots, -k\}$ arranged in two rows, with nodes $1, \dots, k$ in the top row from left to right and with nodes $-1, \dots, -k$ in the bottom row from left to right, and edges drawn such that the connected components of the graph are precisely the parts of the set partition. (This choice of edges is often not unique, and so there are often many graphs representing one and the same set partition; the representation nevertheless is useful and vivid. We often speak of “diagrams” to mean graphs up to such equivalence of choices of edges; of course, we could just as well speak of set partitions.)

There is not just one partition algebra of given rank over a given ground ring, but rather a whole family of them, indexed by the elements of R . More precisely, for every $q \in R$, the partition algebra of rank k over R with parameter q is defined to be the R -algebra with basis the collection of all set partitions of $\{1, \dots, k, -1, \dots, -k\}$, where the product of two basis elements is given by the rule

$$a \cdot b = q^N (a \circ b),$$

where $a \circ b$ is the composite set partition obtained by placing the diagram (i.e., graph) of a above the diagram of b , identifying the bottom row nodes of a with the top row nodes of b , and omitting any closed “loops” in the middle. The number N is the number of connected components formed by the omitted loops.

The parameter q is a deformation parameter. Taking $q = 1$ produces the semigroup algebra (over the base ring) of the partition monoid, in which the product of two set partitions is simply given by their composition.

The partition algebra is regarded as an example of a “diagram algebra” due to the fact that its natural basis is given by certain graphs often called diagrams.

There are a number of predefined elements for the partition algebra. We define the cup/cap pair by $a()$. The simple transpositions are denoted $s()$. Finally, we define elements $e()$, where if $i = (2r + 1)/2$, then $e(i)$ contains the blocks $\{r + 1\}$ and $\{-r - 1\}$ and if $i \in \mathbb{Z}$, then e_i contains the block $\{-i, -i - 1, i, i + 1\}$, with all other blocks being $\{-j, j\}$. So we have:

```
sage: P = PartitionAlgebra(4, 0)
sage: P.a(2)
P{{-4, 4}, {-3, -2}, {-1, 1}, {2, 3}}
sage: P.e(3/2)
P{{-4, 4}, {-3, 3}, {-2}, {-1, 1}, {2}}
sage: P.e(2)
P{{-4, 4}, {-3, -2, 2, 3}, {-1, 1}}
sage: P.e(5/2)
P{{-4, 4}, {-3}, {-2, 2}, {-1, 1}, {3}}
sage: P.s(2)
P{{-4, 4}, {-3, 2}, {-2, 3}, {-1, 1}}
```

```
>>> from sage.all import *
>>> P = PartitionAlgebra(Integer(4), Integer(0))
>>> P.a(Integer(2))
P{{-4, 4}, {-3, -2}, {-1, 1}, {2, 3}}
>>> P.e(Integer(3)/Integer(2))
P{{-4, 4}, {-3, 3}, {-2}, {-1, 1}, {2}}
>>> P.e(Integer(2))
P{{-4, 4}, {-3, -2, 2, 3}, {-1, 1}}
>>> P.e(Integer(5)/Integer(2))
P{{-4, 4}, {-3}, {-2, 2}, {-1, 1}, {3}}
>>> P.s(Integer(2))
P{{-4, 4}, {-3, 2}, {-2, 3}, {-1, 1}}
```

An excellent reference for partition algebras and their various subalgebras (Brauer algebra, Temperley–Lieb algebra, etc) is the paper [HR2005].

INPUT:

- k – rank of the algebra
- q – the deformation parameter q
- `base_ring` – (default: None) a ring containing q ; if None, then Sage automatically chooses the parent of q
- `prefix` – (default: 'P') a label for the basis elements

EXAMPLES:

The following shorthand simultaneously defines the univariate polynomial ring over the rationals as well as the variable x :

```
sage: R.<x> = PolynomialRing(QQ)
sage: R
Univariate Polynomial Ring in x over Rational Field
sage: x
```

(continues on next page)

(continued from previous page)

```
x
sage: x.parent() is R
True
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ, names=('x',)); (x,) = R._first_ngens(1)
>>> R
Univariate Polynomial Ring in x over Rational Field
>>> x
x
>>> x.parent() is R
True
```

We now define the partition algebra of rank 2 with parameter x over $\mathbb{Z}$ in the usual (diagram) basis:

```
sage: R.<x> = ZZ[]
sage: A2 = PartitionAlgebra(2, x, R)
sage: A2
Partition Algebra of rank 2 with parameter x
over Univariate Polynomial Ring in x over Integer Ring
sage: A2.basis().keys()
Partition diagrams of order 2
sage: A2.basis().keys() ([-2, 1, 2], [-1])
[[-2, 1, 2], [-1]]
sage: A2.basis().list()
[P{{-2, -1, 1, 2}}, P{{-2, 1, 2}, {-1}},
 P{{-2}, {-1, 1, 2}}, P{{-2, -1}, {1, 2}},
 P{{-2}, {-1}, {1, 2}}, P{{-2, -1, 1}, {2}},
 P{{-2, 1}, {-1, 2}}, P{{-2, 1}, {-1}, {2}},
 P{{-2, 2}, {-1, 1}}, P{{-2, -1, 2}, {1}},
 P{{-2, 2}, {-1}, {1}}, P{{-2}, {-1, 1}, {2}},
 P{{-2}, {-1, 2}, {1}}, P{{-2, -1}, {1}, {2}},
 P{{-2}, {-1}, {1}, {2}}]
sage: E = A2([[1, 2], [-2, -1]]); E
P{{-2, -1}, {1, 2}}
sage: E in A2.basis().list()
True
sage: E^2
x*P{{-2, -1}, {1, 2}}
sage: E^5
x^4*P{{-2, -1}, {1, 2}}
sage: (A2([[2, -2], [-1, 1]]) - 2*A2([[1, 2], [-1, -2]]))^2
(4*x-4)*P{{-2, -1}, {1, 2}} + P{{-2, 2}, {-1, 1}}
```

```
>>> from sage.all import *
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> A2 = PartitionAlgebra(Integer(2), x, R)
>>> A2
Partition Algebra of rank 2 with parameter x
over Univariate Polynomial Ring in x over Integer Ring
>>> A2.basis().keys()
Partition diagrams of order 2
```

(continues on next page)

(continued from previous page)

```

>>> A2.basis().keys() ([[Integer(2), Integer(1), Integer(2)], [-Integer(1)]]
{-2, 1, 2}, {-1})
>>> A2.basis().list()
[P{{-2, -1, 1, 2}}, P{{-2, 1, 2}, {-1}},
 P{{-2}, {-1, 1, 2}}, P{{-2, -1}, {1, 2}},
 P{{-2}, {-1}, {1, 2}}, P{{-2, -1, 1}, {2}},
 P{{-2, 1}, {-1, 2}}, P{{-2, 1}, {-1}, {2}},
 P{{-2, 2}, {-1, 1}}, P{{-2, -1, 2}, {1}},
 P{{-2, 2}, {-1}, {1}}, P{{-2}, {-1, 1}, {2}},
 P{{-2}, {-1, 2}, {1}}, P{{-2, -1}, {1}, {2}},
 P{{-2}, {-1}, {1}, {2}}]
>>> E = A2([[Integer(1), Integer(2)], [-Integer(2), -Integer(1)]]); E
P{{-2, -1}, {1, 2}}
>>> E in A2.basis().list()
True
>>> E**Integer(2)
x*P{{-2, -1}, {1, 2}}
>>> E**Integer(5)
x^4*P{{-2, -1}, {1, 2}}
>>> (A2([[Integer(2), -Integer(2)], [-Integer(1), Integer(1)]] -
Integer(2)*A2([[Integer(1), Integer(2)], [-Integer(1), -Integer(2)]]))**Integer(2)
(4*x-4)*P{{-2, -1}, {1, 2}} + P{{-2, 2}, {-1, 1}}

```

Next, we construct an element:

```

sage: a2 = A2.an_element(); a2
3*P{{-2}, {-1, 1, 2}} + 2*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}

```

```

>>> from sage.all import *
>>> a2 = A2.an_element(); a2
3*P{{-2}, {-1, 1, 2}} + 2*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}

```

There is a natural embedding into partition algebras on more elements, by adding identity strands:

```

sage: A4 = PartitionAlgebra(4, x, R)
sage: A4(a2)
3*P{{-4, 4}, {-3, 3}, {-2}, {-1, 1, 2}}
+ 2*P{{-4, 4}, {-3, 3}, {-2, -1, 1, 2}}
+ 2*P{{-4, 4}, {-3, 3}, {-2, 1, 2}, {-1}}

```

```

>>> from sage.all import *
>>> A4 = PartitionAlgebra(Integer(4), x, R)
>>> A4(a2)
3*P{{-4, 4}, {-3, 3}, {-2}, {-1, 1, 2}}
+ 2*P{{-4, 4}, {-3, 3}, {-2, -1, 1, 2}}
+ 2*P{{-4, 4}, {-3, 3}, {-2, 1, 2}, {-1}}

```

Thus, the empty partition corresponds to the identity:

```

sage: A4([])
P{{-4, 4}, {-3, 3}, {-2, 2}, {-1, 1}}
sage: A4(5)

```

(continues on next page)

(continued from previous page)

```
5*P{{-4, 4}, {-3, 3}, {-2, 2}, {-1, 1}}
```

```
>>> from sage.all import *
>>> A4([])
P{{-4, 4}, {-3, 3}, {-2, 2}, {-1, 1}}
>>> A4(Integer(5))
5*P{{-4, 4}, {-3, 3}, {-2, 2}, {-1, 1}}
```

The group algebra of the symmetric group is a subalgebra:

```
sage: S3 = SymmetricGroupAlgebra(ZZ, 3)
sage: s3 = S3.an_element(); s3
[1, 2, 3] + 2*[1, 3, 2] + 3*[2, 1, 3] + [3, 1, 2]
sage: A4(s3)
P{{-4, 4}, {-3, 1}, {-2, 3}, {-1, 2}}
+ 2*P{{-4, 4}, {-3, 2}, {-2, 3}, {-1, 1}}
+ 3*P{{-4, 4}, {-3, 3}, {-2, 1}, {-1, 2}}
+ P{{-4, 4}, {-3, 3}, {-2, 2}, {-1, 1}}
sage: A4([2,1])
P{{-4, 4}, {-3, 3}, {-2, 1}, {-1, 2}}
```

```
>>> from sage.all import *
>>> S3 = SymmetricGroupAlgebra(ZZ, Integer(3))
>>> s3 = S3.an_element(); s3
[1, 2, 3] + 2*[1, 3, 2] + 3*[2, 1, 3] + [3, 1, 2]
>>> A4(s3)
P{{-4, 4}, {-3, 1}, {-2, 3}, {-1, 2}}
+ 2*P{{-4, 4}, {-3, 2}, {-2, 3}, {-1, 1}}
+ 3*P{{-4, 4}, {-3, 3}, {-2, 1}, {-1, 2}}
+ P{{-4, 4}, {-3, 3}, {-2, 2}, {-1, 1}}
>>> A4([Integer(2), Integer(1)])
P{{-4, 4}, {-3, 3}, {-2, 1}, {-1, 2}}
```

Be careful not to confuse the embedding of the group algebra of the symmetric group with the embedding of partial set partitions. The latter are embedded by adding the parts $\{i, -i\}$ if possible, and singletons sets for the remaining parts:

```
sage: A4([2,1])
P{{-4, 4}, {-3, 3}, {-2}, {-1}, {1, 2}}
sage: A4([-1,3], [-2,-3,1])
P{{-4, 4}, {-3, -2, 1}, {-1, 3}, {2}}
```

```
>>> from sage.all import *
>>> A4([Integer(2), Integer(1)])
P{{-4, 4}, {-3, 3}, {-2}, {-1}, {1, 2}}
>>> A4([-Integer(1), Integer(3)], [-Integer(2), -Integer(3), Integer(1)])
P{{-4, 4}, {-3, -2, 1}, {-1, 3}, {2}}
```

Another subalgebra is the Brauer algebra, which has perfect matchings as basis elements. The group algebra of the symmetric group is in fact a subalgebra of the Brauer algebra:

```
sage: B3 = BrauerAlgebra(3, x, R)
sage: b3 = B3(s3); b3
B{{-3, 1}, {-2, 3}, {-1, 2}} + 2*B{{-3, 2}, {-2, 3}, {-1, 1}}
+ 3*B{{-3, 3}, {-2, 1}, {-1, 2}} + B{{-3, 3}, {-2, 2}, {-1, 1}}
```

```
>>> from sage.all import *
>>> B3 = BrauerAlgebra(Integer(3), x, R)
>>> b3 = B3(s3); b3
B{{-3, 1}, {-2, 3}, {-1, 2}} + 2*B{{-3, 2}, {-2, 3}, {-1, 1}}
+ 3*B{{-3, 3}, {-2, 1}, {-1, 2}} + B{{-3, 3}, {-2, 2}, {-1, 1}}
```

An important basis of the partition algebra is the *orbit basis*:

```
sage: O2 = A2.orbit_basis()
sage: o2 = O2([[1,2],[1,-2]]) + O2([[1,2,-1,-2]]); o2
O{{-2, -1}, {1, 2}} + O{{-2, -1, 1, 2}}
```

```
>>> from sage.all import *
>>> O2 = A2.orbit_basis()
>>> o2 = O2([[Integer(1),Integer(2)],[-Integer(1),-Integer(2)]]) +
O2([[Integer(1),Integer(2),-Integer(1),-Integer(2)]]); o2
O{{-2, -1}, {1, 2}} + O{{-2, -1, 1, 2}}
```

The diagram basis element corresponds to the sum of all orbit basis elements indexed by coarser set partitions:

```
sage: A2(o2)
P{{-2, -1}, {1, 2}}
```

```
>>> from sage.all import *
>>> A2(o2)
P{{-2, -1}, {1, 2}}
```

We can convert back from the orbit basis to the diagram basis:

```
sage: o2 = O2.an_element(); o2
3*O{{-2}, {-1, 1, 2}} + 2*O{{-2, -1, 1, 2}} + 2*O{{-2, 1, 2}, {-1}}
sage: A2(o2)
3*P{{-2}, {-1, 1, 2}} - 3*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
```

```
>>> from sage.all import *
>>> o2 = O2.an_element(); o2
3*O{{-2}, {-1, 1, 2}} + 2*O{{-2, -1, 1, 2}} + 2*O{{-2, 1, 2}, {-1}}
>>> A2(o2)
3*P{{-2}, {-1, 1, 2}} - 3*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
```

One can work with partition algebras using a symbol for the parameter, leaving the base ring unspecified. This implies that the underlying base ring is Sage's symbolic ring.

```
sage: # needs sage.symbolic
sage: q = var('q')
sage: PA = PartitionAlgebra(2, q); PA
Partition Algebra of rank 2 with parameter q over Symbolic Ring
```

(continues on next page)

(continued from previous page)

```

sage: PA([[1,2],[-2,-1]])^2 == q*PA([[1,2],[-2,-1]])
True
sage: ((PA([[2,-2],[1,-1]]) - 2*PA([[2,-2],[1,-1]]))^2
....: == (4*q-4)*PA([[1,2],[-2,-1]]) + PA([[2,-2],[1,-1]])
True

```

```

>>> from sage.all import *
>>> # needs sage.symbolic
>>> q = var('q')
>>> PA = PartitionAlgebra(Integer(2), q); PA
Partition Algebra of rank 2 with parameter q over Symbolic Ring
>>> PA([[Integer(1),Integer(2)],[-Integer(2),-Integer(1)]])*Integer(2) ==
↳ q*PA([[Integer(1),Integer(2)],[-Integer(2),-Integer(1)]])
True
>>> ((PA([[Integer(2),-Integer(2)],[Integer(1),-Integer(1)]]) -
↳ Integer(2)*PA([[Integer(2),-Integer(2)],[Integer(1),-Integer(1)]])
↳ Integer(2)]))*Integer(2)
... == (Integer(4)*q-Integer(4))*PA([[Integer(1),Integer(2)],[-Integer(2),-
↳ Integer(1)]]) + PA([[Integer(2),-Integer(2)],[Integer(1),-Integer(1)]])
True

```

The identity element of the partition algebra is the set partition $\{\{1, -1\}, \{2, -2\}, \dots, \{k, -k\}\}$:

```

sage: # needs sage.symbolic
sage: P = PA.basis().list()
sage: PA.one()
P{{-2, 2}, {-1, 1}}
sage: PA.one() * P[7] == P[7]
True
sage: P[7] * PA.one() == P[7]
True

```

```

>>> from sage.all import *
>>> # needs sage.symbolic
>>> P = PA.basis().list()
>>> PA.one()
P{{-2, 2}, {-1, 1}}
>>> PA.one() * P[Integer(7)] == P[Integer(7)]
True
>>> P[Integer(7)] * PA.one() == P[Integer(7)]
True

```

We now give some further examples of the use of the other arguments. One may wish to “specialize” the parameter to a chosen element of the base ring:

```

sage: R.<q> = RR[]
sage: PA = PartitionAlgebra(2, q, R, prefix='B')
sage: PA
Partition Algebra of rank 2 with parameter q over
Univariate Polynomial Ring in q over Real Field with 53 bits of precision
sage: PA([[1,2],[-1,-2]])
1.000000000000000*B{{-2, -1}, {1, 2}}

```

(continues on next page)

(continued from previous page)

```

sage: PA = PartitionAlgebra(2, 5, base_ring=ZZ, prefix='B')
sage: PA
Partition Algebra of rank 2 with parameter 5 over Integer Ring
sage: ((PA([[2, -2], [1, -1]]) - 2*PA([[-2, -1], [1, 2]]))^2
....:  == 16*PA([[-2, -1], [1, 2]]) + PA([[2, -2], [1, -1]])
True

```

```

>>> from sage.all import *
>>> R = RR['q']; (q,) = R._first_ngens(1)
>>> PA = PartitionAlgebra(Integer(2), q, R, prefix='B')
>>> PA
Partition Algebra of rank 2 with parameter q over
Univariate Polynomial Ring in q over Real Field with 53 bits of precision
>>> PA([[Integer(1), Integer(2)], [-Integer(1), -Integer(2)]])
1.000000000000000*B{{-2, -1}, {1, 2}}
>>> PA = PartitionAlgebra(Integer(2), Integer(5), base_ring=ZZ, prefix='B')
>>> PA
Partition Algebra of rank 2 with parameter 5 over Integer Ring
>>> ((PA([[Integer(2), -Integer(2)], [Integer(1), -Integer(1)]])) -
↪ Integer(2)*PA([[-Integer(2), -Integer(1)], [Integer(1),
↪ Integer(2)]])**Integer(2)
... == Integer(16)*PA([[-Integer(2), -Integer(1)], [Integer(1), Integer(2)]])) +
↪ PA([[Integer(2), -Integer(2)], [Integer(1), -Integer(1)]])
True

```

Symmetric group algebra elements and elements from other subalgebras of the partition algebra (e.g., BrauerAlgebra and TemperleyLiebAlgebra) can also be coerced into the partition algebra:

```

sage: # needs sage.symbolic
sage: S = SymmetricGroupAlgebra(SR, 2)
sage: B = BrauerAlgebra(2, x, SR)
sage: A = PartitionAlgebra(2, x, SR)
sage: S([2, 1]) * A([[1, -1], [2, -2]])
P{{-2, 1}, {-1, 2}}
sage: B([[-1, -2], [2, 1]]) * A([[1], [-1], [2, -2]])
P{{-2}, {-1}, {1, 2}}
sage: A([[1], [-1], [2, -2]]) * B([[-1, -2], [2, 1]])
P{{-2, -1}, {1}, {2}}

```

```

>>> from sage.all import *
>>> # needs sage.symbolic
>>> S = SymmetricGroupAlgebra(SR, Integer(2))
>>> B = BrauerAlgebra(Integer(2), x, SR)
>>> A = PartitionAlgebra(Integer(2), x, SR)
>>> S([Integer(2), Integer(1)]) * A([Integer(1), -Integer(1)], [Integer(2), -
↪ Integer(2)])
P{{-2, 1}, {-1, 2}}
>>> B([[-Integer(1), -Integer(2)], [Integer(2), Integer(1)]] * A([Integer(1)], [-
↪ Integer(1)], [Integer(2), -Integer(2)])
P{{-2}, {-1}, {1, 2}}
>>> A([Integer(1)], [-Integer(1)], [Integer(2), -Integer(2)]) * B([[-Integer(1), -
↪ Integer(2)], [Integer(2), Integer(1)]])

```

(continues on next page)

(continued from previous page)

```
P{{-2, -1}, {1}, {2}}
```

The same is true if the elements come from a subalgebra of a partition algebra of smaller order, or if they are defined over a different base ring:

```
sage: R = FractionField(ZZ['q']); q = R.gen()
sage: S = SymmetricGroupAlgebra(ZZ, 2)
sage: B = BrauerAlgebra(2, q, ZZ[q])
sage: A = PartitionAlgebra(3, q, R)
sage: S([2, 1]) * A([[1, -1], [2, -3], [3, -2]])
P{{-3, 1}, {-2, 3}, {-1, 2}}
sage: A(B([[-1, -2], [2, 1]]))
P{{-3, 3}, {-2, -1}, {1, 2}}
```

```
>>> from sage.all import *
>>> R = FractionField(ZZ['q']); q = R.gen()
>>> S = SymmetricGroupAlgebra(ZZ, Integer(2))
>>> B = BrauerAlgebra(Integer(2), q, ZZ[q])
>>> A = PartitionAlgebra(Integer(3), q, R)
>>> S([Integer(2), Integer(1)]) * A([Integer(1), -Integer(1)], [Integer(2), -
↪ Integer(3)], [Integer(3), -Integer(2)])
P{{-3, 1}, {-2, 3}, {-1, 2}}
>>> A(B([[-Integer(1), -Integer(2)], [Integer(2), Integer(1)]]))
P{{-3, 3}, {-2, -1}, {1, 2}}
```

class Element

Bases: *Element*

dual()

Return the dual of self.

The dual of an element in the partition algebra is formed by taking the dual of each diagram in the support.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: P = PartitionAlgebra(2, x, R)
sage: elt = P.an_element(); elt
3*P{{-2}, {-1, 1, 2}} + 2*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
sage: elt.dual()
3*P{{-2, -1, 1}, {2}} + 2*P{{-2, -1, 1, 2}} + 2*P{{-2, -1, 2}, {1}}
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> P = PartitionAlgebra(Integer(2), x, R)
>>> elt = P.an_element(); elt
3*P{{-2}, {-1, 1, 2}} + 2*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
>>> elt.dual()
3*P{{-2, -1, 1}, {2}} + 2*P{{-2, -1, 1, 2}} + 2*P{{-2, -1, 2}, {1}}
```

to_orbit_basis()

Return self in the orbit basis of the associated partition algebra.

EXAMPLES:

```

sage: R.<x> = QQ[]
sage: P = PartitionAlgebra(2, x, R)
sage: pp = P.an_element();
sage: pp.to_orbit_basis()
3*O{{-2}, {-1, 1, 2}} + 7*O{{-2, -1, 1, 2}} + 2*O{{-2, 1, 2}, {-1}}
sage: pp = (3*P([[ -2], [-1, 1, 2]]) + 2*P([[ -2, -1, 1, 2]])
....:      + 2*P([[ -2, 1, 2], [-1]])); pp
3*P{{-2}, {-1, 1, 2}} + 2*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
sage: pp.to_orbit_basis()
3*O{{-2}, {-1, 1, 2}} + 7*O{{-2, -1, 1, 2}} + 2*O{{-2, 1, 2}, {-1}}

```

```

>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> P = PartitionAlgebra(Integer(2), x, R)
>>> pp = P.an_element();
>>> pp.to_orbit_basis()
3*O{{-2}, {-1, 1, 2}} + 7*O{{-2, -1, 1, 2}} + 2*O{{-2, 1, 2}, {-1}}
>>> pp = (Integer(3)*P([[-Integer(2)], [-Integer(1), Integer(1),
↪Integer(2)]])) + Integer(2)*P([[-Integer(2), -Integer(1), Integer(1),
↪Integer(2)]]))
...      + Integer(2)*P([[-Integer(2), Integer(1), Integer(2)], [-
↪Integer(1)]])); pp
3*P{{-2}, {-1, 1, 2}} + 2*P{{-2, -1, 1, 2}} + 2*P{{-2, 1, 2}, {-1}}
>>> pp.to_orbit_basis()
3*O{{-2}, {-1, 1, 2}} + 7*O{{-2, -1, 1, 2}} + 2*O{{-2, 1, 2}, {-1}}

```

L(*i*)Return the *i*-th Jucys-Murphy element L_i from [Eny2012].

INPUT:

- *i* – half integer between $1/2$ and k

ALGORITHM:

We use the recursive definition for L_{2i} given in [Cre2020]. See also [Eny2012] and [Eny2013].**Note** $L_{1/2}$ and L_1 differs from [HR2005].

EXAMPLES:

```

sage: R.<n> = QQ[]
sage: P3 = PartitionAlgebra(3, n)
sage: P3.jucys_murphy_element(1/2)
0
sage: P3.jucys_murphy_element(1)
P{{-3, 3}, {-2, 2}, {-1}, {1}}
sage: P3.jucys_murphy_element(2)
P{{-3, 3}, {-2}, {-1, 1}, {2}} - P{{-3, 3}, {-2}, {-1, 1, 2}}
+ P{{-3, 3}, {-2, -1}, {1, 2}} - P{{-3, 3}, {-2, -1, 1}, {2}}

```

(continues on next page)

(continued from previous page)

```

+ P{{-3, 3}, {-2, 1}, {-1, 2}}
sage: P3.jucys_murphy_element(3/2)
n*P{{-3, 3}, {-2, -1, 1, 2}} - P{{-3, 3}, {-2, -1, 2}, {1}}
- P{{-3, 3}, {-2, 1, 2}, {-1}} + P{{-3, 3}, {-2, 2}, {-1, 1}}
sage: P3.L(3/2) * P3.L(2) == P3.L(2) * P3.L(3/2)
True

```

```

>>> from sage.all import *
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P3 = PartitionAlgebra(Integer(3), n)
>>> P3.jucys_murphy_element(Integer(1)/Integer(2))
0
>>> P3.jucys_murphy_element(Integer(1))
P{{-3, 3}, {-2, 2}, {-1}, {1}}
>>> P3.jucys_murphy_element(Integer(2))
P{{-3, 3}, {-2}, {-1, 1}, {2}} - P{{-3, 3}, {-2}, {-1, 1, 2}}
+ P{{-3, 3}, {-2, -1}, {1, 2}} - P{{-3, 3}, {-2, -1, 1}, {2}}
+ P{{-3, 3}, {-2, 1}, {-1, 2}}
>>> P3.jucys_murphy_element(Integer(3)/Integer(2))
n*P{{-3, 3}, {-2, -1, 1, 2}} - P{{-3, 3}, {-2, -1, 2}, {1}}
- P{{-3, 3}, {-2, 1, 2}, {-1}} + P{{-3, 3}, {-2, 2}, {-1, 1}}
>>> P3.L(Integer(3)/Integer(2)) * P3.L(Integer(2)) == P3.L(Integer(2)) * P3.
↪L(Integer(3)/Integer(2))
True

```

We test the relations in Lemma 2.2.3(2) in [Cre2020] (v1):

```

sage: k = 4
sage: R.<n> = QQ[]
sage: P = PartitionAlgebra(k, n)
sage: L = [P.L(i/2) for i in range(1, 2*k+1)]
sage: all(x.dual() == x for x in L)
True
sage: all(x * y == y * x for x, y in Subsets(L, 2)) # long time
True
sage: Lsum = sum(L)
sage: gens = [P.s(i) for i in range(1, k)]
sage: gens += [P.e(i/2) for i in range(1, 2*k)]
sage: all(x * Lsum == Lsum * x for x in gens)
True

```

```

>>> from sage.all import *
>>> k = Integer(4)
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P = PartitionAlgebra(k, n)
>>> L = [P.L(i/Integer(2)) for i in range(Integer(1), Integer(2)*k+Integer(1))]
>>> all(x.dual() == x for x in L)
True
>>> all(x * y == y * x for x, y in Subsets(L, Integer(2))) # long time
True
>>> Lsum = sum(L)
>>> gens = [P.s(i) for i in range(Integer(1), k)]

```

(continues on next page)

(continued from previous page)

```
>>> gens += [P.e(i/Integer(2)) for i in range(Integer(1), Integer(2)*k)]
>>> all(x * Lsum == Lsum * x for x in gens)
True
```

Also the relations in Lemma 2.2.3(3) in [Cre2020] (v1):

```
sage: all(P.e((2*i+1)/2) * P.sigma(2*i/2) * P.e((2*i+1)/2)
.....:      == (n - P.L((2*i-1)/2)) * P.e((2*i+1)/2) for i in range(1,k))
True
sage: all(P.e(i/2) * (P.L(i/2) + P.L((i+1)/2))
.....:      == (P.L(i/2) + P.L((i+1)/2)) * P.e(i/2)
.....:      == n * P.e(i/2) for i in range(1,2*k))
True
sage: all(P.sigma(2*i/2) * P.e((2*i-1)/2) * P.e(2*i/2)
.....:      == P.L(2*i/2) * P.e(2*i/2) for i in range(1,k))
True
sage: all(P.e(2*i/2) * P.e((2*i-1)/2) * P.sigma(2*i/2)
.....:      == P.e(2*i/2) * P.L(2*i/2) for i in range(1,k))
True
sage: all(P.sigma((2*i+1)/2) * P.e((2*i+1)/2) * P.e(2*i/2)
.....:      == P.L(2*i/2) * P.e(2*i/2) for i in range(1,k))
True
sage: all(P.e(2*i/2) * P.e((2*i+1)/2) * P.sigma((2*i+1)/2)
.....:      == P.e(2*i/2) * P.L(2*i/2) for i in range(1,k))
True
```

```
>>> from sage.all import *
>>> all(P.e((Integer(2)*i+Integer(1))/Integer(2)) * P.sigma(Integer(2)*i/
↳Integer(2)) * P.e((Integer(2)*i+Integer(1))/Integer(2))
...      == (n - P.L((Integer(2)*i-Integer(1))/Integer(2))) * P.
↳e((Integer(2)*i+Integer(1))/Integer(2)) for i in range(Integer(1),k))
True
>>> all(P.e(i/Integer(2)) * (P.L(i/Integer(2)) + P.L((i+Integer(1))/
↳Integer(2)))
...      == (P.L(i/Integer(2)) + P.L((i+Integer(1))/Integer(2))) * P.e(i/
↳Integer(2))
...      == n * P.e(i/Integer(2)) for i in range(Integer(1), Integer(2)*k))
True
>>> all(P.sigma(Integer(2)*i/Integer(2)) * P.e((Integer(2)*i-Integer(1))/
↳Integer(2)) * P.e(Integer(2)*i/Integer(2))
...      == P.L(Integer(2)*i/Integer(2)) * P.e(Integer(2)*i/Integer(2)) for i
↳in range(Integer(1),k))
True
>>> all(P.e(Integer(2)*i/Integer(2)) * P.e((Integer(2)*i-Integer(1))/
↳Integer(2)) * P.sigma(Integer(2)*i/Integer(2))
...      == P.e(Integer(2)*i/Integer(2)) * P.L(Integer(2)*i/Integer(2)) for i
↳in range(Integer(1),k))
True
>>> all(P.sigma((Integer(2)*i+Integer(1))/Integer(2)) * P.
↳e((Integer(2)*i+Integer(1))/Integer(2)) * P.e(Integer(2)*i/Integer(2))
...      == P.L(Integer(2)*i/Integer(2)) * P.e(Integer(2)*i/Integer(2)) for i
↳in range(Integer(1),k))
```

(continues on next page)

(continued from previous page)

```

True
>>> all(P.e(Integer(2)*i/Integer(2)) * P.e((Integer(2)*i+Integer(1))/
↪Integer(2)) * P.sigma((Integer(2)*i+Integer(1))/Integer(2))
...      == P.e(Integer(2)*i/Integer(2)) * P.L(Integer(2)*i/Integer(2)) for i in
↪in range(Integer(1),k))
True

```

The same tests for a half integer partition algebra:

```

sage: k = 7/2
sage: R.<n> = QQ[]
sage: P = PartitionAlgebra(k, n)
sage: L = [P.L(i/2) for i in range(1,2*k+1)]
sage: all(x.dual() == x for x in L)
True
sage: all(x * y == y * x for x, y in Subsets(L, 2)) # long time
True
sage: Lsum = sum(L)
sage: gens = [P.s(i) for i in range(1,k-1/2)]
sage: gens += [P.e(i/2) for i in range(1,2*k)]
sage: all(x * Lsum == Lsum * x for x in gens)
True
sage: all(P.e((2*i+1)/2) * P.sigma(2*i/2) * P.e((2*i+1)/2)
....:      == (n - P.L((2*i-1)/2)) * P.e((2*i+1)/2) for i in range(1,floor(k)))
True
sage: all(P.e(i/2) * (P.L(i/2) + P.L((i+1)/2))
....:      == (P.L(i/2) + P.L((i+1)/2)) * P.e(i/2)
....:      == n * P.e(i/2) for i in range(1,2*k))
True
sage: all(P.sigma(2*i/2) * P.e((2*i-1)/2) * P.e(2*i/2)
....:      == P.L(2*i/2) * P.e(2*i/2) for i in range(1,ceil(k)))
True
sage: all(P.e(2*i/2) * P.e((2*i-1)/2) * P.sigma(2*i/2)
....:      == P.e(2*i/2) * P.L(2*i/2) for i in range(1,ceil(k)))
True
sage: all(P.sigma((2*i+1)/2) * P.e((2*i+1)/2) * P.e(2*i/2)
....:      == P.L(2*i/2) * P.e(2*i/2) for i in range(1,floor(k)))
True
sage: all(P.e(2*i/2) * P.e((2*i+1)/2) * P.sigma((2*i+1)/2)
....:      == P.e(2*i/2) * P.L(2*i/2) for i in range(1,floor(k)))
True

```

```

>>> from sage.all import *
>>> k = Integer(7)/Integer(2)
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P = PartitionAlgebra(k, n)
>>> L = [P.L(i/Integer(2)) for i in range(Integer(1),Integer(2)*k+Integer(1))]
>>> all(x.dual() == x for x in L)
True
>>> all(x * y == y * x for x, y in Subsets(L, Integer(2))) # long time
True
>>> Lsum = sum(L)

```

(continues on next page)

(continued from previous page)

```

>>> gens = [P.s(i) for i in range(Integer(1),k-Integer(1)/Integer(2))]
>>> gens += [P.e(i/Integer(2)) for i in range(Integer(1),Integer(2)*k)]
>>> all(x * Lsum == Lsum * x for x in gens)
True
>>> all(P.e((Integer(2)*i+Integer(1))/Integer(2)) * P.sigma(Integer(2)*i/
↳Integer(2)) * P.e((Integer(2)*i+Integer(1))/Integer(2))
...      == (n - P.L((Integer(2)*i-Integer(1))/Integer(2))) * P.
↳e((Integer(2)*i+Integer(1))/Integer(2)) for i in range(Integer(1),floor(k)))
True
>>> all(P.e(i/Integer(2)) * (P.L(i/Integer(2)) + P.L((i+Integer(1))/
↳Integer(2)))
...      == (P.L(i/Integer(2)) + P.L((i+Integer(1))/Integer(2))) * P.e(i/
↳Integer(2))
...      == n * P.e(i/Integer(2)) for i in range(Integer(1),Integer(2)*k))
True
>>> all(P.sigma(Integer(2)*i/Integer(2)) * P.e((Integer(2)*i-Integer(1))/
↳Integer(2)) * P.e(Integer(2)*i/Integer(2))
...      == P.L(Integer(2)*i/Integer(2)) * P.e(Integer(2)*i/Integer(2)) for i
↳in range(Integer(1),ceil(k)))
True
>>> all(P.e(Integer(2)*i/Integer(2)) * P.e((Integer(2)*i-Integer(1))/
↳Integer(2)) * P.sigma(Integer(2)*i/Integer(2))
...      == P.e(Integer(2)*i/Integer(2)) * P.L(Integer(2)*i/Integer(2)) for i
↳in range(Integer(1),ceil(k)))
True
>>> all(P.sigma((Integer(2)*i+Integer(1))/Integer(2)) * P.
↳e((Integer(2)*i+Integer(1))/Integer(2)) * P.e(Integer(2)*i/Integer(2))
...      == P.L(Integer(2)*i/Integer(2)) * P.e(Integer(2)*i/Integer(2)) for i
↳in range(Integer(1),floor(k)))
True
>>> all(P.e(Integer(2)*i/Integer(2)) * P.e((Integer(2)*i+Integer(1))/
↳Integer(2)) * P.sigma((Integer(2)*i+Integer(1))/Integer(2))
...      == P.e(Integer(2)*i/Integer(2)) * P.L(Integer(2)*i/Integer(2)) for i
↳in range(Integer(1),floor(k)))
True

```

a(i)

Return the element a_i in self.

The element a_i is the cap and cup at $(i, i + 1)$, so it contains the blocks $\{i, i + 1\}$, $\{-i, -i - 1\}$. Other blocks are of the form $\{-j, j\}$.

INPUT:

- i – integer between 1 and $k - 1$

EXAMPLES:

```

sage: R.<n> = QQ[]
sage: P3 = PartitionAlgebra(3, n)
sage: P3.a(1)
P{{-3, 3}, {-2, -1}, {1, 2}}
sage: P3.a(2)
P{{-3, -2}, {-1, 1}, {2, 3}}

```

(continues on next page)

(continued from previous page)

```

sage: P3 = PartitionAlgebra(5/2, n)
sage: P3.a(1)
P{{-3, 3}, {-2, -1}, {1, 2}}
sage: P3.a(2)
Traceback (most recent call last):
...
ValueError: i must be an integer between 1 and 1

```

```

>>> from sage.all import *
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P3 = PartitionAlgebra(Integer(3), n)
>>> P3.a(Integer(1))
P{{-3, 3}, {-2, -1}, {1, 2}}
>>> P3.a(Integer(2))
P{{-3, -2}, {-1, 1}, {2, 3}}

>>> P3 = PartitionAlgebra(Integer(5)/Integer(2), n)
>>> P3.a(Integer(1))
P{{-3, 3}, {-2, -1}, {1, 2}}
>>> P3.a(Integer(2))
Traceback (most recent call last):
...
ValueError: i must be an integer between 1 and 1

```

e(i)

Return the element e_i in self.

If $i = (2r + 1)/2$, then e_i contains the blocks $\{r + 1\}$ and $\{-r - 1\}$. If $i \in \mathbf{Z}$, then e_i contains the block $\{-i, -i - 1, i, i + 1\}$. Other blocks are of the form $\{-j, j\}$.

INPUT:

- i - half integer between $1/2$ and $k - 1/2$

EXAMPLES:

```

sage: R.<n> = QQ[]
sage: P3 = PartitionAlgebra(3, n)
sage: P3.e(1)
P{{-3, 3}, {-2, -1, 1, 2}}
sage: P3.e(2)
P{{-3, -2, 2, 3}, {-1, 1}}
sage: P3.e(1/2)
P{{-3, 3}, {-2, 2}, {-1}, {1}}
sage: P3.e(5/2)
P{{-3}, {-2, 2}, {-1, 1}, {3}}
sage: P3.e(0)
Traceback (most recent call last):
...
ValueError: i must be an (half) integer between 1/2 and 5/2
sage: P3.e(3)
Traceback (most recent call last):
...

```

(continues on next page)

(continued from previous page)

```
ValueError: i must be an (half) integer between 1/2 and 5/2
```

```
sage: P2h = PartitionAlgebra(5/2,n)
sage: [P2h.e(k/2) for k in range(1,5)]
[P{{-3, 3}, {-2, 2}, {-1}, {1}},
 P{{-3, 3}, {-2, -1, 1, 2}},
 P{{-3, 3}, {-2}, {-1, 1}, {2}},
 P{{-3, -2, 2, 3}, {-1, 1}}]
```

```
>>> from sage.all import *
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P3 = PartitionAlgebra(Integer(3), n)
>>> P3.e(Integer(1))
P{{-3, 3}, {-2, -1, 1, 2}}
>>> P3.e(Integer(2))
P{{-3, -2, 2, 3}, {-1, 1}}
>>> P3.e(Integer(1)/Integer(2))
P{{-3, 3}, {-2, 2}, {-1}, {1}}
>>> P3.e(Integer(5)/Integer(2))
P{{-3}, {-2, 2}, {-1, 1}, {3}}
>>> P3.e(Integer(0))
Traceback (most recent call last):
...
ValueError: i must be an (half) integer between 1/2 and 5/2
>>> P3.e(Integer(3))
Traceback (most recent call last):
...
ValueError: i must be an (half) integer between 1/2 and 5/2

>>> P2h = PartitionAlgebra(Integer(5)/Integer(2),n)
>>> [P2h.e(k/Integer(2)) for k in range(Integer(1),Integer(5))]
[P{{-3, 3}, {-2, 2}, {-1}, {1}},
 P{{-3, 3}, {-2, -1, 1, 2}},
 P{{-3, 3}, {-2}, {-1, 1}, {2}},
 P{{-3, -2, 2, 3}, {-1, 1}}]
```

generator_a(i)

Return the element a_i in self.

The element a_i is the cap and cup at $(i, i + 1)$, so it contains the blocks $\{i, i + 1\}$, $\{-i, -i - 1\}$. Other blocks are of the form $\{-j, j\}$.

INPUT:

- i – integer between 1 and $k - 1$

EXAMPLES:

```
sage: R.<n> = QQ[]
sage: P3 = PartitionAlgebra(3, n)
sage: P3.a(1)
P{{-3, 3}, {-2, -1}, {1, 2}}
sage: P3.a(2)
P{{-3, -2}, {-1, 1}, {2, 3}}
```

(continues on next page)

(continued from previous page)

```

sage: P3 = PartitionAlgebra(5/2, n)
sage: P3.a(1)
P{{-3, 3}, {-2, -1}, {1, 2}}
sage: P3.a(2)
Traceback (most recent call last):
...
ValueError: i must be an integer between 1 and 1

```

```

>>> from sage.all import *
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P3 = PartitionAlgebra(Integer(3), n)
>>> P3.a(Integer(1))
P{{-3, 3}, {-2, -1}, {1, 2}}
>>> P3.a(Integer(2))
P{{-3, -2}, {-1, 1}, {2, 3}}

>>> P3 = PartitionAlgebra(Integer(5)/Integer(2), n)
>>> P3.a(Integer(1))
P{{-3, 3}, {-2, -1}, {1, 2}}
>>> P3.a(Integer(2))
Traceback (most recent call last):
...
ValueError: i must be an integer between 1 and 1

```

generator_e(i)

Return the element e_i in self.

If $i = (2r + 1)/2$, then e_i contains the blocks $\{r + 1\}$ and $\{-r - 1\}$. If $i \in \mathbf{Z}$, then e_i contains the block $\{-i, -i - 1, i, i + 1\}$. Other blocks are of the form $\{-j, j\}$.

INPUT:

- i - half integer between $1/2$ and $k - 1/2$

EXAMPLES:

```

sage: R.<n> = QQ[]
sage: P3 = PartitionAlgebra(3, n)
sage: P3.e(1)
P{{-3, 3}, {-2, -1, 1, 2}}
sage: P3.e(2)
P{{-3, -2, 2, 3}, {-1, 1}}
sage: P3.e(1/2)
P{{-3, 3}, {-2, 2}, {-1}, {1}}
sage: P3.e(5/2)
P{{-3}, {-2, 2}, {-1, 1}, {3}}
sage: P3.e(0)
Traceback (most recent call last):
...
ValueError: i must be an (half) integer between 1/2 and 5/2
sage: P3.e(3)
Traceback (most recent call last):
...

```

(continues on next page)

(continued from previous page)

```
ValueError: i must be an (half) integer between 1/2 and 5/2
```

```
sage: P2h = PartitionAlgebra(5/2,n)
sage: [P2h.e(k/2) for k in range(1,5)]
P{{-3, 3}, {-2, 2}, {-1}, {1}},
P{{-3, 3}, {-2, -1, 1, 2}},
P{{-3, 3}, {-2}, {-1, 1}, {2}},
P{{-3, -2, 2, 3}, {-1, 1}}
```

```
>>> from sage.all import *
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P3 = PartitionAlgebra(Integer(3), n)
>>> P3.e(Integer(1))
P{{-3, 3}, {-2, -1, 1, 2}}
>>> P3.e(Integer(2))
P{{-3, -2, 2, 3}, {-1, 1}}
>>> P3.e(Integer(1)/Integer(2))
P{{-3, 3}, {-2, 2}, {-1}, {1}}
>>> P3.e(Integer(5)/Integer(2))
P{{-3}, {-2, 2}, {-1, 1}, {3}}
>>> P3.e(Integer(0))
Traceback (most recent call last):
...
ValueError: i must be an (half) integer between 1/2 and 5/2
>>> P3.e(Integer(3))
Traceback (most recent call last):
...
ValueError: i must be an (half) integer between 1/2 and 5/2

>>> P2h = PartitionAlgebra(Integer(5)/Integer(2),n)
>>> [P2h.e(k/Integer(2)) for k in range(Integer(1),Integer(5))]
P{{-3, 3}, {-2, 2}, {-1}, {1}},
P{{-3, 3}, {-2, -1, 1, 2}},
P{{-3, 3}, {-2}, {-1, 1}, {2}},
P{{-3, -2, 2, 3}, {-1, 1}}
```

generator_s(i)

Return the i -th simple transposition s_i in self.

Borrowing the notation from the symmetric group, the i -th simple transposition s_i has blocks of the form $\{-i, i+1\}$, $\{-i-1, i\}$. Other blocks are of the form $\{-j, j\}$.

INPUT:

- i – integer between 1 and $k-1$

EXAMPLES:

```
sage: R.<n> = QQ[]
sage: P3 = PartitionAlgebra(3, n)
sage: P3.s(1)
P{{-3, 3}, {-2, 1}, {-1, 2}}
sage: P3.s(2)
P{{-3, 2}, {-2, 3}, {-1, 1}}
```

(continues on next page)

(continued from previous page)

```
sage: R.<n> = ZZ[]
sage: P2h = PartitionAlgebra(5/2,n)
sage: P2h.s(1)
P{{-3, 3}, {-2, 1}, {-1, 2}}
```

```
>>> from sage.all import *
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P3 = PartitionAlgebra(Integer(3), n)
>>> P3.s(Integer(1))
P{{-3, 3}, {-2, 1}, {-1, 2}}
>>> P3.s(Integer(2))
P{{-3, 2}, {-2, 3}, {-1, 1}}

>>> R = ZZ['n']; (n,) = R._first_ngens(1)
>>> P2h = PartitionAlgebra(Integer(5)/Integer(2),n)
>>> P2h.s(Integer(1))
P{{-3, 3}, {-2, 1}, {-1, 2}}
```

jucys_murphy_element(*i*)

Return the *i*-th Jucys-Murphy element L_i from [Eny2012].

INPUT:

- *i* – half integer between $1/2$ and k

ALGORITHM:

We use the recursive definition for L_{2i} given in [Cre2020]. See also [Eny2012] and [Eny2013].

Note

$L_{1/2}$ and L_1 differs from [HR2005].

EXAMPLES:

```
sage: R.<n> = QQ[]
sage: P3 = PartitionAlgebra(3, n)
sage: P3.jucys_murphy_element(1/2)
0
sage: P3.jucys_murphy_element(1)
P{{-3, 3}, {-2, 2}, {-1}, {1}}
sage: P3.jucys_murphy_element(2)
P{{-3, 3}, {-2}, {-1, 1}, {2}} - P{{-3, 3}, {-2}, {-1, 1, 2}}
+ P{{-3, 3}, {-2, -1}, {1, 2}} - P{{-3, 3}, {-2, -1, 1}, {2}}
+ P{{-3, 3}, {-2, 1}, {-1, 2}}
sage: P3.jucys_murphy_element(3/2)
n*P{{-3, 3}, {-2, -1, 1, 2}} - P{{-3, 3}, {-2, -1, 2}, {1}}
- P{{-3, 3}, {-2, 1, 2}, {-1}} + P{{-3, 3}, {-2, 2}, {-1, 1}}
sage: P3.L(3/2) * P3.L(2) == P3.L(2) * P3.L(3/2)
True
```

```

>>> from sage.all import *
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P3 = PartitionAlgebra(Integer(3), n)
>>> P3.jucys_murphy_element(Integer(1)/Integer(2))
0
>>> P3.jucys_murphy_element(Integer(1))
P{{-3, 3}, {-2, 2}, {-1}, {1}}
>>> P3.jucys_murphy_element(Integer(2))
P{{-3, 3}, {-2}, {-1, 1}, {2}} - P{{-3, 3}, {-2}, {-1, 1, 2}}
+ P{{-3, 3}, {-2, -1}, {1, 2}} - P{{-3, 3}, {-2, -1, 1}, {2}}
+ P{{-3, 3}, {-2, 1}, {-1, 2}}
>>> P3.jucys_murphy_element(Integer(3)/Integer(2))
n*P{{-3, 3}, {-2, -1, 1, 2}} - P{{-3, 3}, {-2, -1, 2}, {1}}
- P{{-3, 3}, {-2, 1, 2}, {-1}} + P{{-3, 3}, {-2, 2}, {-1, 1}}
>>> P3.L(Integer(3)/Integer(2)) * P3.L(Integer(2)) == P3.L(Integer(2)) * P3.
↳L(Integer(3)/Integer(2))
True

```

We test the relations in Lemma 2.2.3(2) in [Cre2020] (v1):

```

sage: k = 4
sage: R.<n> = QQ[]
sage: P = PartitionAlgebra(k, n)
sage: L = [P.L(i/2) for i in range(1, 2*k+1)]
sage: all(x.dual() == x for x in L)
True
sage: all(x * y == y * x for x, y in Subsets(L, 2)) # long time
True
sage: Lsum = sum(L)
sage: gens = [P.s(i) for i in range(1, k)]
sage: gens += [P.e(i/2) for i in range(1, 2*k)]
sage: all(x * Lsum == Lsum * x for x in gens)
True

```

```

>>> from sage.all import *
>>> k = Integer(4)
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P = PartitionAlgebra(k, n)
>>> L = [P.L(i/Integer(2)) for i in range(Integer(1), Integer(2)*k+Integer(1))]
>>> all(x.dual() == x for x in L)
True
>>> all(x * y == y * x for x, y in Subsets(L, Integer(2))) # long time
True
>>> Lsum = sum(L)
>>> gens = [P.s(i) for i in range(Integer(1), k)]
>>> gens += [P.e(i/Integer(2)) for i in range(Integer(1), Integer(2)*k)]
>>> all(x * Lsum == Lsum * x for x in gens)
True

```

Also the relations in Lemma 2.2.3(3) in [Cre2020] (v1):

```

sage: all(P.e((2*i+1)/2) * P.sigma(2*i/2) * P.e((2*i+1)/2)
.....:      == (n - P.L((2*i-1)/2)) * P.e((2*i+1)/2) for i in range(1, k))

```

(continues on next page)

(continued from previous page)

```

True
sage: all(P.e(i/2) * (P.L(i/2) + P.L((i+1)/2))
....:      == (P.L(i/2) + P.L((i+1)/2)) * P.e(i/2)
....:      == n * P.e(i/2) for i in range(1,2*k))
True
sage: all(P.sigma(2*i/2) * P.e((2*i-1)/2) * P.e(2*i/2)
....:      == P.L(2*i/2) * P.e(2*i/2) for i in range(1,k))
True
sage: all(P.e(2*i/2) * P.e((2*i-1)/2) * P.sigma(2*i/2)
....:      == P.e(2*i/2) * P.L(2*i/2) for i in range(1,k))
True
sage: all(P.sigma((2*i+1)/2) * P.e((2*i+1)/2) * P.e(2*i/2)
....:      == P.L(2*i/2) * P.e(2*i/2) for i in range(1,k))
True
sage: all(P.e(2*i/2) * P.e((2*i+1)/2) * P.sigma((2*i+1)/2)
....:      == P.e(2*i/2) * P.L(2*i/2) for i in range(1,k))
True

```

```

>>> from sage.all import *
>>> all(P.e((Integer(2)*i+Integer(1))/Integer(2)) * P.sigma(Integer(2)*i/
↪Integer(2)) * P.e((Integer(2)*i+Integer(1))/Integer(2))
...      == (n - P.L((Integer(2)*i-Integer(1))/Integer(2))) * P.
↪e((Integer(2)*i+Integer(1))/Integer(2)) for i in range(Integer(1),k))
True
>>> all(P.e(i/Integer(2)) * (P.L(i/Integer(2)) + P.L((i+Integer(1))/
↪Integer(2)))
...      == (P.L(i/Integer(2)) + P.L((i+Integer(1))/Integer(2))) * P.e(i/
↪Integer(2))
...      == n * P.e(i/Integer(2)) for i in range(Integer(1),Integer(2)*k))
True
>>> all(P.sigma(Integer(2)*i/Integer(2)) * P.e((Integer(2)*i-Integer(1))/
↪Integer(2)) * P.e(Integer(2)*i/Integer(2))
...      == P.L(Integer(2)*i/Integer(2)) * P.e(Integer(2)*i/Integer(2)) for i
↪in range(Integer(1),k))
True
>>> all(P.e(Integer(2)*i/Integer(2)) * P.e((Integer(2)*i-Integer(1))/
↪Integer(2)) * P.sigma(Integer(2)*i/Integer(2))
...      == P.e(Integer(2)*i/Integer(2)) * P.L(Integer(2)*i/Integer(2)) for i
↪in range(Integer(1),k))
True
>>> all(P.sigma((Integer(2)*i+Integer(1))/Integer(2)) * P.
↪e((Integer(2)*i+Integer(1))/Integer(2)) * P.e(Integer(2)*i/Integer(2))
...      == P.L(Integer(2)*i/Integer(2)) * P.e(Integer(2)*i/Integer(2)) for i
↪in range(Integer(1),k))
True
>>> all(P.e(Integer(2)*i/Integer(2)) * P.e((Integer(2)*i+Integer(1))/
↪Integer(2)) * P.sigma((Integer(2)*i+Integer(1))/Integer(2))
...      == P.e(Integer(2)*i/Integer(2)) * P.L(Integer(2)*i/Integer(2)) for i
↪in range(Integer(1),k))
True

```

The same tests for a half integer partition algebra:

```

sage: k = 7/2
sage: R.<n> = QQ[]
sage: P = PartitionAlgebra(k, n)
sage: L = [P.L(i/2) for i in range(1, 2*k+1)]
sage: all(x.dual() == x for x in L)
True
sage: all(x * y == y * x for x, y in Subsets(L, 2)) # long time
True
sage: Lsum = sum(L)
sage: gens = [P.s(i) for i in range(1, k-1/2)]
sage: gens += [P.e(i/2) for i in range(1, 2*k)]
sage: all(x * Lsum == Lsum * x for x in gens)
True
sage: all(P.e((2*i+1)/2) * P.sigma(2*i/2) * P.e((2*i+1)/2)
.....:      == (n - P.L((2*i-1)/2)) * P.e((2*i+1)/2) for i in range(1, floor(k)))
True
sage: all(P.e(i/2) * (P.L(i/2) + P.L((i+1)/2))
.....:      == (P.L(i/2) + P.L((i+1)/2)) * P.e(i/2)
.....:      == n * P.e(i/2) for i in range(1, 2*k))
True
sage: all(P.sigma(2*i/2) * P.e((2*i-1)/2) * P.e(2*i/2)
.....:      == P.L(2*i/2) * P.e(2*i/2) for i in range(1, ceil(k)))
True
sage: all(P.e(2*i/2) * P.e((2*i-1)/2) * P.sigma(2*i/2)
.....:      == P.e(2*i/2) * P.L(2*i/2) for i in range(1, ceil(k)))
True
sage: all(P.sigma((2*i+1)/2) * P.e((2*i+1)/2) * P.e(2*i/2)
.....:      == P.L(2*i/2) * P.e(2*i/2) for i in range(1, floor(k)))
True
sage: all(P.e(2*i/2) * P.e((2*i+1)/2) * P.sigma((2*i+1)/2)
.....:      == P.e(2*i/2) * P.L(2*i/2) for i in range(1, floor(k)))
True

```

```

>>> from sage.all import *
>>> k = Integer(7)/Integer(2)
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P = PartitionAlgebra(k, n)
>>> L = [P.L(i/Integer(2)) for i in range(Integer(1), Integer(2)*k+Integer(1))]
>>> all(x.dual() == x for x in L)
True
>>> all(x * y == y * x for x, y in Subsets(L, Integer(2))) # long time
True
>>> Lsum = sum(L)
>>> gens = [P.s(i) for i in range(Integer(1), k-Integer(1)/Integer(2))]
>>> gens += [P.e(i/Integer(2)) for i in range(Integer(1), Integer(2)*k)]
>>> all(x * Lsum == Lsum * x for x in gens)
True
>>> all(P.e((Integer(2)*i+Integer(1))/Integer(2)) * P.sigma(Integer(2)*i/
↳ Integer(2)) * P.e((Integer(2)*i+Integer(1))/Integer(2))
...      == (n - P.L((Integer(2)*i-Integer(1))/Integer(2))) * P.
↳ e((Integer(2)*i+Integer(1))/Integer(2)) for i in range(Integer(1), floor(k)))
True

```

(continues on next page)

(continued from previous page)

```

>>> all(P.e(i/Integer(2)) * (P.L(i/Integer(2)) + P.L((i+Integer(1))/
↪Integer(2)))
...      == (P.L(i/Integer(2)) + P.L((i+Integer(1))/Integer(2))) * P.e(i/
↪Integer(2))
...      == n * P.e(i/Integer(2)) for i in range(Integer(1),Integer(2)*k))
True
>>> all(P.sigma(Integer(2)*i/Integer(2)) * P.e((Integer(2)*i-Integer(1))/
↪Integer(2)) * P.e(Integer(2)*i/Integer(2))
...      == P.L(Integer(2)*i/Integer(2)) * P.e(Integer(2)*i/Integer(2)) for i
↪in range(Integer(1),ceil(k))
True
>>> all(P.e(Integer(2)*i/Integer(2)) * P.e((Integer(2)*i-Integer(1))/
↪Integer(2)) * P.sigma(Integer(2)*i/Integer(2))
...      == P.e(Integer(2)*i/Integer(2)) * P.L(Integer(2)*i/Integer(2)) for i
↪in range(Integer(1),ceil(k))
True
>>> all(P.sigma((Integer(2)*i+Integer(1))/Integer(2)) * P.
↪e((Integer(2)*i+Integer(1))/Integer(2)) * P.e(Integer(2)*i/Integer(2))
...      == P.L(Integer(2)*i/Integer(2)) * P.e(Integer(2)*i/Integer(2)) for i
↪in range(Integer(1),floor(k))
True
>>> all(P.e(Integer(2)*i/Integer(2)) * P.e((Integer(2)*i+Integer(1))/
↪Integer(2)) * P.sigma((Integer(2)*i+Integer(1))/Integer(2))
...      == P.e(Integer(2)*i/Integer(2)) * P.L(Integer(2)*i/Integer(2)) for i
↪in range(Integer(1),floor(k))
True

```

orbit_basis()

Return the orbit basis of self.

EXAMPLES:

```

sage: R.<x> = QQ[]
sage: P2 = PartitionAlgebra(2, x, R)
sage: O2 = P2.orbit_basis(); O2
Orbit basis of Partition Algebra of rank 2 with parameter x over
Univariate Polynomial Ring in x over Rational Field
sage: pp = 7 * P2[{-1}, {-2, 1, 2}] - 2 * P2[{-2}, {-1, 1}, {2}]; pp
-2*P[{-2}, {-1, 1}, {2}] + 7*P[{-2, 1, 2}, {-1}]
sage: op = pp.to_orbit_basis(); op
-2*O[{-2}, {-1, 1}, {2}] - 2*O[{-2}, {-1, 1, 2}]
- 2*O[{-2, -1, 1}, {2}] + 5*O[{-2, -1, 1, 2}]
+ 7*O[{-2, 1, 2}, {-1}] - 2*O[{-2, 2}, {-1, 1}]
sage: op == O2(op)
True
sage: pp * op.leading_term()
4*P[{-2}, {-1, 1}, {2}] - 4*P[{-2, -1, 1}, {2}]
+ 14*P[{-2, -1, 1, 2}] - 14*P[{-2, 1, 2}, {-1}]

```

```

>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> P2 = PartitionAlgebra(Integer(2), x, R)

```

(continues on next page)

(continued from previous page)

```

>>> O2 = P2.orbit_basis(); O2
Orbit basis of Partition Algebra of rank 2 with parameter x over
Univariate Polynomial Ring in x over Rational Field
>>> pp = Integer(7) * P2[{-Integer(1)}, {-Integer(2), Integer(1), Integer(2)}
↪] - Integer(2) * P2[{-Integer(2)}, {-Integer(1), Integer(1)}, {Integer(2)}];
↪ pp
-2*P{{-2}, {-1, 1}, {2}} + 7*P{{-2, 1, 2}, {-1}}
>>> op = pp.to_orbit_basis(); op
-2*O{{-2}, {-1, 1}, {2}} - 2*O{{-2}, {-1, 1, 2}}
- 2*O{{-2, -1, 1}, {2}} + 5*O{{-2, -1, 1, 2}}
+ 7*O{{-2, 1, 2}, {-1}} - 2*O{{-2, 2}, {-1, 1}}
>>> op == O2(op)
True
>>> pp * op.leading_term()
4*P{{-2}, {-1, 1}, {2}} - 4*P{{-2, -1, 1}, {2}}
+ 14*P{{-2, -1, 1, 2}} - 14*P{{-2, 1, 2}, {-1}}

```

potts_representation (*y=None*)Return the *PottsRepresentation* with magnetic field direction *y* of *self*.**Note**The deformation parameter *d* of *self* must be a positive integer.

INPUT:

- *y* – (optional) an integer between 1 and *d*; ignored if the order of *self* is an integer, otherwise the default is 1

EXAMPLES:

```

sage: PA = algebras.Partition(5/2, QQ(4))
sage: PR = PA.potts_representation()

sage: PA = algebras.Partition(5/2, 3/2)
sage: PA.potts_representation()
Traceback (most recent call last):
...
ValueError: the partition algebra deformation parameter must
be a positive integer

```

```

>>> from sage.all import *
>>> PA = algebras.Partition(Integer(5)/Integer(2), QQ(Integer(4)))
>>> PR = PA.potts_representation()

>>> PA = algebras.Partition(Integer(5)/Integer(2), Integer(3)/Integer(2))
>>> PA.potts_representation()
Traceback (most recent call last):
...
ValueError: the partition algebra deformation parameter must
be a positive integer

```


s(*i*)

Return the i -th simple transposition s_i in `self`.

Borrowing the notation from the symmetric group, the i -th simple transposition s_i has blocks of the form $\{-i, i+1\}$, $\{-i-1, i\}$. Other blocks are of the form $\{-j, j\}$.

INPUT:

- i – integer between 1 and $k-1$

EXAMPLES:

```
sage: R.<n> = QQ[]
sage: P3 = PartitionAlgebra(3, n)
sage: P3.s(1)
P{{-3, 3}, {-2, 1}, {-1, 2}}
sage: P3.s(2)
P{{-3, 2}, {-2, 3}, {-1, 1}}

sage: R.<n> = ZZ[]
sage: P2h = PartitionAlgebra(5/2, n)
sage: P2h.s(1)
P{{-3, 3}, {-2, 1}, {-1, 2}}
```

```
>>> from sage.all import *
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P3 = PartitionAlgebra(Integer(3), n)
>>> P3.s(Integer(1))
P{{-3, 3}, {-2, 1}, {-1, 2}}
>>> P3.s(Integer(2))
P{{-3, 2}, {-2, 3}, {-1, 1}}

>>> R = ZZ['n']; (n,) = R._first_ngens(1)
>>> P2h = PartitionAlgebra(Integer(5)/Integer(2), n)
>>> P2h.s(Integer(1))
P{{-3, 3}, {-2, 1}, {-1, 2}}
```

sigma(*i*)

Return the element σ_i from [Eny2012] of `self`.

INPUT:

- i – half integer between $1/2$ and $k-1/2$

Note

In [Cre2020] and [Eny2013], these are the elements σ_{2i} .

EXAMPLES:

```
sage: R.<n> = QQ[]
sage: P3 = PartitionAlgebra(3, n)
sage: P3.sigma(1)
P{{-3, 3}, {-2, 2}, {-1, 1}}
sage: P3.sigma(3/2)
```

(continues on next page)

(continued from previous page)

```

P{{-3, 3}, {-2, 1}, {-1, 2}}
sage: P3.sigma(2)
-P{{-3, -1, 1, 3}, {-2, 2}} + P{{-3, -1, 3}, {-2, 1, 2}}
+ P{{-3, 1, 3}, {-2, -1, 2}} - P{{-3, 3}, {-2, -1, 1, 2}}
+ P{{-3, 3}, {-2, 2}, {-1, 1}}
sage: P3.sigma(5/2)
-P{{-3, -1, 1, 2}, {-2, 3}} + P{{-3, -1, 2}, {-2, 1, 3}}
+ P{{-3, 1, 2}, {-2, -1, 3}} - P{{-3, 2}, {-2, -1, 1, 3}}
+ P{{-3, 2}, {-2, 3}, {-1, 1}}

```

```

>>> from sage.all import *
>>> R = QQ['n']; (n,) = R._first_ngens(1)
>>> P3 = PartitionAlgebra(Integer(3), n)
>>> P3.sigma(Integer(1))
P{{-3, 3}, {-2, 2}, {-1, 1}}
>>> P3.sigma(Integer(3)/Integer(2))
P{{-3, 3}, {-2, 1}, {-1, 2}}
>>> P3.sigma(Integer(2))
-P{{-3, -1, 1, 3}, {-2, 2}} + P{{-3, -1, 3}, {-2, 1, 2}}
+ P{{-3, 1, 3}, {-2, -1, 2}} - P{{-3, 3}, {-2, -1, 1, 2}}
+ P{{-3, 3}, {-2, 2}, {-1, 1}}
>>> P3.sigma(Integer(5)/Integer(2))
-P{{-3, -1, 1, 2}, {-2, 3}} + P{{-3, -1, 2}, {-2, 1, 3}}
+ P{{-3, 1, 2}, {-2, -1, 3}} - P{{-3, 2}, {-2, -1, 1, 3}}
+ P{{-3, 2}, {-2, 3}, {-1, 1}}

```

We test the relations in Lemma 2.2.3(1) in [Cre2020] (v1):

```

sage: k = 4
sage: R.<x> = QQ[]
sage: P = PartitionAlgebra(k, x)
sage: all(P.sigma(i/2).dual() == P.sigma(i/2))
.....: for i in range(1, 2*k))
True
sage: all(P.sigma(i)*P.sigma(i+1/2) == P.sigma(i+1/2)*P.sigma(i) == P.s(i))
.....: for i in range(1, floor(k)))
True
sage: all(P.sigma(i)*P.e(i) == P.e(i)*P.sigma(i) == P.e(i))
.....: for i in range(1, floor(k)))
True
sage: all(P.sigma(i+1/2)*P.e(i) == P.e(i)*P.sigma(i+1/2) == P.e(i))
.....: for i in range(1, floor(k)))
True

sage: k = 9/2
sage: R.<x> = QQ[]
sage: P = PartitionAlgebra(k, x)
sage: all(P.sigma(i/2).dual() == P.sigma(i/2))
.....: for i in range(1, 2*k-1))
True
sage: all(P.sigma(i)*P.sigma(i+1/2) == P.sigma(i+1/2)*P.sigma(i) == P.s(i))
.....: for i in range(1, k-1/2))

```

(continues on next page)

(continued from previous page)

```

True
sage: all(P.sigma(i)*P.e(i) == P.e(i)*P.sigma(i) == P.e(i)
....:     for i in range(1,floor(k)))
True
sage: all(P.sigma(i+1/2)*P.e(i) == P.e(i)*P.sigma(i+1/2) == P.e(i)
....:     for i in range(1,floor(k)))
True

```

```

>>> from sage.all import *
>>> k = Integer(4)
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> P = PartitionAlgebra(k, x)
>>> all(P.sigma(i/Integer(2)).dual() == P.sigma(i/Integer(2))
...     for i in range(Integer(1),Integer(2)*k))
True
>>> all(P.sigma(i)*P.sigma(i+Integer(1)/Integer(2)) == P.sigma(i+Integer(1)/
↳Integer(2))*P.sigma(i) == P.s(i)
...     for i in range(Integer(1),floor(k)))
True
>>> all(P.sigma(i)*P.e(i) == P.e(i)*P.sigma(i) == P.e(i)
...     for i in range(Integer(1),floor(k)))
True
>>> all(P.sigma(i+Integer(1)/Integer(2))*P.e(i) == P.e(i)*P.
↳sigma(i+Integer(1)/Integer(2)) == P.e(i)
...     for i in range(Integer(1),floor(k)))
True

>>> k = Integer(9)/Integer(2)
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> P = PartitionAlgebra(k, x)
>>> all(P.sigma(i/Integer(2)).dual() == P.sigma(i/Integer(2))
...     for i in range(Integer(1),Integer(2)*k-Integer(1)))
True
>>> all(P.sigma(i)*P.sigma(i+Integer(1)/Integer(2)) == P.sigma(i+Integer(1)/
↳Integer(2))*P.sigma(i) == P.s(i)
...     for i in range(Integer(1),k-Integer(1)/Integer(2)))
True
>>> all(P.sigma(i)*P.e(i) == P.e(i)*P.sigma(i) == P.e(i)
...     for i in range(Integer(1),floor(k)))
True
>>> all(P.sigma(i+Integer(1)/Integer(2))*P.e(i) == P.e(i)*P.
↳sigma(i+Integer(1)/Integer(2)) == P.e(i)
...     for i in range(Integer(1),floor(k)))
True

```

class sage.combinat.diagram_algebras.**PartitionDiagram** (*parent, d, check=True*)

Bases: *AbstractPartitionDiagram*

The element class for a partition diagram.

A partition diagram for an integer k is a partition of the set $\{1, \dots, k, -1, \dots, -k\}$

EXAMPLES:

```

sage: from sage.combinat.diagram_algebras import PartitionDiagram, PartitionDiagrams
sage: PartitionDiagrams(1)
Partition diagrams of order 1
sage: PartitionDiagrams(1).list()
[{{-1, 1}}, {{-1}, {1}}]
sage: PartitionDiagram([[1, -1]])
{{-1, 1}}
sage: PartitionDiagram(((1, -2), (2, -1))).parent()
Partition diagrams of order 2

```

```

>>> from sage.all import *
>>> from sage.combinat.diagram_algebras import PartitionDiagram, PartitionDiagrams
>>> PartitionDiagrams(Integer(1))
Partition diagrams of order 1
>>> PartitionDiagrams(Integer(1)).list()
[{{-1, 1}}, {{-1}, {1}}]
>>> PartitionDiagram([[Integer(1), -Integer(1)]])
{{-1, 1}}
>>> PartitionDiagram(((Integer(1), -Integer(2)), (Integer(2), -Integer(1)))).parent()
Partition diagrams of order 2

```

class sage.combinat.diagram_algebras.**PartitionDiagrams**(*order*, *category=None*)

Bases: *AbstractPartitionDiagrams*

This class represents all partition diagrams of integer or integer $+1/2$ order.

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: pd = da.PartitionDiagrams(1); pd
Partition diagrams of order 1
sage: pd.list()
[{{-1, 1}}, {{-1}, {1}}]

sage: pd = da.PartitionDiagrams(3/2); pd
Partition diagrams of order 3/2
sage: pd.list()
[{{-2, -1, 1, 2}},
 {{-2, 1, 2}, {-1}},
 {{-2, 2}, {-1, 1}},
 {{-2, -1, 2}, {1}},
 {{-2, 2}, {-1}, {1}}]

```

```

>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> pd = da.PartitionDiagrams(Integer(1)); pd
Partition diagrams of order 1
>>> pd.list()
[{{-1, 1}}, {{-1}, {1}}]

>>> pd = da.PartitionDiagrams(Integer(3)/Integer(2)); pd
Partition diagrams of order 3/2

```

(continues on next page)

(continued from previous page)

```
>>> pd.list()
[{{-2, -1, 1, 2}},
 {{-2, 1, 2}, {-1}},
 {{-2, 2}, {-1, 1}},
 {{-2, -1, 2}, {1}},
 {{-2, 2}, {-1}, {1}}]
```

Elementalias of *PartitionDiagram***cardinality()**The cardinality of partition diagrams of half-integer order n is the $2n$ -th Bell number.**EXAMPLES:**

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.PartitionDiagrams(3)
sage: pd.cardinality()
203

sage: pd = da.PartitionDiagrams(7/2)
sage: pd.cardinality()
877
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> pd = da.PartitionDiagrams(Integer(3))
>>> pd.cardinality()
203

>>> pd = da.PartitionDiagrams(Integer(7)/Integer(2))
>>> pd.cardinality()
877
```

class `sage.combinat.diagram_algebras.PlanarAlgebra` ($k, q, \text{base\_ring}, \text{prefix}$)Bases: *SubPartitionAlgebra*, *UnitDiagramMixin*

A planar algebra.

The planar algebra of rank k is an algebra with basis indexed by the collection of all planar set partitions of $\{1, \dots, k, -1, \dots, -k\}$.This algebra is thus a subalgebra of the partition algebra. For more information, see *PartitionAlgebra*.**INPUT:**

- k – rank of the algebra
- q – the deformation parameter q
- `base_ring` – (default: `None`) a ring containing q ; if `None` then just takes the parent of q
- `prefix` – (default: `'Pl'`) a label for the basis elements

EXAMPLES:We define the planar algebra of rank 2 with parameter x over $\mathbf{Z}$:

```

sage: R.<x> = ZZ[]
sage: Pl = PlanarAlgebra(2, x, R); Pl
Planar Algebra of rank 2 with parameter x over Univariate Polynomial Ring in x
↳over Integer Ring
sage: Pl.basis().keys()
Planar diagrams of order 2
sage: Pl.basis().keys()([[ -1, 1], [2, -2]])
{{-2, 2}, {-1, 1}}
sage: Pl.basis().list()
[Pl{{-2}, {-1}, {1, 2}},
 Pl{{-2}, {-1}, {1}, {2}},
 Pl{{-2, 1}, {-1}, {2}},
 Pl{{-2, 2}, {-1}, {1}},
 Pl{{-2, 1, 2}, {-1}},
 Pl{{-2, 2}, {-1, 1}},
 Pl{{-2}, {-1, 1}, {2}},
 Pl{{-2}, {-1, 2}, {1}},
 Pl{{-2}, {-1, 1, 2}},
 Pl{{-2, -1}, {1, 2}},
 Pl{{-2, -1}, {1}, {2}},
 Pl{{-2, -1, 1}, {2}},
 Pl{{-2, -1, 2}, {1}},
 Pl{{-2, -1, 1, 2}}]
sage: E = Pl([[1, 2], [-1, -2]])
sage: E^2 == x*E
True
sage: E^5 == x^4*E
True

```

```

>>> from sage.all import *
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> Pl = PlanarAlgebra(Integer(2), x, R); Pl
Planar Algebra of rank 2 with parameter x over Univariate Polynomial Ring in x
↳over Integer Ring
>>> Pl.basis().keys()
Planar diagrams of order 2
>>> Pl.basis().keys()([[-Integer(1), Integer(1)], [Integer(2), -Integer(2)]])
{{-2, 2}, {-1, 1}}
>>> Pl.basis().list()
[Pl{{-2}, {-1}, {1, 2}},
 Pl{{-2}, {-1}, {1}, {2}},
 Pl{{-2, 1}, {-1}, {2}},
 Pl{{-2, 2}, {-1}, {1}},
 Pl{{-2, 1, 2}, {-1}},
 Pl{{-2, 2}, {-1, 1}},
 Pl{{-2}, {-1, 1}, {2}},
 Pl{{-2}, {-1, 2}, {1}},
 Pl{{-2}, {-1, 1, 2}},
 Pl{{-2, -1}, {1, 2}},
 Pl{{-2, -1}, {1}, {2}},
 Pl{{-2, -1, 1}, {2}},
 Pl{{-2, -1, 2}, {1}},

```

(continues on next page)

(continued from previous page)

```

Pl{{-2, -1, 1, 2}}]
>>> E = Pl([[Integer(1), Integer(2)], [-Integer(1), -Integer(2)]])
>>> E**Integer(2) == x**E
True
>>> E**Integer(5) == x**Integer(4)*E
True

```

class sage.combinat.diagram_algebras.**PlanarDiagram**(parent, d, check=True)

Bases: [AbstractPartitionDiagram](#)

The element class for a planar diagram.

A planar diagram for an integer k is a partition of the set $\{1, \dots, k, -1, \dots, -k\}$ so that the diagram is non-crossing.

EXAMPLES:

```

sage: from sage.combinat.diagram_algebras import PlanarDiagrams
sage: PlanarDiagrams(2)
Planar diagrams of order 2
sage: PlanarDiagrams(2).list()
[{{-2}, {-1}, {1, 2}},
 {{-2}, {-1}, {1}, {2}},
 {{-2, 1}, {-1}, {2}},
 {{-2, 2}, {-1}, {1}},
 {{-2, 1, 2}, {-1}},
 {{-2, 2}, {-1, 1}},
 {{-2}, {-1, 1}, {2}},
 {{-2}, {-1, 2}, {1}},
 {{-2}, {-1, 1, 2}},
 {{-2, -1}, {1, 2}},
 {{-2, -1}, {1}, {2}},
 {{-2, -1, 1}, {2}},
 {{-2, -1, 2}, {1}},
 {{-2, -1, 1, 2}}]

```

```

>>> from sage.all import *
>>> from sage.combinat.diagram_algebras import PlanarDiagrams
>>> PlanarDiagrams(Integer(2))
Planar diagrams of order 2
>>> PlanarDiagrams(Integer(2)).list()
[{{-2}, {-1}, {1, 2}},
 {{-2}, {-1}, {1}, {2}},
 {{-2, 1}, {-1}, {2}},
 {{-2, 2}, {-1}, {1}},
 {{-2, 1, 2}, {-1}},
 {{-2, 2}, {-1, 1}},
 {{-2}, {-1, 1}, {2}},
 {{-2}, {-1, 2}, {1}},
 {{-2}, {-1, 1, 2}},
 {{-2, -1}, {1, 2}},
 {{-2, -1}, {1}, {2}},
 {{-2, -1, 1}, {2}},

```

(continues on next page)

(continued from previous page)

```
{{-2, -1, 2}, {1}},
{{-2, -1, 1, 2}}]
```

check()

Check the validity of the input for `self`.

class `sage.combinat.diagram_algebras.PlanarDiagrams` (*order, category=None*)

Bases: *AbstractPartitionDiagrams*

All planar diagrams of integer or integer $+1/2$ order.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pld = da.PlanarDiagrams(1); pld
Planar diagrams of order 1
sage: pld.list()
[{{-1, 1}}, {{-1}, {1}}]

sage: pld = da.PlanarDiagrams(3/2); pld
Planar diagrams of order 3/2
sage: pld.list()
[{{-2, 1, 2}, {-1}},
 {{-2, 2}, {-1}, {1}},
 {{-2, 2}, {-1, 1}},
 {{-2, -1, 2}, {1}},
 {{-2, -1, 1, 2}}]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> pld = da.PlanarDiagrams(Integer(1)); pld
Planar diagrams of order 1
>>> pld.list()
[{{-1, 1}}, {{-1}, {1}}]

>>> pld = da.PlanarDiagrams(Integer(3)/Integer(2)); pld
Planar diagrams of order 3/2
>>> pld.list()
[{{-2, 1, 2}, {-1}},
 {{-2, 2}, {-1}, {1}},
 {{-2, 2}, {-1, 1}},
 {{-2, -1, 2}, {1}},
 {{-2, -1, 1, 2}}]
```

Element

alias of *PlanarDiagram*

cardinality()

Return the cardinality of `self`.

The number of all planar diagrams of order k is the $2k$ -th Catalan number.

EXAMPLES:


```
sage: import sage.combinat.diagram_algebras as da
sage: pld = da.PlanarDiagrams(3)
sage: pld.cardinality()
132
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> pld = da.PlanarDiagrams(Integer(3))
>>> pld.cardinality()
132
```

```
class sage.combinat.diagram_algebras.PottsRepresentation(PA, y)
```

Bases: `CombinatorialFreeModule`

The Potts representation of the partition algebra.

Let $P_n(d)$ be the *PartitionAlgebra* over R with the deformation parameter $d \in \mathbf{Z}_{>0}$ being a positive integer. Recall the multiplication convention of diagrams in $P_n(d)$ computing DD' by placing D above D' .

The *Potts representation* is the right $P_n(d)$ -module on $M = V^{\otimes n}$, with $V = R^d$, with the action given as follows. We identify the natural basis vectors in M with words of length n in the alphabet $\{1, \dots, d\}$ (which we call colors). For a basis vector w and diagram D , define $w \cdot D$ as the sum over all v such that every part in wDv (consider this as coloring the nodes of D) is given by the same color.

If n is a half integer, then there is an extra fixed color for the node $[n]$, which is called the *magnetic field direction* from the physics interpretation of this representation.

EXAMPLES:

In this example, we consider $R = \mathbf{Q}$ and use the Potts representation to construct the centralizer algebra of the left S_{d-1} -action on $V^{\otimes n}$ with $V = \mathbf{Q}^d$ being the permutation action.

```
sage: PA = algebras.Partition(5/2, QQ(2))
sage: PR = PA.potts_representation(2)
sage: mats = [PR.representation_matrix(x) for x in PA.basis()]
sage: MS = mats[0].parent()
sage: CM = MS.submodule(mats)
sage: CM.dimension()
16
```

```
>>> from sage.all import *
>>> PA = algebras.Partition(Integer(5)/Integer(2), QQ(Integer(2)))
>>> PR = PA.potts_representation(Integer(2))
>>> mats = [PR.representation_matrix(x) for x in PA.basis()]
>>> MS = mats[Integer(0)].parent()
>>> CM = MS.submodule(mats)
>>> CM.dimension()
16
```

We check that this commutes with the S_{d-1} -action:

```
sage: all((g * v) * x == g * (v * x) for g in PR.symmetric_group()
.....:      for v in PR.basis() for x in PA.basis())
True
```

```
>>> from sage.all import *
>>> all((g * v) * x == g * (v * x) for g in PR.symmetric_group()
...     for v in PR.basis() for x in PA.basis())
True
```

Next, we see that the centralizer of the S_d -action is smaller than the semisimple quotient of the partition algebra:

```
sage: PA.dimension()
52
sage: len(PA.radical_basis())
9
sage: SQ = PA.semisimple_quotient()
sage: SQ.dimension()
43
```

```
>>> from sage.all import *
>>> PA.dimension()
52
>>> len(PA.radical_basis())
9
>>> SQ = PA.semisimple_quotient()
>>> SQ.dimension()
43
```

Next, we get orthogonal idempotents that project onto the central orthogonal idempotents in the semisimple quotient and construct the corresponding Peirce summands $e_i P_n(d) e_i$:

```
sage: # long time
sage: orth_idems = PA.orthogonal_idempotents_central_mod_radical()
sage: algs = [PA.peirce_summand(idm, idm) for idm in orth_idems]
sage: [A.dimension() for A in algs]
[16, 2, 1, 25]
```

```
>>> from sage.all import *
>>> # long time
>>> orth_idems = PA.orthogonal_idempotents_central_mod_radical()
>>> algs = [PA.peirce_summand(idm, idm) for idm in orth_idems]
>>> [A.dimension() for A in algs]
[16, 2, 1, 25]
```

We saw that we obtain the entire endomorphism algebra since $d = 2$ and S_{d-1} is the trivial group. Hence, the 16 dimensional Peirce summand computed above is isomorphic to this endomorphism algebra (both are 4×4 matrix algebras over $\mathbb{Q}$). Hence, we have a natural quotient construction of the centralizer algebra from the partition algebra.

Next, we consider a case with a nontrivial S_d -action (now it is S_d since the partition algebra has integer rank). We perform the same computations as before:

```
sage: PA = algebras.Partition(2, QQ(2))
sage: PA.dimension()
15
sage: PA.semisimple_quotient().dimension()
10
```

(continues on next page)

(continued from previous page)

```

sage: orth_idems = PA.orthogonal_idempotents_central_mod_radical()
sage: algs = [PA.peirce_summand(idm, idm) for idm in orth_idems]
sage: [A.dimension() for A in algs]
[4, 2, 4, 1]

sage: PR = PA.potts_representation()
sage: mats = [PR.representation_matrix(x) for x in PA.basis()]
sage: MS = mats[0].parent()
sage: cat = Algebras(QQ).WithBasis().Subobjects()
sage: CM = MS.submodule(mats, category=cat)
sage: CM.dimension()
8

```

```

>>> from sage.all import *
>>> PA = algebras.Partition(Integer(2), QQ(Integer(2)))
>>> PA.dimension()
15
>>> PA.semisimple_quotient().dimension()
10
>>> orth_idems = PA.orthogonal_idempotents_central_mod_radical()
>>> algs = [PA.peirce_summand(idm, idm) for idm in orth_idems]
>>> [A.dimension() for A in algs]
[4, 2, 4, 1]

>>> PR = PA.potts_representation()
>>> mats = [PR.representation_matrix(x) for x in PA.basis()]
>>> MS = mats[Integer(0)].parent()
>>> cat = Algebras(QQ).WithBasis().Subobjects()
>>> CM = MS.submodule(mats, category=cat)
>>> CM.dimension()
8

```

To do the remainder of the computation, we need to monkey patch a `product_on_basis` method:

```

sage: CM.product_on_basis
NotImplemented
sage: CM.product_on_basis = lambda x,y: CM.retract(CM.basis()[x].lift() * CM.
↪basis()[y].lift())
sage: CM.orthogonal_idempotents_central_mod_radical()
(1/2*B[0] + 1/2*B[3] + 1/2*B[5] + 1/2*B[6],
 1/2*B[0] - 1/2*B[3] + 1/2*B[5] - 1/2*B[6])
sage: CM.peirce_decomposition()
[[Free module generated by {0, 1, 2, 3} over Rational Field,
  Free module generated by {} over Rational Field],
 [Free module generated by {} over Rational Field,
  Free module generated by {0, 1, 2, 3} over Rational Field]]

```

```

>>> from sage.all import *
>>> CM.product_on_basis
NotImplemented
>>> CM.product_on_basis = lambda x,y: CM.retract(CM.basis()[x].lift() * CM.
↪basis()[y].lift())

```

(continues on next page)

(continued from previous page)

```

>>> CM.orthogonal_idempotents_central_mod_radical()
(1/2*B[0] + 1/2*B[3] + 1/2*B[5] + 1/2*B[6],
 1/2*B[0] - 1/2*B[3] + 1/2*B[5] - 1/2*B[6])
>>> CM.peirce_decomposition()
[[Free module generated by {0, 1, 2, 3} over Rational Field,
  Free module generated by {} over Rational Field],
 [Free module generated by {} over Rational Field,
  Free module generated by {0, 1, 2, 3} over Rational Field]]

```

Hence, we see that the centralizer algebra is isomorphic to a product of two 2×2 matrix algebras (over $\mathbf{Q}$), which are naturally a part of the partition algebra decomposition.

Lastly, we verify the commuting actions:

```

sage: all((g * v) * x == g * (v * x) for g in PR.symmetric_group()
....:      for v in PR.basis() for x in PA.basis())
True

```

```

>>> from sage.all import *
>>> all((g * v) * x == g * (v * x) for g in PR.symmetric_group()
...     for v in PR.basis() for x in PA.basis())
True

```

REFERENCES:

- [MR1998]

class Element

Bases: IndexedFreeModuleElement

magnetic_field_direction()

Return the magnetic field direction defining self.

EXAMPLES:

```

sage: PA = algebras.Partition(7/2, QQ(4))
sage: PR = PA.potts_representation(2)
sage: PR.magnetic_field_direction()
2

```

```

>>> from sage.all import *
>>> PA = algebras.Partition(Integer(7)/Integer(2), QQ(Integer(4)))
>>> PR = PA.potts_representation(Integer(2))
>>> PR.magnetic_field_direction()
2

```

number_of_colors()

Return the number of colors defining self.

EXAMPLES:

```

sage: PA = algebras.Partition(3, QQ(4))
sage: PR = PA.potts_representation()
sage: PR.number_of_colors()
4

```

```
>>> from sage.all import *
>>> PA = algebras.Partition(Integer(3), QQ(Integer(4)))
>>> PR = PA.potts_representation()
>>> PR.number_of_colors()
4
```

number_of_factors()

Return the number of factors defining self.

EXAMPLES:

```
sage: PA = algebras.Partition(7/2, QQ(4))
sage: PR = PA.potts_representation()
sage: PR.number_of_factors()
3
```

```
>>> from sage.all import *
>>> PA = algebras.Partition(Integer(7)/Integer(2), QQ(Integer(4)))
>>> PR = PA.potts_representation()
>>> PR.number_of_factors()
3
```

partition_algebra()

Return the partition algebra that self is a representation of.

EXAMPLES:

```
sage: PA = algebras.Partition(3, QQ(4))
sage: PR = PA.potts_representation()
sage: PR.partition_algebra() is PA
True
```

```
>>> from sage.all import *
>>> PA = algebras.Partition(Integer(3), QQ(Integer(4)))
>>> PR = PA.potts_representation()
>>> PR.partition_algebra() is PA
True
```

representation_matrix(elt)

Return the representation matrix of self in self.

EXAMPLES:

```
sage: PA = algebras.Partition(7/2, QQ(2))
sage: PR = PA.potts_representation()
sage: PR.representation_matrix(PA.an_element())
[7 0 3 0 2 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
```

(continues on next page)

(continued from previous page)

```

[0 0 0 0 0 0 0 0]

sage: all(b.to_vector() * PR.representation_matrix(x) # long time
.....:      == (b * x).to_vector()
.....:      for b in PR.basis() for x in PA.basis())
True

sage: PA = algebras.Partition(2, QQ(2))
sage: PR = PA.potts_representation()
sage: [PR.representation_matrix(x) for x in PA.basis()]
[
[1 0 0 0] [1 0 1 0] [1 1 0 0] [1 0 0 1] [1 1 1 1] [1 0 0 0]
[0 0 0 0] [0 0 0 0] [0 0 0 0] [0 0 0 0] [0 0 0 0] [1 0 0 0]
[0 0 0 0] [0 0 0 0] [0 0 0 0] [0 0 0 0] [0 0 0 0] [0 0 0 1]
[0 0 0 1], [0 1 0 1], [0 0 1 1], [1 0 0 1], [1 1 1 1], [0 0 0 1],

[1 0 0 0] [1 0 1 0] [1 0 0 0] [1 0 0 0] [1 0 1 0] [1 1 0 0]
[0 0 1 0] [1 0 1 0] [0 1 0 0] [0 0 0 1] [0 1 0 1] [1 1 0 0]
[0 1 0 0] [0 1 0 1] [0 0 1 0] [1 0 0 0] [1 0 1 0] [0 0 1 1]
[0 0 0 1], [0 1 0 1], [0 0 0 1], [0 0 0 1], [0 1 0 1], [0 0 1 1],

[1 1 0 0] [1 0 0 1] [1 1 1 1]
[0 0 1 1] [1 0 0 1] [1 1 1 1]
[1 1 0 0] [1 0 0 1] [1 1 1 1]
[0 0 1 1], [1 0 0 1], [1 1 1 1]
]

sage: PA = algebras.Partition(5/2, QQ(2))
sage: PR = PA.potts_representation()
sage: all(PR.representation_matrix(x) * PR.representation_matrix(y) # long
↪time
.....:      == PR.representation_matrix(x * y)
.....:      for x in PA.basis() for y in PA.basis())
True

sage: PA = algebras.Partition(2, QQ(4))
sage: PR = PA.potts_representation()
sage: all(PR.representation_matrix(x) * PR.representation_matrix(y)
.....:      == PR.representation_matrix(x * y)
.....:      for x in PA.basis() for y in PA.basis())
True

```

```

>>> from sage.all import *
>>> PA = algebras.Partition(Integer(7)/Integer(2), QQ(Integer(2)))
>>> PR = PA.potts_representation()
>>> PR.representation_matrix(PA.an_element())
[7 0 3 0 2 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]

```

(continues on next page)

(continued from previous page)

```

[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]

>>> all(b.to_vector() * PR.representation_matrix(x) # long time
...     == (b * x).to_vector()
...     for b in PR.basis() for x in PA.basis())
True

>>> PA = algebras.Partition(Integer(2), QQ(Integer(2)))
>>> PR = PA.potts_representation()
>>> [PR.representation_matrix(x) for x in PA.basis()]
[
[1 0 0 0] [1 0 1 0] [1 1 0 0] [1 0 0 1] [1 1 1 1] [1 0 0 0]
[0 0 0 0] [0 0 0 0] [0 0 0 0] [0 0 0 0] [0 0 0 0] [1 0 0 0]
[0 0 0 0] [0 0 0 0] [0 0 0 0] [0 0 0 0] [0 0 0 0] [0 0 0 1]
[0 0 0 1], [0 1 0 1], [0 0 1 1], [1 0 0 1], [1 1 1 1], [0 0 0 1],
<BLANKLINE>
[1 0 0 0] [1 0 1 0] [1 0 0 0] [1 0 0 0] [1 0 1 0] [1 1 0 0]
[0 0 1 0] [1 0 1 0] [0 1 0 0] [0 0 0 1] [0 1 0 1] [1 1 0 0]
[0 1 0 0] [0 1 0 1] [0 0 1 0] [1 0 0 0] [1 0 1 0] [0 0 1 1]
[0 0 0 1], [0 1 0 1], [0 0 0 1], [0 0 0 1], [0 1 0 1], [0 0 1 1],
<BLANKLINE>
[1 1 0 0] [1 0 0 1] [1 1 1 1]
[0 0 1 1] [1 0 0 1] [1 1 1 1]
[1 1 0 0] [1 0 0 1] [1 1 1 1]
[0 0 1 1], [1 0 0 1], [1 1 1 1]
]

>>> PA = algebras.Partition(Integer(5)/Integer(2), QQ(Integer(2)))
>>> PR = PA.potts_representation()
>>> all(PR.representation_matrix(x) * PR.representation_matrix(y) # long time
...     == PR.representation_matrix(x * y)
...     for x in PA.basis() for y in PA.basis())
True

>>> PA = algebras.Partition(Integer(2), QQ(Integer(4)))
>>> PR = PA.potts_representation()
>>> all(PR.representation_matrix(x) * PR.representation_matrix(y)
...     == PR.representation_matrix(x * y)
...     for x in PA.basis() for y in PA.basis())
True

```

symmetric_group()

Return the symmetric group that naturally acts on self.

EXAMPLES:

```

sage: PA = algebras.Partition(3, QQ(4))
sage: PR = PA.potts_representation()
sage: PR.symmetric_group()
Symmetric group of order 4! as a permutation group

```

(continues on next page)

(continued from previous page)

```

sage: PA = algebras.Partition(7/2, QQ(4))
sage: PR = PA.potts_representation()
sage: PR.symmetric_group().domain()
{2, 3, 4}
sage: PR = PA.potts_representation(2)
sage: PR.symmetric_group().domain()
{1, 3, 4}
sage: PR = PA.potts_representation(4)
sage: PR.symmetric_group().domain()
{1, 2, 3}

```

```

>>> from sage.all import *
>>> PA = algebras.Partition(Integer(3), QQ(Integer(4)))
>>> PR = PA.potts_representation()
>>> PR.symmetric_group()
Symmetric group of order 4! as a permutation group

>>> PA = algebras.Partition(Integer(7)/Integer(2), QQ(Integer(4)))
>>> PR = PA.potts_representation()
>>> PR.symmetric_group().domain()
{2, 3, 4}
>>> PR = PA.potts_representation(Integer(2))
>>> PR.symmetric_group().domain()
{1, 3, 4}
>>> PR = PA.potts_representation(Integer(4))
>>> PR.symmetric_group().domain()
{1, 2, 3}

```

class sage.combinat.diagram_algebras.**PropagatingIdeal** (*k, q, base_ring, prefix*)

Bases: *SubPartitionAlgebra*

A propagating ideal.

The propagating ideal of rank k is a non-unital algebra with basis indexed by the collection of ideal set partitions of $\{1, \dots, k, -1, \dots, -k\}$. We say a set partition is *ideal* if its propagating number is less than k .

This algebra is a non-unital subalgebra and an ideal of the partition algebra. For more information, see *PartitionAlgebra*.

EXAMPLES:

We now define the propagating ideal of rank 2 with parameter x over $\mathbb{Z}$:

```

sage: R.<x> = QQ[]
sage: I = PropagatingIdeal(2, x, R); I
Propagating Ideal of rank 2 with parameter x
over Univariate Polynomial Ring in x over Rational Field
sage: I.basis().keys()
Ideal diagrams of order 2
sage: I.basis().list()
[I{{-2, -1, 1, 2}},
 I{{-2, 1, 2}, {-1}},
 I{{-2}, {-1, 1, 2}},
 I{{-2, -1}, {1, 2}},

```

(continues on next page)

(continued from previous page)

```

I{{-2}, {-1}, {1, 2}},
I{{-2, -1, 1}, {2}},
I{{-2, 1}, {-1}, {2}},
I{{-2, -1, 2}, {1}},
I{{-2, 2}, {-1}, {1}},
I{{-2}, {-1, 1}, {2}},
I{{-2}, {-1, 2}, {1}},
I{{-2, -1}, {1}, {2}},
I{{-2}, {-1}, {1}, {2}}}
sage: E = I([[1, 2], [-1, -2]])
sage: E^2 == x*E
True
sage: E^5 == x^4*E
True

```

```

>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> I = PropagatingIdeal(Integer(2), x, R); I
Propagating Ideal of rank 2 with parameter x
over Univariate Polynomial Ring in x over Rational Field
>>> I.basis().keys()
Ideal diagrams of order 2
>>> I.basis().list()
[I{{-2, -1, 1, 2}},
 I{{-2, 1, 2}, {-1}},
 I{{-2}, {-1, 1, 2}},
 I{{-2, -1}, {1, 2}},
 I{{-2}, {-1}, {1, 2}},
 I{{-2, -1, 1}, {2}},
 I{{-2, 1}, {-1}, {2}},
 I{{-2, -1, 2}, {1}},
 I{{-2, 2}, {-1}, {1}},
 I{{-2}, {-1, 1}, {2}},
 I{{-2}, {-1, 2}, {1}},
 I{{-2, -1}, {1}, {2}},
 I{{-2}, {-1}, {1}, {2}}}
>>> E = I([[Integer(1), Integer(2)], [-Integer(1), -Integer(2)]])
>>> E**Integer(2) == x*E
True
>>> E**Integer(5) == x**Integer(4)*E
True

```

class `Element`

Bases: `Element`

An element of a propagating ideal.

We need to take care of exponents since we are not unital.

class `sage.combinat.diagram_algebras.SubPartitionAlgebra` (*k*, *q*, *base_ring*, *prefix*, *diagrams*, *category=None*)

Bases: `DiagramBasis`

A subalgebra of the partition algebra in the diagram basis indexed by a subset of the diagrams.

class Element

Bases: *Element*

to_orbit_basis()

Return *self* in the orbit basis of the associated ambient partition algebra.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: B = BrauerAlgebra(2, x, R)
sage: bb = B([[-2, -1], [1, 2]]); bb
B{{-2, -1}, {1, 2}}
sage: bb.to_orbit_basis()
O{{-2, -1}, {1, 2}} + O{{-2, -1, 1, 2}}
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> B = BrauerAlgebra(Integer(2), x, R)
>>> bb = B([[-Integer(2), -Integer(1)], [Integer(1), Integer(2)]]); bb
B{{-2, -1}, {1, 2}}
>>> bb.to_orbit_basis()
O{{-2, -1}, {1, 2}} + O{{-2, -1, 1, 2}}
```

ambient()

Return the partition algebra *self* is a sub-algebra of.

EXAMPLES:

```
sage: x = var('x') #_
↪needs sage.symbolic
sage: BA = BrauerAlgebra(2, x) #_
↪needs sage.symbolic
sage: BA.ambient() #_
↪needs sage.symbolic
Partition Algebra of rank 2 with parameter x over Symbolic Ring
```

```
>>> from sage.all import *
>>> x = var('x') #_
↪needs sage.symbolic
>>> BA = BrauerAlgebra(Integer(2), x) #_
↪ # needs sage.symbolic
>>> BA.ambient() #_
↪needs sage.symbolic
Partition Algebra of rank 2 with parameter x over Symbolic Ring
```

lift()

Return the lift map from diagram subalgebra to the ambient space.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: BA = BrauerAlgebra(2, x, R)
sage: E = BA([[1, 2], [-1, -2]])
sage: lifted = BA.lift(E); lifted
```

(continues on next page)

(continued from previous page)

```
B{{-2, -1}, {1, 2}}
sage: lifted.parent() is BA.ambient()
True
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> BA = BrauerAlgebra(Integer(2), x, R)
>>> E = BA([[Integer(1), Integer(2)], [-Integer(1), -Integer(2)]])
>>> lifted = BA.lift(E); lifted
B{{-2, -1}, {1, 2}}
>>> lifted.parent() is BA.ambient()
True
```

retract(*x*)

Retract an appropriate partition algebra element to the corresponding element in the partition subalgebra.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: BA = BrauerAlgebra(2, x, R)
sage: PA = BA.ambient()
sage: E = PA([[1, 2], [-1, -2]])
sage: BA.retract(E) in BA
True
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> BA = BrauerAlgebra(Integer(2), x, R)
>>> PA = BA.ambient()
>>> E = PA([[Integer(1), Integer(2)], [-Integer(1), -Integer(2)]])
>>> BA.retract(E) in BA
True
```

`sage.combinat.diagram_algebras.TL_diagram_ascii_art(diagram, use_unicode=False, blobs=[])`

Return ascii art for a Temperley-Lieb diagram *diagram*.

INPUT:

- *diagram* – list of pairs of matchings of the set $\{-1, \dots, -n, 1, \dots, n\}$
- *use_unicode* – boolean (default: `False`); whether or not to use unicode art instead of ascii art
- *blobs* – (optional) a list of matchings with blobs on them

EXAMPLES:

```
sage: from sage.combinat.diagram_algebras import TL_diagram_ascii_art
sage: TL = [(-15, -12), (-14, -13), (-11, 15), (-10, 14), (-9, -6),
.....:      (-8, -7), (-5, -4), (-3, 1), (-2, -1), (2, 3), (4, 5),
.....:      (6, 11), (7, 8), (9, 10), (12, 13)]
sage: TL_diagram_ascii_art(TL, use_unicode=False)
o o o o o o o o o o o o o o o
| '- ' '- ' | '- ' '- ' | '- ' | |
|           '-----'      | |
```

(continues on next page)

(continued from previous page)

```
sage: TL_diagram_ascii_art(TL, use_unicode=True)
```

```
sage: TL = [(-20,-9), (-19,-10), (-18,-11), (-17,-16), (-15,-12), (2,3),  
.....:      (-14,-13), (-8,16), (-7,7), (-6,6), (-5,1), (-4,-3), (-2,-1),  
.....:      (4,5), (8,15), (9,10), (11,14), (12,13), (17,20), (18,19)]
```

```
sage: TL_diagram_ascii_art(TL, use_unicode=False, blobs=[(-2,-1), (-5,1)])
```

```
sage: TL_diagram_ascii_art(TL, use_unicode=True, blobs=[(-2,-1), (-5,1)])
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram_algebras import TL_diagram_ascii_art
>>> TL = [(-Integer(15),-Integer(12)), (-Integer(14),-Integer(13)), (-Integer(11),
↪ Integer(15)), (-Integer(10),Integer(14)), (-Integer(9),-Integer(6)),
...      (-Integer(8),-Integer(7)), (-Integer(5),-Integer(4)), (-Integer(3),
↪ Integer(1)), (-Integer(2),-Integer(1)), (Integer(2),Integer(3)), (Integer(4),
↪ Integer(5)),
...      (Integer(6),Integer(11)), (Integer(7), Integer(8)), (Integer(9),
↪ Integer(10)), (Integer(12),Integer(13))]
```

(continues on next page)

(continues on next page)

(continued from previous page)



```
class sage.combinat.diagram_algebras.TemperleyLiebAlgebra(k, q, base_ring, prefix)
```

Bases: [SubPartitionAlgebra](#), [UnitDiagramMixin](#)

A Temperley–Lieb algebra.

The Temperley–Lieb algebra of rank k is an algebra with basis indexed by the collection of planar set partitions of $\{1, \dots, k, -1, \dots, -k\}$ with block size 2.

This algebra is thus a subalgebra of the partition algebra. For more information, see [PartitionAlgebra](#).

INPUT:

- k – rank of the algebra
- q – the deformation parameter q
- `base_ring` – (default: `None`) a ring containing q ; if `None` then just takes the parent of q
- `prefix` – (default: `'T'`) a label for the basis elements

EXAMPLES:

We define the Temperley–Lieb algebra of rank 2 with parameter x over $\mathbb{Z}$:

```
sage: R.<x> = ZZ[]
sage: T = TemperleyLiebAlgebra(2, x, R); T
Temperley–Lieb Algebra of rank 2 with parameter x
over Univariate Polynomial Ring in x over Integer Ring
sage: T.basis()
Lazy family (Term map from Temperley Lieb diagrams of order 2
to Temperley–Lieb Algebra of rank 2 with parameter x over
Univariate Polynomial Ring in x over Integer
Ring(i))_{i in Temperley Lieb diagrams of order 2}
sage: T.basis().keys()
Temperley Lieb diagrams of order 2
sage: T.basis().keys() ([[ -1, 1], [2, -2]])
{{ -2, 2}, { -1, 1}}
sage: b = T.basis().list(); b
[T{{ -2, -1}, {1, 2}}, T{{ -2, 2}, { -1, 1}}]
sage: b[0]
T{{ -2, -1}, {1, 2}}
sage: b[0]^2 == x*b[0]
True
sage: b[0]^5 == x^4*b[0]
True
```

```
>>> from sage.all import *
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> T = TemperleyLiebAlgebra(Integer(2), x, R); T
Temperley–Lieb Algebra of rank 2 with parameter x
over Univariate Polynomial Ring in x over Integer Ring
>>> T.basis()
Lazy family (Term map from Temperley Lieb diagrams of order 2
to Temperley–Lieb Algebra of rank 2 with parameter x over
```

(continues on next page)

(continued from previous page)

```

Univariate Polynomial Ring in x over Integer
Ring(i))_{i in Temperley Lieb diagrams of order 2}
>>> T.basis().keys()
Temperley Lieb diagrams of order 2
>>> T.basis().keys() ([[Integer(1), Integer(1)], [Integer(2), -Integer(2)]])
{{-2, 2}, {-1, 1}}
>>> b = T.basis().list(); b
[T{{-2, -1}, {1, 2}}, T{{-2, 2}, {-1, 1}}]
>>> b[Integer(0)]
T{{-2, -1}, {1, 2}}
>>> b[Integer(0)]**Integer(2) == x*b[Integer(0)]
True
>>> b[Integer(0)]**Integer(5) == x**Integer(4)*b[Integer(0)]
True

```

The Temperley-Lieb algebra is a cellular algebra, and we verify that the dimensions of the simple modules at $q = 0$ is given by [OEIS sequence A050166](#):

```

sage: for k in range(1,5):
.....:     TL = TemperleyLiebAlgebra(2*k, 0, QQ)
.....:     print("".join("{:3}".format(TL.cell_module(la).simple_module().
↳dimension())
.....:         for la in reversed(TL.cell_poset()) if la != 0))
1
1 2
1 4 5
1 6 14 14
sage: for k in range(1,4):
.....:     TL = TemperleyLiebAlgebra(2*k+1, 0, QQ)
.....:     print("".join("{:3}".format(TL.cell_module(la).simple_module().
↳dimension())
.....:         for la in reversed(TL.cell_poset()) if la != 0))
1 2
1 4 5
1 6 14 14

```

```

>>> from sage.all import *
>>> for k in range(Integer(1),Integer(5)):
...     TL = TemperleyLiebAlgebra(Integer(2)*k, Integer(0), QQ)
...     print("".join("{:3}".format(TL.cell_module(la).simple_module().
↳dimension())
...         for la in reversed(TL.cell_poset()) if la != Integer(0)))
1
1 2
1 4 5
1 6 14 14
>>> for k in range(Integer(1),Integer(4)):
...     TL = TemperleyLiebAlgebra(Integer(2)*k+Integer(1), Integer(0), QQ)
...     print("".join("{:3}".format(TL.cell_module(la).simple_module().
↳dimension())
...         for la in reversed(TL.cell_poset()) if la != Integer(0)))
1 2

```

(continues on next page)

(continued from previous page)

```

1  4  5
1  6 14 14

```

Additional examples when the Temperley-Lieb algebra is not semisimple:

```

sage: TL = TemperleyLiebAlgebra(8, -1, QQ)
sage: for la in TL.cell_poset():
....:     CM = TL.cell_module(la)
....:     if not CM.nonzero_bilinear_form():
....:         continue
....:     print(la, CM.dimension(), CM.simple_module().dimension())
....:
0 14 1
2 28 28
4 20 13
6 7 7
8 1 1
sage: for k in range(1,5):
....:     TL = TemperleyLiebAlgebra(2*k, -1, QQ)
....:     print("".join("{:3}".format(TL.cell_module(la).simple_module().
->dimension())
....:         for la in reversed(TL.cell_poset())
....:         if TL.cell_module(la).nonzero_bilinear_form()))
1  1
1  3  1
1  4  9  1
1  7 13 28 1
sage: C5.<z5> = CyclotomicField(5)
sage: for k in range(1,5):
....:     TL = TemperleyLiebAlgebra(2*k, z5+~z5, C5)
....:     print("".join("{:3}".format(TL.cell_module(la).simple_module().
->dimension())
....:         for la in reversed(TL.cell_poset())
....:         if TL.cell_module(la).nonzero_bilinear_form()))
1  1
1  3  2
1  5  8  5
1  7 20 21 13

```

```

>>> from sage.all import *
>>> TL = TemperleyLiebAlgebra(Integer(8), -Integer(1), QQ)
>>> for la in TL.cell_poset():
...     CM = TL.cell_module(la)
...     if not CM.nonzero_bilinear_form():
...         continue
...     print(la, CM.dimension(), CM.simple_module().dimension())
....:
0 14 1
2 28 28
4 20 13
6 7 7
8 1 1

```

(continues on next page)

(continued from previous page)

```

>>> for k in range(Integer(1), Integer(5)):
...     TL = TemperleyLiebAlgebra(Integer(2)*k, -Integer(1), QQ)
...     print("".join("{:3}".format(TL.cell_module(la).simple_module().
↪dimension()))
...         for la in reversed(TL.cell_poset())
...         if TL.cell_module(la).nonzero_bilinear_form()))
1  1
1  3  1
1  4  9  1
1  7 13 28  1
>>> C5 = CyclotomicField(Integer(5), names=('z5',)); (z5,) = C5._first_ngens(1)
>>> for k in range(Integer(1), Integer(5)):
...     TL = TemperleyLiebAlgebra(Integer(2)*k, z5+~z5, C5)
...     print("".join("{:3}".format(TL.cell_module(la).simple_module().
↪dimension()))
...         for la in reversed(TL.cell_poset())
...         if TL.cell_module(la).nonzero_bilinear_form()))
1  1
1  3  2
1  5  8  5
1  7 20 21 13

```

cell_module_indices (*la*)

Return the indices of the cell module of *self* indexed by *la*.

This is the finite set $M(\lambda)$.

EXAMPLES:

```

sage: R.<q> = QQ[]
sage: TL = TemperleyLiebAlgebra(8, q, R)
sage: TL.cell_module_indices(4)
Half Temperley-Lieb diagrams of order 8 with 4 defects

```

```

>>> from sage.all import *
>>> R = QQ['q']; (q,) = R._first_ngens(1)
>>> TL = TemperleyLiebAlgebra(Integer(8), q, R)
>>> TL.cell_module_indices(Integer(4))
Half Temperley-Lieb diagrams of order 8 with 4 defects

```

cell_poset ()

Return the cell poset of *self*.

EXAMPLES:

```

sage: R.<q> = QQ[]
sage: TL = TemperleyLiebAlgebra(7, q, R)
sage: TL.cell_poset().cover_relations()
[[1, 3], [3, 5], [5, 7]]

sage: TL = TemperleyLiebAlgebra(8, q, R)
sage: TL.cell_poset().cover_relations()
[[0, 2], [2, 4], [4, 6], [6, 8]]

```

```

>>> from sage.all import *
>>> R = QQ['q']; (q,) = R._first_ngens(1)
>>> TL = TemperleyLiebAlgebra(Integer(7), q, R)
>>> TL.cell_poset().cover_relations()
[[1, 3], [3, 5], [5, 7]]

>>> TL = TemperleyLiebAlgebra(Integer(8), q, R)
>>> TL.cell_poset().cover_relations()
[[0, 2], [2, 4], [4, 6], [6, 8]]

```

cellular_involution(*x*)

Return the cellular involution of *x* in self.

EXAMPLES:

```

sage: TL = TemperleyLiebAlgebra(4, QQ.zero(), QQ)
sage: ascii_art(TL.an_element())
      o o o o      o o o o      o o o o
      | '- ' |      | '- ' |
2* '- ' '- ' + 2* '-----' + 3* '---. |
      .-. .-.      .-. .-.      .-. | |
      o o o o      o o o o      o o o o
sage: ascii_art(TL.cellular_involution(TL.an_element()))
      o o o o      o o o o      o o o o
      | '- ' '- '      '- ' | |
2* '- ' '- ' + 2* .-----. + 3* .---' |
      .-. .-.      | .-. |      | .-. |
      o o o o      o o o o      o o o o

```

```

>>> from sage.all import *
>>> TL = TemperleyLiebAlgebra(Integer(4), QQ.zero(), QQ)
>>> ascii_art(TL.an_element())
      o o o o      o o o o      o o o o
      | '- ' |      | '- ' |
2* '- ' '- ' + 2* '-----' + 3* '---. |
      .-. .-.      .-. .-.      .-. | |
      o o o o      o o o o      o o o o
>>> ascii_art(TL.cellular_involution(TL.an_element()))
      o o o o      o o o o      o o o o
      | '- ' '- '      '- ' | |
2* '- ' '- ' + 2* .-----. + 3* .---' |
      .-. .-.      | .-. |      | .-. |
      o o o o      o o o o      o o o o

```

class sage.combinat.diagram_algebras.**TemperleyLiebDiagram**(*parent, d, check=True*)

Bases: [AbstractPartitionDiagram](#)

The element class for a Temperley-Lieb diagram.

A Temperley-Lieb diagram for an integer k is a partition of the set $\{1, \dots, k, -1, \dots, -k\}$ so that the blocks are all of size 2 and the diagram is planar.

EXAMPLES:

```
sage: from sage.combinat.diagram_algebras import TemperleyLiebDiagrams
sage: TemperleyLiebDiagrams(2)
Temperley Lieb diagrams of order 2
sage: TemperleyLiebDiagrams(2).list()
[{{-2, -1}, {1, 2}}, {{-2, 2}, {-1, 1}}]
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram_algebras import TemperleyLiebDiagrams
>>> TemperleyLiebDiagrams(Integer(2))
Temperley Lieb diagrams of order 2
>>> TemperleyLiebDiagrams(Integer(2)).list()
[{{-2, -1}, {1, 2}}, {{-2, 2}, {-1, 1}}]
```

check()

Check the validity of the input for `self`.

class `sage.combinat.diagram_algebras.TemperleyLiebDiagrams` (*order*, *category=None*)

Bases: `AbstractPartitionDiagrams`

All Temperley-Lieb diagrams of integer or integer +1/2 order.

For more information on Temperley-Lieb diagrams, see [TemperleyLiebAlgebra](#).

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: td = da.TemperleyLiebDiagrams(3); td
Temperley Lieb diagrams of order 3
sage: td.list()
[{{-3, 3}, {-2, -1}, {1, 2}},
 {{-3, 1}, {-2, -1}, {2, 3}},
 {{-3, -2}, {-1, 1}, {2, 3}},
 {{-3, -2}, {-1, 3}, {1, 2}},
 {{-3, 3}, {-2, 2}, {-1, 1}}]

sage: td = da.TemperleyLiebDiagrams(5/2); td
Temperley Lieb diagrams of order 5/2
sage: td.list()
[{{-3, 3}, {-2, -1}, {1, 2}}, {{-3, 3}, {-2, 2}, {-1, 1}}]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> td = da.TemperleyLiebDiagrams(Integer(3)); td
Temperley Lieb diagrams of order 3
>>> td.list()
[{{-3, 3}, {-2, -1}, {1, 2}},
 {{-3, 1}, {-2, -1}, {2, 3}},
 {{-3, -2}, {-1, 1}, {2, 3}},
 {{-3, -2}, {-1, 3}, {1, 2}},
 {{-3, 3}, {-2, 2}, {-1, 1}}]

>>> td = da.TemperleyLiebDiagrams(Integer(5)/Integer(2)); td
Temperley Lieb diagrams of order 5/2
```

(continues on next page)

(continued from previous page)

```
>>> td.list()
[{{-3, 3}, {-2, -1}, {1, 2}}, {{-3, 3}, {-2, 2}, {-1, 1}}]
```

Element

alias of *TemperleyLiebDiagram*

cardinality()

Return the cardinality of self.

The number of Temperley–Lieb diagrams of integer order k is the k -th Catalan number.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: td = da.TemperleyLiebDiagrams(3)
sage: td.cardinality()
5
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> td = da.TemperleyLiebDiagrams(Integer(3))
>>> td.cardinality()
5
```

class sage.combinat.diagram_algebras.**UnitDiagramMixin**

Bases: object

Mixin class for diagram algebras that have the unit indexed by the *identity_set_partition()*.

one_basis()

The following constructs the identity element of self.

It is not called directly; instead one should use `DA.one()` if `DA` is a defined diagram algebra.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: P = PartitionAlgebra(2, x, R)
sage: P.one_basis()
{{-2, 2}, {-1, 1}}
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> P = PartitionAlgebra(Integer(2), x, R)
>>> P.one_basis()
{{-2, 2}, {-1, 1}}
```

sage.combinat.diagram_algebras.brauer_diagrams(k)

Return a generator of all Brauer diagrams of order k .

A Brauer diagram of order k is a partition diagram of order k with block size 2.

INPUT:

- k – the order of the Brauer diagrams

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: [SetPartition(p) for p in da.brauer_diagrams(2)]
[{{-2, -1}, {1, 2}}, {{-2, 1}, {-1, 2}}, {{-2, 2}, {-1, 1}}]
sage: [SetPartition(p) for p in da.brauer_diagrams(5/2)]
[{{-3, 3}, {-2, -1}, {1, 2}},
 {{-3, 3}, {-2, 1}, {-1, 2}},
 {{-3, 3}, {-2, 2}, {-1, 1}}]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> [SetPartition(p) for p in da.brauer_diagrams(Integer(2))]
[{{-2, -1}, {1, 2}}, {{-2, 1}, {-1, 2}}, {{-2, 2}, {-1, 1}}]
>>> [SetPartition(p) for p in da.brauer_diagrams(Integer(5)/Integer(2))]
[{{-3, 3}, {-2, -1}, {1, 2}},
 {{-3, 3}, {-2, 1}, {-1, 2}},
 {{-3, 3}, {-2, 2}, {-1, 1}}]
```

`sage.combinat.diagram_algebras.diagram_latex`(*diagram*, *fill=False*, *edge_options=None*, *edge_additions=None*)

Return latex code for the diagram *diagram* using tikz.

EXAMPLES:

```
sage: from sage.combinat.diagram_algebras import PartitionDiagrams, diagram_latex
sage: P = PartitionDiagrams(2)
sage: D = P([[1,2],[-2,-1]])
sage: print(diagram_latex(D)) # indirect doctest
\begin{tikzpicture}[scale = 0.5,thick, baseline={ (0,-1ex/2) }]
\tikzstyle{vertex} = [shape = circle, minimum size = 7pt, inner sep = 1pt]
\node[vertex] (G--2) at (1.5, -1) [shape = circle, draw] {};
\node[vertex] (G--1) at (0.0, -1) [shape = circle, draw] {};
\node[vertex] (G-1) at (0.0, 1) [shape = circle, draw] {};
\node[vertex] (G-2) at (1.5, 1) [shape = circle, draw] {};
\draw[] (G--2) .. controls +(-0.5, 0.5) and +(0.5, 0.5) .. (G--1);
\draw[] (G-1) .. controls +(0.5, -0.5) and +(-0.5, -0.5) .. (G-2);
\end{tikzpicture}
```

```
>>> from sage.all import *
>>> from sage.combinat.diagram_algebras import PartitionDiagrams, diagram_latex
>>> P = PartitionDiagrams(Integer(2))
>>> D = P([[Integer(1),Integer(2)],[-Integer(2),-Integer(1)]])
>>> print(diagram_latex(D)) # indirect doctest
\begin{tikzpicture}[scale = 0.5,thick, baseline={ (0,-1ex/2) }]
\tikzstyle{vertex} = [shape = circle, minimum size = 7pt, inner sep = 1pt]
\node[vertex] (G--2) at (1.5, -1) [shape = circle, draw] {};
\node[vertex] (G--1) at (0.0, -1) [shape = circle, draw] {};
\node[vertex] (G-1) at (0.0, 1) [shape = circle, draw] {};
\node[vertex] (G-2) at (1.5, 1) [shape = circle, draw] {};
\draw[] (G--2) .. controls +(-0.5, 0.5) and +(0.5, 0.5) .. (G--1);
\draw[] (G-1) .. controls +(0.5, -0.5) and +(-0.5, -0.5) .. (G-2);
\end{tikzpicture}
```

`sage.combinat.diagram_algebras.ideal_diagrams`(*k*)

Return a generator of all “ideal” diagrams of order k .

An ideal diagram of order k is a partition diagram of order k with propagating number less than k .

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: all_diagrams = da.partition_diagrams(2)
sage: [SetPartition(p) for p in all_diagrams if p not in da.ideal_diagrams(2)]
[{{-2, 1}, {-1, 2}}, {{-2, 2}, {-1, 1}}]

sage: all_diagrams = da.partition_diagrams(3/2)
sage: [SetPartition(p) for p in all_diagrams if p not in da.ideal_diagrams(3/2)]
[{{-2, 2}, {-1, 1}}]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> all_diagrams = da.partition_diagrams(Integer(2))
>>> [SetPartition(p) for p in all_diagrams if p not in da.ideal_
↳ diagrams(Integer(2))]
[{{-2, 1}, {-1, 2}}, {{-2, 2}, {-1, 1}}]

>>> all_diagrams = da.partition_diagrams(Integer(3)/Integer(2))
>>> [SetPartition(p) for p in all_diagrams if p not in da.ideal_
↳ diagrams(Integer(3)/Integer(2))]
[{{-2, 2}, {-1, 1}}]
```

`sage.combinat.diagram_algebras.identity_set_partition( $k$ )`

Return the identity set partition $\{\{1, -1\}, \dots, \{k, -k\}\}$.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: SetPartition(da.identity_set_partition(2))
{{-2, 2}, {-1, 1}}
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> SetPartition(da.identity_set_partition(Integer(2)))
{{-2, 2}, {-1, 1}}
```

`sage.combinat.diagram_algebras.is_planar( $sp$ )`

Return True if the diagram corresponding to the set partition sp is planar; otherwise, return False.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: da.is_planar( da.to_set_partition([[1, -2], [2, -1]]))
False
sage: da.is_planar( da.to_set_partition([[1, -1], [2, -2]]))
True
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
```

(continues on next page)

(continued from previous page)

```

>>> da.is_planar( da.to_set_partition([[Integer(1),-Integer(2)],[Integer(2),-
↪Integer(1)]]))
False
>>> da.is_planar( da.to_set_partition([[Integer(1),-Integer(1)],[Integer(2),-
↪Integer(2)]]))
True

```

sage.combinat.diagram_algebras.**pair_to_graph**(*sp1*, *sp2*)

Return a graph consisting of the disjoint union of the graphs of set partitions *sp1* and *sp2* along with edges joining the bottom row (negative numbers) of *sp1* to the top row (positive numbers) of *sp2*.

The vertices of the graph *sp1* appear in the result as pairs (*k*, 1), whereas the vertices of the graph *sp2* appear as pairs (*k*, 2).

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: sp1 = da.to_set_partition([[1,-2],[2,-1]])
sage: sp2 = da.to_set_partition([[1,-2],[2,-1]])
sage: g = da.pair_to_graph( sp1, sp2 ); g
Graph on 8 vertices

sage: g.vertices(sort=True)
[(-2, 1), (-2, 2), (-1, 1), (-1, 2), (1, 1), (1, 2), (2, 1), (2, 2)]
sage: g.edges(sort=True)
[((-2, 1), (1, 1), None), ((-2, 1), (2, 2), None),
((-2, 2), (1, 2), None), ((-1, 1), (1, 2), None),
((-1, 1), (2, 1), None), ((-1, 2), (2, 2), None)]

```

```

>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> sp1 = da.to_set_partition([[Integer(1),-Integer(2)],[Integer(2),-Integer(1)]])
>>> sp2 = da.to_set_partition([[Integer(1),-Integer(2)],[Integer(2),-Integer(1)]])
>>> g = da.pair_to_graph( sp1, sp2 ); g
Graph on 8 vertices

>>> g.vertices(sort=True)
[(-2, 1), (-2, 2), (-1, 1), (-1, 2), (1, 1), (1, 2), (2, 1), (2, 2)]
>>> g.edges(sort=True)
[((-2, 1), (1, 1), None), ((-2, 1), (2, 2), None),
((-2, 2), (1, 2), None), ((-1, 1), (1, 2), None),
((-1, 1), (2, 1), None), ((-1, 2), (2, 2), None)]

```

Another example which used to be wrong until [Issue #15958](#):

```

sage: sp3 = da.to_set_partition([[1, -1], [2], [-2]])
sage: sp4 = da.to_set_partition([[1], [-1], [2], [-2]])
sage: g = da.pair_to_graph( sp3, sp4 ); g
Graph on 8 vertices

sage: g.vertices(sort=True)
[(-2, 1), (-2, 2), (-1, 1), (-1, 2), (1, 1), (1, 2), (2, 1), (2, 2)]
sage: g.edges(sort=True)

```

(continues on next page)

(continued from previous page)

```
[((-2, 1), (2, 2), None), ((-1, 1), (1, 1), None),
 ((-1, 1), (1, 2), None)]
```

```
>>> from sage.all import *
>>> sp3 = da.to_set_partition([[Integer(1), -Integer(1)], [Integer(2)], [-
↳ Integer(2)]]])
>>> sp4 = da.to_set_partition([[Integer(1)], [-Integer(1)], [Integer(2)], [-
↳ Integer(2)]]])
>>> g = da.pair_to_graph( sp3, sp4 ); g
Graph on 8 vertices

>>> g.vertices(sort=True)
[(-2, 1), (-2, 2), (-1, 1), (-1, 2), (1, 1), (1, 2), (2, 1), (2, 2)]
>>> g.edges(sort=True)
[((-2, 1), (2, 2), None), ((-1, 1), (1, 1), None),
 ((-1, 1), (1, 2), None)]
```

`sage.combinat.diagram_algebras.partition_diagrams(k)`

Return a generator of all partition diagrams of order k .

A partition diagram of order $k \in \mathbf{Z}$ is a set partition of $\{1, \dots, k, -1, \dots, -k\}$. If we have $k - 1/2 \in \mathbf{ZZ}$, then a partition diagram of order $k \in 1/2\mathbf{Z}$ is a set partition of $\{1, \dots, k + 1/2, -1, \dots, -(k + 1/2)\}$ with $k + 1/2$ and $-(k + 1/2)$ in the same block. See [HR2005].

INPUT:

- k – the order of the partition diagrams

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: [SetPartition(p) for p in da.partition_diagrams(2)]
[{{-2, -1, 1, 2}},
 {{-2, 1, 2}, {-1}},
 {{-2}, {-1, 1, 2}},
 {{-2, -1}, {1, 2}},
 {{-2}, {-1}, {1, 2}},
 {{-2, -1, 1}, {2}},
 {{-2, 1}, {-1, 2}},
 {{-2, 1}, {-1}, {2}},
 {{-2, 2}, {-1, 1}},
 {{-2, -1, 2}, {1}},
 {{-2, 2}, {-1}, {1}},
 {{-2}, {-1, 1}, {2}},
 {{-2}, {-1, 2}, {1}},
 {{-2, -1}, {1}, {2}},
 {{-2}, {-1}, {1}, {2}}]
sage: [SetPartition(p) for p in da.partition_diagrams(3/2)]
[{{-2, -1, 1, 2}},
 {{-2, 1, 2}, {-1}},
 {{-2, 2}, {-1, 1}},
 {{-2, -1, 2}, {1}},
 {{-2, 2}, {-1}, {1}}]
```



```

>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> [SetPartition(p) for p in da.partition_diagrams(Integer(2))]
[{{-2, -1, 1, 2}},
 {{-2, 1, 2}, {-1}},
 {{-2}, {-1, 1, 2}},
 {{-2, -1}, {1, 2}},
 {{-2}, {-1}, {1, 2}},
 {{-2, -1, 1}, {2}},
 {{-2, 1}, {-1, 2}},
 {{-2, 1}, {-1}, {2}},
 {{-2, 2}, {-1, 1}},
 {{-2, -1, 2}, {1}},
 {{-2, 2}, {-1}, {1}},
 {{-2}, {-1, 1}, {2}},
 {{-2}, {-1, 2}, {1}},
 {{-2, -1}, {1}, {2}},
 {{-2}, {-1}, {1}, {2}}]
>>> [SetPartition(p) for p in da.partition_diagrams(Integer(3)/Integer(2))]
[{{-2, -1, 1, 2}},
 {{-2, 1, 2}, {-1}},
 {{-2, 2}, {-1, 1}},
 {{-2, -1, 2}, {1}},
 {{-2, 2}, {-1}, {1}}]

```

`sage.combinat.diagram_algebras.planar_diagrams(k)`

Return a generator of all planar diagrams of order *k*.

A planar diagram of order *k* is a partition diagram of order *k* that has no crossings.

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: all_diagrams = [SetPartition(p) for p in da.partition_diagrams(2)]
sage: da2 = [SetPartition(p) for p in da.planar_diagrams(2)]
sage: [p for p in all_diagrams if p not in da2]
[{{-2, 1}, {-1, 2}}]
sage: all_diagrams = [SetPartition(p) for p in da.partition_diagrams(5/2)]
sage: da5o2 = [SetPartition(p) for p in da.planar_diagrams(5/2)]
sage: [p for p in all_diagrams if p not in da5o2]
[{{-3, -1, 3}, {-2, 1, 2}},
 {{-3, -2, 1, 3}, {-1, 2}},
 {{-3, -1, 1, 3}, {-2, 2}},
 {{-3, 1, 3}, {-2, -1, 2}},
 {{-3, 1, 3}, {-2, 2}, {-1}},
 {{-3, 1, 3}, {-2}, {-1, 2}},
 {{-3, -1, 2, 3}, {-2, 1}},
 {{-3, 3}, {-2, 1}, {-1, 2}},
 {{-3, -1, 3}, {-2, 1}, {2}},
 {{-3, -1, 3}, {-2, 2}, {1}}]

```

```

>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da

```

(continues on next page)

(continued from previous page)

```

>>> all_diagrams = [SetPartition(p) for p in da.partition_diagrams(Integer(2))]
>>> da2 = [SetPartition(p) for p in da.planar_diagrams(Integer(2))]
>>> [p for p in all_diagrams if p not in da2]
[{{-2, 1}, {-1, 2}}]
>>> all_diagrams = [SetPartition(p) for p in da.partition_diagrams(Integer(5)/
↳ Integer(2))]
>>> da5o2 = [SetPartition(p) for p in da.planar_diagrams(Integer(5)/Integer(2))]
>>> [p for p in all_diagrams if p not in da5o2]
[{{-3, -1, 3}, {-2, 1, 2}},
 {{-3, -2, 1, 3}, {-1, 2}},
 {{-3, -1, 1, 3}, {-2, 2}},
 {{-3, 1, 3}, {-2, -1, 2}},
 {{-3, 1, 3}, {-2, 2}, {-1}},
 {{-3, 1, 3}, {-2}, {-1, 2}},
 {{-3, -1, 2, 3}, {-2, 1}},
 {{-3, 3}, {-2, 1}, {-1, 2}},
 {{-3, -1, 3}, {-2, 1}, {2}},
 {{-3, -1, 3}, {-2, 2}, {1}}]

```

sage.combinat.diagram_algebras.**planar_partitions_rec**(X)

Iterate over all planar set partitions of X by using a recursive algorithm.

ALGORITHM:

To construct the set partition $\rho = \{\rho_1, \dots, \rho_k\}$ of $[n]$, we remove the part of the set partition containing the last element of X, which, we consider to be $\rho_k = \{i_1, \dots, i_m\}$ without loss of generality. The remaining parts come from the planar set partitions of $\{1, \dots, i_1 - 1\}, \{i_1 + 1, \dots, i_2 - 1\}, \dots, \{i_m + 1, \dots, n\}$.

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: list(da.planar_partitions_rec([1, 2, 3]))
[([1, 2], [3]), ([1], [2], [3]), ([2], [1, 3]), ([1], [2, 3]), ([1, 2, 3],)]

```

```

>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> list(da.planar_partitions_rec([Integer(1), Integer(2), Integer(3)]))
[([1, 2], [3]), ([1], [2], [3]), ([2], [1, 3]), ([1], [2, 3]), ([1, 2, 3],)]

```

sage.combinat.diagram_algebras.**propagating_number**(sp)

Return the propagating number of the set partition sp.

The propagating number is the number of blocks with both a positive and negative number.

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: sp1 = da.to_set_partition([[1, -2], [2, -1]])
sage: sp2 = da.to_set_partition([[1, 2], [-2, -1]])
sage: da.propagating_number(sp1)
2
sage: da.propagating_number(sp2)
0

```

```

>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> sp1 = da.to_set_partition([[Integer(1), -Integer(2)], [Integer(2), -Integer(1)]])
>>> sp2 = da.to_set_partition([[Integer(1), Integer(2)], [-Integer(2), -Integer(1)]])
>>> da.propagating_number(sp1)
2
>>> da.propagating_number(sp2)
0

```

`sage.combinat.diagram_algebras.temperley_lieb_diagrams(k)`

Return a generator of all Temperley–Lieb diagrams of order k .

A Temperley–Lieb diagram of order k is a partition diagram of order k with block size 2 and is planar.

INPUT:

- k – the order of the Temperley–Lieb diagrams

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: [SetPartition(p) for p in da.temperley_lieb_diagrams(2)]
[{{-2, -1}, {1, 2}}, {{-2, 2}, {-1, 1}}]
sage: [SetPartition(p) for p in da.temperley_lieb_diagrams(5/2)]
[{{-3, 3}, {-2, -1}, {1, 2}}, {{-3, 3}, {-2, 2}, {-1, 1}}]

```

```

>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> [SetPartition(p) for p in da.temperley_lieb_diagrams(Integer(2))]
[{{-2, -1}, {1, 2}}, {{-2, 2}, {-1, 1}}]
>>> [SetPartition(p) for p in da.temperley_lieb_diagrams(Integer(5)/Integer(2))]
[{{-3, 3}, {-2, -1}, {1, 2}}, {{-3, 3}, {-2, 2}, {-1, 1}}]

```

`sage.combinat.diagram_algebras.to_Brauer_partition(l, k=None)`

Same as `to_set_partition()` but assumes omitted elements are connected straight through.

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: f = lambda sp: SetPartition(da.to_Brauer_partition(sp))
sage: f([[1, 2], [-1, -2]]) == SetPartition([[1, 2], [-1, -2]])
True
sage: f([[1, 3], [-1, -3]]) == SetPartition([[1, 3], [-3, -1], [2, -2]])
True
sage: f([[1, -4], [-3, -1], [3, 4]]) == SetPartition([[-3, -1], [2, -2], [1, -4], [3, 4]])
True
sage: p = SetPartition([[1, 2], [-1, -2], [3, -3], [4, -4]])
sage: SetPartition(da.to_Brauer_partition([[1, 2], [-1, -2]], k=4)) == p
True

```

```

>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> f = lambda sp: SetPartition(da.to_Brauer_partition(sp))
>>> f([[Integer(1), Integer(2)], [-Integer(1), -Integer(2)]]) == _

```

(continues on next page)

(continued from previous page)

```

↪SetPartition([[Integer(1), Integer(2)], [-Integer(1), -Integer(2)]])
True
>>> f([[Integer(1), Integer(3)], [-Integer(1), -Integer(3)]]) == ↪
↪SetPartition([[Integer(1), Integer(3)], [-Integer(3), -Integer(1)], [Integer(2), -
↪Integer(2)]])
True
>>> f([[Integer(1), -Integer(4)], [-Integer(3), -Integer(1)], [Integer(3),
↪Integer(4)]]) == SetPartition([[-Integer(3), -Integer(1)], [Integer(2), -
↪Integer(2)], [Integer(1), -Integer(4)], [Integer(3), Integer(4)]])
True
>>> p = SetPartition([[Integer(1), Integer(2)], [-Integer(1), -Integer(2)],
↪[Integer(3), -Integer(3)], [Integer(4), -Integer(4)]])
>>> SetPartition(da.to_Brauer_partition([[Integer(1), Integer(2)], [-Integer(1), -
↪Integer(2)]], k=Integer(4))) == p
True

```

sage.combinat.diagram_algebras.to_graph(sp)

Return a graph representing the set partition sp.

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: g = da.to_graph( da.to_set_partition([[1, -2], [2, -1]])); g
Graph on 4 vertices

sage: g.vertices(sort=True)
[-2, -1, 1, 2]
sage: g.edges(sort=True)
[(-2, 1, None), (-1, 2, None)]

```

```

>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> g = da.to_graph( da.to_set_partition([[Integer(1), -Integer(2)], [Integer(2), -
↪Integer(1)]])]); g
Graph on 4 vertices

>>> g.vertices(sort=True)
[-2, -1, 1, 2]
>>> g.edges(sort=True)
[(-2, 1, None), (-1, 2, None)]

```

sage.combinat.diagram_algebras.to_set_partition(l, k=None)

Convert input to a set partition of $\{1, \dots, k, -1, \dots, -k\}$.

Convert a list of a list of numbers to a set partitions. Each list of numbers in the outer list specifies the numbers contained in one of the blocks in the set partition.

If k is specified, then the set partition will be a set partition of $\{1, \dots, k, -1, \dots, -k\}$. Otherwise, k will default to the minimum number needed to contain all of the specified numbers.

INPUT:

- l – list of lists of integers
- k – integer (default: None)

OUTPUT: list of sets

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: f = lambda sp: SetPartition(da.to_set_partition(sp))
sage: f([[1,-1],[2,-2]]) == SetPartition(da.identity_set_partition(2))
True
sage: da.to_set_partition([[1]])
[{1}, {-1}]
sage: da.to_set_partition([[1,-1],[-2,3]], 9/2)
[{-1, 1}, {-2, 3}, {2}, {-4, 4}, {-5, 5}, {-3}]
```

```
>>> from sage.all import *
>>> import sage.combinat.diagram_algebras as da
>>> f = lambda sp: SetPartition(da.to_set_partition(sp))
>>> f([[Integer(1), -Integer(1)], [Integer(2), -Integer(2)]]) == SetPartition(da.
↳ identity_set_partition(Integer(2)))
True
>>> da.to_set_partition([[Integer(1)]])
[{1}, {-1}]
>>> da.to_set_partition([[Integer(1), -Integer(1)], [-Integer(2), Integer(3)]],
↳ Integer(9)/Integer(2))
[{-1, 1}, {-2, 3}, {2}, {-4, 4}, {-5, 5}, {-3}]
```

5.4 Clifford Algebras

AUTHORS:

- Travis Scrimshaw (2013-09-06): Initial version
- Trevor K. Karn (2022-07-27): Rewrite basis indexing using FrozenBitset

class sage.algebras.clifford_algebra.**CliffordAlgebra** (Q , *names*, *category=None*)

Bases: [CombinatorialFreeModule](#)

The Clifford algebra of a quadratic form.

Let $Q : V \rightarrow \mathbf{k}$ denote a quadratic form on a vector space V over a field $\mathbf{k}$. The Clifford algebra $Cl(V, Q)$ is defined as $T(V)/I_Q$ where $T(V)$ is the tensor algebra of V and I_Q is the two-sided ideal generated by all elements of the form $v \otimes v - Q(v)$ for all $v \in V$.

We abuse notation to denote the projection of a pure tensor $x_1 \otimes x_2 \otimes \cdots \otimes x_m \in T(V)$ onto $T(V)/I_Q = Cl(V, Q)$ by $x_1 \wedge x_2 \wedge \cdots \wedge x_m$. This is motivated by the fact that $Cl(V, Q)$ is the exterior algebra $\wedge V$ when $Q = 0$ (one can also think of a Clifford algebra as a quantization of the exterior algebra). See [ExteriorAlgebra](#) for the concept of an exterior algebra.

From the definition, a basis of $Cl(V, Q)$ is given by monomials of the form

$$\{e_{i_1} \wedge \cdots \wedge e_{i_k} \mid 1 \leq i_1 < \cdots < i_k \leq n\},$$

where $n = \dim(V)$ and where $\{e_1, e_2, \dots, e_n\}$ is any fixed basis of V . Hence

$$\dim(Cl(V, Q)) = \sum_{k=0}^n \binom{n}{k} = 2^n.$$

Note

The algebra $Cl(V, Q)$ is a $\mathbf{Z}/2\mathbf{Z}$ -graded algebra, but not (in general) $\mathbf{Z}$ -graded (in a reasonable way).

This construction satisfies the following universal property. Let $i : V \rightarrow Cl(V, Q)$ denote the natural inclusion (which is an embedding). Then for every associative $\mathbf{k}$ -algebra A and any $\mathbf{k}$ -linear map $j : V \rightarrow A$ satisfying

$$j(v)^2 = Q(v) \cdot 1_A$$

for all $v \in V$, there exists a unique $\mathbf{k}$ -algebra homomorphism $f : Cl(V, Q) \rightarrow A$ such that $f \circ i = j$. This property determines the Clifford algebra uniquely up to canonical isomorphism. The inclusion i is commonly used to identify V with a vector subspace of $Cl(V)$.

The Clifford algebra $Cl(V, Q)$ is a $\mathbf{Z}_2$ -graded algebra (where $\mathbf{Z}_2 = \mathbf{Z}/2\mathbf{Z}$); this grading is determined by placing all elements of V in degree 1. It is also an $\mathbf{N}$ -filtered algebra, with the filtration too being defined by placing all elements of V in degree 1. The `degree()` gives the $\mathbf{N}$ -filtration degree, and to get the super degree use instead `is_even_odd()`.

The Clifford algebra also can be considered as a covariant functor from the category of vector spaces equipped with quadratic forms to the category of algebras. In fact, if (V, Q) and (W, R) are two vector spaces endowed with quadratic forms, and if $g : W \rightarrow V$ is a linear map preserving the quadratic form, then we can define an algebra morphism $Cl(g) : Cl(W, R) \rightarrow Cl(V, Q)$ by requiring that it send every $w \in W$ to $g(w) \in V$. Since the quadratic form R on W is uniquely determined by the quadratic form Q on V (due to the assumption that g preserves the quadratic form), this fact can be rewritten as follows: If (V, Q) is a vector space with a quadratic form, and W is another vector space, and $\phi : W \rightarrow V$ is any linear map, then we obtain an algebra morphism $Cl(\phi) : Cl(W, \phi(Q)) \rightarrow Cl(V, Q)$ where $\phi(Q) = \phi^T \cdot Q \cdot \phi$ (we consider ϕ as a matrix) is the quadratic form Q pulled back to W . In fact, the map ϕ preserves the quadratic form because of

$$\phi(Q)(x) = x^T \cdot \phi^T \cdot Q \cdot \phi \cdot x = (\phi \cdot x)^T \cdot Q \cdot (\phi \cdot x) = Q(\phi(x)).$$

Hence we have $\phi(w)^2 = Q(\phi(w)) = \phi(Q)(w)$ for all $w \in W$.

REFERENCES:

- [Wikipedia article Clifford algebra](#)

INPUT:

- Q – a quadratic form
- `names` – (default: 'e') the generator names

EXAMPLES:

To create a Clifford algebra, all one needs to do is specify a quadratic form:

```
sage: Q = QuadraticForm(ZZ, 3, [1, 2, 3, 4, 5, 6])
sage: Cl = CliffordAlgebra(Q)
sage: Cl
The Clifford algebra of the Quadratic form in 3 variables
over Integer Ring with coefficients:
[ 1 2 3 ]
[ * 4 5 ]
[ * * 6 ]
```

```
>>> from sage.all import *
>>> Q = QuadraticForm(ZZ, Integer(3), [Integer(1), Integer(2), Integer(3),
```

(continues on next page)

(continued from previous page)

```

↪Integer(4),Integer(5),Integer(6)])
>>> Cl = CliffordAlgebra(Q)
>>> Cl
The Clifford algebra of the Quadratic form in 3 variables
over Integer Ring with coefficients:
[ 1 2 3 ]
[ * 4 5 ]
[ * * 6 ]

```

We can also explicitly name the generators. In this example, the Clifford algebra we construct is an exterior algebra (since we choose the quadratic form to be zero):

```

sage: Q = QuadraticForm(ZZ, 4, [0]*10)
sage: Cl.<a,b,c,d> = CliffordAlgebra(Q)
sage: a*d
a*d
sage: d*c*b*a + a + 4*b*c
a*b*c*d + 4*b*c + a

```

```

>>> from sage.all import *
>>> Q = QuadraticForm(ZZ, Integer(4), [Integer(0)]*Integer(10))
>>> Cl = CliffordAlgebra(Q, names=('a', 'b', 'c', 'd')); (a, b, c, d,) = Cl._
↪first_ngens(4)
>>> a*d
a*d
>>> d*c*b*a + a + Integer(4)*b*c
a*b*c*d + 4*b*c + a

```

Element

alias of CliffordAlgebraElement

algebra_generators()

Return the algebra generators of self.

EXAMPLES:

```

sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.algebra_generators()
Finite family {'x': x, 'y': y, 'z': z}

```

```

>>> from sage.all import *
>>> Q = QuadraticForm(ZZ, Integer(3), [Integer(1),Integer(2),Integer(3),
↪Integer(4),Integer(5),Integer(6)])
>>> Cl = CliffordAlgebra(Q, names=('x', 'y', 'z')); (x, y, z,) = Cl._first_
↪ngens(3)
>>> Cl.algebra_generators()
Finite family {'x': x, 'y': y, 'z': z}

```

center_basis()

Return a list of elements which correspond to a basis for the center of self.

This assumes that the ground ring can be used to compute the kernel of a matrix.

See also

`supercenter_basis()`, <http://math.stackexchange.com/questions/129183/center-of-clifford-algebra-depending-on-the-parity-of-dim-v>

Todo

Deprecate this in favor of a method called `center()` once subalgebras are properly implemented in Sage.

EXAMPLES:

```
sage: Q = QuadraticForm(QQ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Z = Cl.center_basis(); Z
(1, -2/5*x*y*z + x - 3/5*y + 2/5*z)
sage: all(z*b - b*z == 0 for z in Z for b in Cl.basis())
True

sage: Q = QuadraticForm(QQ, 3, [1,-2,-3, 4, 2, 1])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Z = Cl.center_basis(); Z
(1, -x*y*z + x + 3/2*y - z)
sage: all(z*b - b*z == 0 for z in Z for b in Cl.basis())
True

sage: Q = QuadraticForm(QQ, 2, [1,-2,-3])
sage: Cl.<x,y> = CliffordAlgebra(Q)
sage: Cl.center_basis()
(1,)

sage: Q = QuadraticForm(QQ, 2, [-1,1,-3])
sage: Cl.<x,y> = CliffordAlgebra(Q)
sage: Cl.center_basis()
(1,)
```

```
>>> from sage.all import *
>>> Q = QuadraticForm(QQ, Integer(3), [Integer(1),Integer(2),Integer(3),
↳Integer(4),Integer(5),Integer(6)])
>>> Cl = CliffordAlgebra(Q, names=('x', 'y', 'z',)); (x, y, z,) = Cl._first_
↳ngens(3)
>>> Z = Cl.center_basis(); Z
(1, -2/5*x*y*z + x - 3/5*y + 2/5*z)
>>> all(z*b - b*z == Integer(0) for z in Z for b in Cl.basis())
True

>>> Q = QuadraticForm(QQ, Integer(3), [Integer(1),-Integer(2),-Integer(3),
↳Integer(4), Integer(2), Integer(1)])
>>> Cl = CliffordAlgebra(Q, names=('x', 'y', 'z',)); (x, y, z,) = Cl._first_
↳ngens(3)
>>> Z = Cl.center_basis(); Z
(1, -x*y*z + x + 3/2*y - z)
```

(continues on next page)

(continued from previous page)

```

>>> all(z*b - b*z == Integer(0) for z in Z for b in Cl.basis())
True

>>> Q = QuadraticForm(QQ, Integer(2), [Integer(1), -Integer(2), -Integer(3)])
>>> Cl = CliffordAlgebra(Q, names=('x', 'y')); (x, y) = Cl._first_ngens(2)
>>> Cl.center_basis()
(1,)

>>> Q = QuadraticForm(QQ, Integer(2), [-Integer(1), Integer(1), -Integer(3)])
>>> Cl = CliffordAlgebra(Q, names=('x', 'y')); (x, y) = Cl._first_ngens(2)
>>> Cl.center_basis()
(1,)

```

A degenerate case:

```

sage: Q = QuadraticForm(QQ, 3, [4, 4, -4, 1, -2, 1])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.center_basis()
(1, x*y*z + x - 2*y - 2*z, x*y + x*z - 2*y*z)

```

```

>>> from sage.all import *
>>> Q = QuadraticForm(QQ, Integer(3), [Integer(4), Integer(4), -Integer(4),
↳ Integer(1), -Integer(2), Integer(1)])
>>> Cl = CliffordAlgebra(Q, names=('x', 'y', 'z')); (x, y, z) = Cl._first_
↳ ngens(3)
>>> Cl.center_basis()
(1, x*y*z + x - 2*y - 2*z, x*y + x*z - 2*y*z)

```

The most degenerate case (the exterior algebra):

```

sage: Q = QuadraticForm(QQ, 3)
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.center_basis()
(1, x*y, x*z, y*z, x*y*z)

```

```

>>> from sage.all import *
>>> Q = QuadraticForm(QQ, Integer(3))
>>> Cl = CliffordAlgebra(Q, names=('x', 'y', 'z')); (x, y, z) = Cl._first_
↳ ngens(3)
>>> Cl.center_basis()
(1, x*y, x*z, y*z, x*y*z)

```

degree_on_basis(*m*)

Return the degree of the monomial indexed by *m*.

We are considering the Clifford algebra to be *N*-filtered, and the degree of the monomial *m* is the length of *m*.

EXAMPLES:

```

sage: Q = QuadraticForm(ZZ, 3, [1, 2, 3, 4, 5, 6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.degree_on_basis((0,))
1

```

(continues on next page)

(continued from previous page)

```
sage: Cl.degree_on_basis((0,1))
2
```

```
>>> from sage.all import *
>>> Q = QuadraticForm(ZZ, Integer(3), [Integer(1),Integer(2),Integer(3),
↳Integer(4),Integer(5),Integer(6)])
>>> Cl = CliffordAlgebra(Q, names=('x', 'y', 'z',)); (x, y, z,) = Cl._first_
↳ngens(3)
>>> Cl.degree_on_basis((Integer(0),))
1
>>> Cl.degree_on_basis((Integer(0),Integer(1)))
2
```

dimension()

Return the rank of `self` as a free module.

Let V be a free R -module of rank n ; then, $Cl(V, Q)$ is a free R -module of rank 2^n .

EXAMPLES:

```
sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.dimension()
8
```

```
>>> from sage.all import *
>>> Q = QuadraticForm(ZZ, Integer(3), [Integer(1),Integer(2),Integer(3),
↳Integer(4),Integer(5),Integer(6)])
>>> Cl = CliffordAlgebra(Q, names=('x', 'y', 'z',)); (x, y, z,) = Cl._first_
↳ngens(3)
>>> Cl.dimension()
8
```

free_module()

Return the underlying free module V of `self`.

This is the free module on which the quadratic form that was used to construct `self` is defined.

EXAMPLES:

```
sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.free_module()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> Q = QuadraticForm(ZZ, Integer(3), [Integer(1),Integer(2),Integer(3),
↳Integer(4),Integer(5),Integer(6)])
>>> Cl = CliffordAlgebra(Q, names=('x', 'y', 'z',)); (x, y, z,) = Cl._first_
↳ngens(3)
>>> Cl.free_module()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

gen(i)

Return the i -th standard generator of the algebra `self`.

This is the i -th basis vector of the vector space on which the quadratic form defining `self` is defined, regarded as an element of `self`.

EXAMPLES:

```
sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: [Cl.gen(i) for i in range(3)]
[x, y, z]
```

```
>>> from sage.all import *
>>> Q = QuadraticForm(ZZ, Integer(3), [Integer(1),Integer(2),Integer(3),
↳ Integer(4),Integer(5),Integer(6)])
>>> Cl = CliffordAlgebra(Q, names=('x', 'y', 'z')); (x, y, z) = Cl._first_
↳ ngens(3)
>>> [Cl.gen(i) for i in range(Integer(3))]
[x, y, z]
```

gens()

Return the generators of `self` (as an algebra).

EXAMPLES:

```
sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.gens()
(x, y, z)
```

```
>>> from sage.all import *
>>> Q = QuadraticForm(ZZ, Integer(3), [Integer(1),Integer(2),Integer(3),
↳ Integer(4),Integer(5),Integer(6)])
>>> Cl = CliffordAlgebra(Q, names=('x', 'y', 'z')); (x, y, z) = Cl._first_
↳ ngens(3)
>>> Cl.gens()
(x, y, z)
```

graded_algebra()

Return the associated graded algebra of `self`.

EXAMPLES:

```
sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.graded_algebra()
The exterior algebra of rank 3 over Integer Ring
```

```
>>> from sage.all import *
>>> Q = QuadraticForm(ZZ, Integer(3), [Integer(1),Integer(2),Integer(3),
↳ Integer(4),Integer(5),Integer(6)])
>>> Cl = CliffordAlgebra(Q, names=('x', 'y', 'z')); (x, y, z) = Cl._first_
↳ ngens(3)
```

(continues on next page)

(continued from previous page)

```
>>> Cl.graded_algebra()
The exterior algebra of rank 3 over Integer Ring
```

is_commutative()

Check if self is a commutative algebra.

EXAMPLES:

```
sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.is_commutative()
False
```

```
>>> from sage.all import *
>>> Q = QuadraticForm(ZZ, Integer(3), [Integer(1),Integer(2),Integer(3),
↳Integer(4),Integer(5),Integer(6)])
>>> Cl = CliffordAlgebra(Q, names=('x', 'y', 'z',)); (x, y, z,) = Cl._first_
↳ngens(3)
>>> Cl.is_commutative()
False
```

lift_isometry(m, names=None)Lift an invertible isometry m of the quadratic form of self to a Clifford algebra morphism.

Given an invertible linear map $m : V \rightarrow W$ (here represented by a matrix acting on column vectors), this method returns the algebra morphism $Cl(m)$ from $Cl(V, Q)$ to $Cl(W, m^{-1}(Q))$, where $Cl(V, Q)$ is the Clifford algebra self and where $m^{-1}(Q)$ is the pullback of the quadratic form Q to W along the inverse map $m^{-1} : W \rightarrow V$. See the documentation of [CliffordAlgebra](#) for how this pullback and the morphism $Cl(m)$ are defined.

INPUT:

- m – an isometry of the quadratic form of self
- names – (default: 'e') the names of the generators of the Clifford algebra of the codomain of (the map represented by) m

OUTPUT: the algebra morphism $Cl(m)$ from self to $Cl(W, m^{-1}(Q))$

EXAMPLES:

```
sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: m = matrix([[1,1,2],[0,1,1],[0,0,1]])
sage: phi = Cl.lift_isometry(m, 'abc')
sage: phi(x)
a
sage: phi(y)
a + b
sage: phi(x*y)
a*b + 1
sage: phi(x) * phi(y)
a*b + 1
sage: phi(z*y)
a*b - a*c - b*c
```

(continues on next page)

(continued from previous page)

```
sage: phi(z) * phi(y)
a*b - a*c - b*c
sage: phi(x + z) * phi(y + z) == phi((x + z) * (y + z))
True
```

```
>>> from sage.all import *
>>> Q = QuadraticForm(ZZ, Integer(3), [Integer(1), Integer(2), Integer(3),
↳ Integer(4), Integer(5), Integer(6)])
>>> Cl = CliffordAlgebra(Q, names=('x', 'y', 'z')); (x, y, z) = Cl._first_
↳ ngens(3)
>>> m = matrix([[Integer(1), Integer(1), Integer(2)], [Integer(0), Integer(1),
↳ Integer(1)], [Integer(0), Integer(0), Integer(1)]])
>>> phi = Cl.lift_isometry(m, 'abc')
>>> phi(x)
a
>>> phi(y)
a + b
>>> phi(x*y)
a*b + 1
>>> phi(x) * phi(y)
a*b + 1
>>> phi(z*y)
a*b - a*c - b*c
>>> phi(z) * phi(y)
a*b - a*c - b*c
>>> phi(x + z) * phi(y + z) == phi((x + z) * (y + z))
True
```

lift_module_morphism (*m*, *names=None*)

Lift the matrix *m* to an algebra morphism of Clifford algebras.

Given a linear map $m : W \rightarrow V$ (here represented by a matrix acting on column vectors), this method returns the algebra morphism $Cl(m) : Cl(W, m(Q)) \rightarrow Cl(V, Q)$, where $Cl(V, Q)$ is the Clifford algebra `self` and where $m(Q)$ is the pullback of the quadratic form *Q* to *W*. See the documentation of [CliffordAlgebra](#) for how this pullback and the morphism $Cl(m)$ are defined.

Note

This is a map into `self`.

INPUT:

- *m* – a matrix
- *names* – (default: 'e') the names of the generators of the Clifford algebra of the domain of (the map represented by) *m*

OUTPUT: the algebra morphism $Cl(m)$ from $Cl(W, m(Q))$ to `self`

EXAMPLES:

```
sage: Q = QuadraticForm(ZZ, 3, [1, 2, 3, 4, 5, 6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
```

(continues on next page)

(continued from previous page)

```

sage: m = matrix([[1,-1,-1],[0,1,-1],[1,1,1]])
sage: phi = Cl.lift_module_morphism(m, 'abc')
sage: phi
Generic morphism:
  From: The Clifford algebra of the Quadratic form in 3 variables over
↳Integer Ring with coefficients:
[ 10 17 3 ]
[ * 11 0 ]
[ * * 5 ]
  To:   The Clifford algebra of the Quadratic form in 3 variables over
↳Integer Ring with coefficients:
[ 1 2 3 ]
[ * 4 5 ]
[ * * 6 ]
sage: a,b,c = phi.domain().gens()
sage: phi(a)
x + z
sage: phi(b)
-x + y + z
sage: phi(c)
-x - y + z
sage: phi(a + 3*b)
-2*x + 3*y + 4*z
sage: phi(a) + 3*phi(b)
-2*x + 3*y + 4*z
sage: phi(a*b)
x*y + 2*x*z - y*z + 7
sage: phi(b*a)
-x*y - 2*x*z + y*z + 10
sage: phi(a*b + c)
x*y + 2*x*z - y*z - x - y + z + 7
sage: phi(a*b) + phi(c)
x*y + 2*x*z - y*z - x - y + z + 7

```

```

>>> from sage.all import *
>>> Q = QuadraticForm(ZZ, Integer(3), [Integer(1),Integer(2),Integer(3),
↳Integer(4),Integer(5),Integer(6)])
>>> Cl = CliffordAlgebra(Q, names=('x', 'y', 'z')); (x, y, z) = Cl._first_
↳ngens(3)
>>> m = matrix([[Integer(1),-Integer(1),-Integer(1)],[Integer(0),Integer(1),-
↳Integer(1)],[Integer(1),Integer(1),Integer(1)]])
>>> phi = Cl.lift_module_morphism(m, 'abc')
>>> phi
Generic morphism:
  From: The Clifford algebra of the Quadratic form in 3 variables over
↳Integer Ring with coefficients:
[ 10 17 3 ]
[ * 11 0 ]
[ * * 5 ]
  To:   The Clifford algebra of the Quadratic form in 3 variables over
↳Integer Ring with coefficients:
[ 1 2 3 ]

```

(continues on next page)

(continued from previous page)

```

[ * 4 5 ]
[ * * 6 ]
>>> a,b,c = phi.domain().gens()
>>> phi(a)
x + z
>>> phi(b)
-x + y + z
>>> phi(c)
-x - y + z
>>> phi(a + Integer(3)*b)
-2*x + 3*y + 4*z
>>> phi(a) + Integer(3)*phi(b)
-2*x + 3*y + 4*z
>>> phi(a*b)
x*y + 2*x*z - y*z + 7
>>> phi(b*a)
-x*y - 2*x*z + y*z + 10
>>> phi(a*b + c)
x*y + 2*x*z - y*z - x - y + z + 7
>>> phi(a*b) + phi(c)
x*y + 2*x*z - y*z - x - y + z + 7

```

We check that the map is an algebra morphism:

```

sage: phi(a)*phi(b)
x*y + 2*x*z - y*z + 7
sage: phi(a*b)
x*y + 2*x*z - y*z + 7
sage: phi(a*a)
10
sage: phi(a)*phi(a)
10
sage: phi(b*a)
-x*y - 2*x*z + y*z + 10
sage: phi(b) * phi(a)
-x*y - 2*x*z + y*z + 10
sage: phi((a + b)*(a + c)) == phi(a + b) * phi(a + c)
True

```

```

>>> from sage.all import *
>>> phi(a)*phi(b)
x*y + 2*x*z - y*z + 7
>>> phi(a*b)
x*y + 2*x*z - y*z + 7
>>> phi(a*a)
10
>>> phi(a)*phi(a)
10
>>> phi(b*a)
-x*y - 2*x*z + y*z + 10
>>> phi(b) * phi(a)
-x*y - 2*x*z + y*z + 10

```

(continues on next page)

(continued from previous page)

```
>>> phi((a + b)*(a + c)) == phi(a + b) * phi(a + c)
True
```

We can also lift arbitrary linear maps:

```
sage: m = matrix([[1,1],[0,1],[1,1]])
sage: phi = Cl.lift_module_morphism(m, 'ab')
sage: a,b = phi.domain().gens()
sage: phi(a)
x + z
sage: phi(b)
x + y + z
sage: phi(a*b)
x*y - y*z + 15
sage: phi(a)*phi(b)
x*y - y*z + 15
sage: phi(b*a)
-x*y + y*z + 12
sage: phi(b)*phi(a)
-x*y + y*z + 12

sage: m = matrix([[1,1,1,2],[0,1,1,1],[0,1,1,1]])
sage: phi = Cl.lift_module_morphism(m, 'abcd')
sage: a,b,c,d = phi.domain().gens()
sage: phi(a)
x
sage: phi(b)
x + y + z
sage: phi(c)
x + y + z
sage: phi(d)
2*x + y + z
sage: phi(a*b*c + d*a)
-x*y - x*z + 21*x + 7
sage: phi(a*b*c*d)
21*x*y + 21*x*z + 42
```

```
>>> from sage.all import *
>>> m = matrix([[Integer(1),Integer(1)],[Integer(0),Integer(1)],[Integer(1),
↳Integer(1)]])
>>> phi = Cl.lift_module_morphism(m, 'ab')
>>> a,b = phi.domain().gens()
>>> phi(a)
x + z
>>> phi(b)
x + y + z
>>> phi(a*b)
x*y - y*z + 15
>>> phi(a)*phi(b)
x*y - y*z + 15
>>> phi(b*a)
-x*y + y*z + 12
```

(continues on next page)

(continued from previous page)

```

>>> phi(b)*phi(a)
-x*y + y*z + 12

>>> m = matrix([[Integer(1),Integer(1),Integer(1),Integer(2)], [Integer(0),
↳Integer(1),Integer(1),Integer(1)], [Integer(0),Integer(1),Integer(1),
↳Integer(1)]])
>>> phi = Cl.lift_module_morphism(m, 'abcd')
>>> a,b,c,d = phi.domain().gens()
>>> phi(a)
x
>>> phi(b)
x + y + z
>>> phi(c)
x + y + z
>>> phi(d)
2*x + y + z
>>> phi(a*b*c + d*a)
-x*y - x*z + 21*x + 7
>>> phi(a*b*c*d)
21*x*y + 21*x*z + 42

```

ngens()

Return the number of algebra generators of self.

EXAMPLES:

```

sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.ngens()
3

```

```

>>> from sage.all import *
>>> Q = QuadraticForm(ZZ, Integer(3), [Integer(1),Integer(2),Integer(3),
↳Integer(4),Integer(5),Integer(6)])
>>> Cl = CliffordAlgebra(Q, names=('x', 'y', 'z')); (x, y, z) = Cl._first_
↳ngens(3)
>>> Cl.ngens()
3

```

one_basis()

Return the basis index of the element 1. The element 1 is indexed by the emptyset, which is represented by the `sage.data_structures.bitset.Bitset 0`.

EXAMPLES:

```

sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.one_basis()
0

```

```

>>> from sage.all import *
>>> Q = QuadraticForm(ZZ, Integer(3), [Integer(1),Integer(2),Integer(3),

```

(continues on next page)

(continued from previous page)

```

↪Integer(4),Integer(5),Integer(6)])
>>> Cl = CliffordAlgebra(Q, names=('x', 'y', 'z')); (x, y, z) = Cl._first_
↪ngens(3)
>>> Cl.one_basis()
0

```

pseudoscalar()

Return the unit pseudoscalar of *self*.

Given the basis $e_1, e_2, \dots, e_n$ of the underlying R -module, the unit pseudoscalar is defined as $e_1 \cdot e_2 \cdots e_n$.

This depends on the choice of basis.

EXAMPLES:

```

sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.pseudoscalar()
x*y*z

sage: Q = QuadraticForm(ZZ, 0, [])
sage: Cl = CliffordAlgebra(Q)
sage: Cl.pseudoscalar()
1

```

```

>>> from sage.all import *
>>> Q = QuadraticForm(ZZ, Integer(3), [Integer(1),Integer(2),Integer(3),
↪Integer(4),Integer(5),Integer(6)])
>>> Cl = CliffordAlgebra(Q, names=('x', 'y', 'z')); (x, y, z) = Cl._first_
↪ngens(3)
>>> Cl.pseudoscalar()
x*y*z

>>> Q = QuadraticForm(ZZ, Integer(0), [])
>>> Cl = CliffordAlgebra(Q)
>>> Cl.pseudoscalar()
1

```

REFERENCES:

- [Wikipedia article Classification_of_Clifford_algebras#Unit_pseudoscalar](#)

quadratic_form()

Return the quadratic form of *self*.

This is the quadratic form used to define *self*. The quadratic form on *self* is yet to be implemented.

EXAMPLES:

```

sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.quadratic_form()
Quadratic form in 3 variables over Integer Ring with coefficients:
[ 1 2 3 ]
[ * 4 5 ]
[ * * 6 ]

```

```

>>> from sage.all import *
>>> Q = QuadraticForm(ZZ, Integer(3), [Integer(1), Integer(2), Integer(3),
↳ Integer(4), Integer(5), Integer(6)])
>>> Cl = CliffordAlgebra(Q, names=('x', 'y', 'z',)); (x, y, z,) = Cl._first_
↳ ngens(3)
>>> Cl.quadratic_form()
Quadratic form in 3 variables over Integer Ring with coefficients:
[ 1 2 3 ]
[ * 4 5 ]
[ * * 6 ]

```

supercenter_basis()

Return a list of elements which correspond to a basis for the supercenter of *self*.

This assumes that the ground ring can be used to compute the kernel of a matrix.

See also

center_basis(), <http://math.stackexchange.com/questions/129183/center-of-clifford-algebra-depending-on-the-parity-of-dim-v>

Todo

Deprecate this in favor of a method called *supercenter()* once subalgebras are properly implemented in Sage.

EXAMPLES:

```

sage: Q = QuadraticForm(QQ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: SZ = Cl.supercenter_basis(); SZ
(1,)
sage: all(z.supercommutator(b) == 0 for z in SZ for b in Cl.basis())
True

sage: Q = QuadraticForm(QQ, 3, [1,-2,-3, 4, 2, 1])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.supercenter_basis()
(1,)

sage: Q = QuadraticForm(QQ, 2, [1,-2,-3])
sage: Cl.<x,y> = CliffordAlgebra(Q)
sage: Cl.supercenter_basis()
(1,)

sage: Q = QuadraticForm(QQ, 2, [-1,1,-3])
sage: Cl.<x,y> = CliffordAlgebra(Q)
sage: Cl.supercenter_basis()
(1,)

```

```

>>> from sage.all import *
>>> Q = QuadraticForm(QQ, Integer(3), [Integer(1), Integer(2), Integer(3),
↳ Integer(4), Integer(5), Integer(6)])
>>> Cl = CliffordAlgebra(Q, names=('x', 'y', 'z',)); (x, y, z,) = Cl._first_
↳ ngens(3)
>>> SZ = Cl.supercenter_basis(); SZ
(1,)
>>> all(z.supercommutator(b) == Integer(0) for z in SZ for b in Cl.basis())
True

>>> Q = QuadraticForm(QQ, Integer(3), [Integer(1), -Integer(2), -Integer(3),
↳ Integer(4), Integer(2), Integer(1)])
>>> Cl = CliffordAlgebra(Q, names=('x', 'y', 'z',)); (x, y, z,) = Cl._first_
↳ ngens(3)
>>> Cl.supercenter_basis()
(1,)

>>> Q = QuadraticForm(QQ, Integer(2), [Integer(1), -Integer(2), -Integer(3)])
>>> Cl = CliffordAlgebra(Q, names=('x', 'y',)); (x, y,) = Cl._first_ngens(2)
>>> Cl.supercenter_basis()
(1,)

>>> Q = QuadraticForm(QQ, Integer(2), [-Integer(1), Integer(1), -Integer(3)])
>>> Cl = CliffordAlgebra(Q, names=('x', 'y',)); (x, y,) = Cl._first_ngens(2)
>>> Cl.supercenter_basis()
(1,)

```

Singular vectors of a quadratic form generate in the supercenter:

```

sage: Q = QuadraticForm(QQ, 3, [1/2, -2, 4, 256/249, 3, -185/8])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.supercenter_basis()
(1, x + 249/322*y + 22/161*z)

sage: Q = QuadraticForm(QQ, 3, [4, 4, -4, 1, -2, 1])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.supercenter_basis()
(1, x + 2*z, y + z, x*y + x*z - 2*y*z)

```

```

>>> from sage.all import *
>>> Q = QuadraticForm(QQ, Integer(3), [Integer(1)/Integer(2), -Integer(2),
↳ Integer(4), Integer(256)/Integer(249), Integer(3), -Integer(185)/Integer(8)])
>>> Cl = CliffordAlgebra(Q, names=('x', 'y', 'z',)); (x, y, z,) = Cl._first_
↳ ngens(3)
>>> Cl.supercenter_basis()
(1, x + 249/322*y + 22/161*z)

>>> Q = QuadraticForm(QQ, Integer(3), [Integer(4), Integer(4), -Integer(4),
↳ Integer(1), -Integer(2), Integer(1)])
>>> Cl = CliffordAlgebra(Q, names=('x', 'y', 'z',)); (x, y, z,) = Cl._first_
↳ ngens(3)
>>> Cl.supercenter_basis()
(1, x + 2*z, y + z, x*y + x*z - 2*y*z)

```

The most degenerate case:

```
sage: Q = QuadraticForm(QQ, 3)
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.supercenter_basis()
(1, x, y, z, x*y, x*z, y*z, x*y*z)
```

```
>>> from sage.all import *
>>> Q = QuadraticForm(QQ, Integer(3))
>>> Cl = CliffordAlgebra(Q, names=('x', 'y', 'z')); (x, y, z,) = Cl._first_
↳ngens(3)
>>> Cl.supercenter_basis()
(1, x, y, z, x*y, x*z, y*z, x*y*z)
```

class sage.algebras.clifford_algebra.**CliffordAlgebraIndices** (*Qdim, degree=None*)

Bases: `UniqueRepresentation`, `Parent`

A facade parent for the indices of Clifford algebra. Users should not create instances of this class directly.

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```
sage: from sage.algebras.clifford_algebra import CliffordAlgebraIndices
sage: idx = CliffordAlgebraIndices(7)
sage: idx.cardinality() == 2^7
True
sage: len(idx) == 2^7
True

sage: idx = CliffordAlgebraIndices(7, 3)
sage: idx.cardinality() == binomial(7, 3)
True
sage: len(idx) == binomial(7, 3)
True
```

```
>>> from sage.all import *
>>> from sage.algebras.clifford_algebra import CliffordAlgebraIndices
>>> idx = CliffordAlgebraIndices(Integer(7))
>>> idx.cardinality() == Integer(2)**Integer(7)
True
>>> len(idx) == Integer(2)**Integer(7)
True

>>> idx = CliffordAlgebraIndices(Integer(7), Integer(3))
>>> idx.cardinality() == binomial(Integer(7), Integer(3))
True
>>> len(idx) == binomial(Integer(7), Integer(3))
True
```

class sage.algebras.clifford_algebra.**ExteriorAlgebra** (*R, names*)

Bases: `CliffordAlgebra`

An exterior algebra of a free module over a commutative ring.

Let V be a module over a commutative ring R . The exterior algebra (or Grassmann algebra) $\Lambda(V)$ of V is defined as the quotient of the tensor algebra $T(V)$ of V modulo the two-sided ideal generated by all tensors of the form $x \otimes x$ with $x \in V$. The multiplication on $\Lambda(V)$ is denoted by $\wedge$ (so $v_1 \wedge v_2 \wedge \cdots \wedge v_n$ is the projection of $v_1 \otimes v_2 \otimes \cdots \otimes v_n$ onto $\Lambda(V)$) and called the “exterior product” or “wedge product”.

If V is a rank- n free R -module with a basis $\{e_1, \dots, e_n\}$, then $\Lambda(V)$ is the R -algebra noncommutatively generated by the n generators $e_1, \dots, e_n$ subject to the relations $e_i^2 = 0$ for all i , and $e_i e_j = -e_j e_i$ for all $i < j$. As an R -module, $\Lambda(V)$ then has a basis $(\bigwedge_{i \in I} e_i)$ with I ranging over the subsets of $\{1, 2, \dots, n\}$ (where $\bigwedge_{i \in I} e_i$ is the wedge product of e_i for i running through all elements of I from smallest to largest), and hence is free of rank 2^n .

The exterior algebra of an R -module V can also be realized as the Clifford algebra of V for the quadratic form Q given by $Q(v) = 0$ for all vectors $v \in V$. See [CliffordAlgebra](#) for the notion of a Clifford algebra.

The exterior algebra of an R -module V is a connected $\mathbf{Z}$ -graded Hopf superalgebra. It is commutative in the super sense (i.e., the odd elements anticommute and square to 0).

This class implements the exterior algebra $\Lambda(R^n)$ for n a nonnegative integer.

INPUT:

- R – the base ring, *or* the free module whose exterior algebra is to be computed
- `names` – list of strings to name the generators of the exterior algebra; this list can either have one entry only (in which case the generators will be called $e + '0', e + '1', \dots, e + 'n-1'$, with e being said entry), or have n entries (in which case these entries will be used directly as names for the generators)
- n – the number of generators, i.e., the rank of the free module whose exterior algebra is to be computed (this doesn't have to be provided if it can be inferred from the rest of the input)

REFERENCES:

- [Wikipedia article Exterior algebra](#)

Element

alias of `ExteriorAlgebraElement`

antipode_on_basis(m)

Return the antipode on the basis element indexed by m .

Given a basis element ω , the antipode is defined by $S(\omega) = (-1)^{\deg(\omega)} \omega$.

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: E.antipode_on_basis(())
1
sage: E.antipode_on_basis((1,))
-y
sage: E.antipode_on_basis((1,2))
y*z
```

```
>>> from sage.all import *
>>> E = ExteriorAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z) = E._first_
↳ngens(3)
>>> E.antipode_on_basis(())
1
>>> E.antipode_on_basis((Integer(1),))
-y
>>> E.antipode_on_basis((Integer(1), Integer(2)))
y*z
```

boundary (*s_coeff*)

Return the boundary operator ∂ defined by the structure coefficients *s_coeff* of a Lie algebra.

For more on the boundary operator, see [ExteriorAlgebraBoundary](#).

INPUT:

- *s_coeff* – dictionary whose keys are in $I \times I$, where I is the index set of the underlying vector space V , and whose values can be coerced into 1-forms (degree 1 elements) in E (usually, these values will just be elements of V)

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: E.boundary({(0,1): z, (1,2): x, (2,0): y})
Boundary endomorphism of The exterior algebra of rank 3 over Rational Field
```

```
>>> from sage.all import *
>>> E = ExteriorAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z) = E._first_
↳ngens(3)
>>> E.boundary({(Integer(0),Integer(1)): z, (Integer(1),Integer(2)): x,
↳(Integer(2),Integer(0)): y})
Boundary endomorphism of The exterior algebra of rank 3 over Rational Field
```

coboundary (*s_coeff*)

Return the coboundary operator d defined by the structure coefficients *s_coeff* of a Lie algebra.

For more on the coboundary operator, see [ExteriorAlgebraCoboundary](#).

INPUT:

- *s_coeff* – dictionary whose keys are in $I \times I$, where I is the index set of the underlying vector space V , and whose values can be coerced into 1-forms (degree 1 elements) in E (usually, these values will just be elements of V)

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: E.coboundary({(0,1): z, (1,2): x, (2,0): y})
Coboundary endomorphism of The exterior algebra of rank 3 over Rational Field
```

```
>>> from sage.all import *
>>> E = ExteriorAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z) = E._first_
↳ngens(3)
>>> E.coboundary({(Integer(0),Integer(1)): z, (Integer(1),Integer(2)): x,
↳(Integer(2),Integer(0)): y})
Coboundary endomorphism of The exterior algebra of rank 3 over Rational Field
```

coproduct_on_basis (*a*)

Return the coproduct on the basis element indexed by *a*.

The coproduct is defined by

$$\Delta(e_{i_1} \wedge \cdots \wedge e_{i_m}) = \sum_{k=0}^m \sum_{\sigma \in Ush_{k,m-k}} (-1)^\sigma (e_{i_{\sigma(1)}} \wedge \cdots \wedge e_{i_{\sigma(k)}}) \otimes (e_{i_{\sigma(k+1)}} \wedge \cdots \wedge e_{i_{\sigma(m)}}),$$

where $Ush_{k,m-k}$ denotes the set of all $(k, m-k)$ -unshuffles (i.e., permutations in S_m which are increasing on the interval $\{1, 2, \dots, k\}$ and on the interval $\{k+1, k+2, \dots, k+m\}$).

Warning

This coproduct is a homomorphism of superalgebras, not a homomorphism of algebras!

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: E.coproduct_on_basis((0,))
1 # x + x # 1
sage: E.coproduct_on_basis((0,1))
1 # x*y + x # y - y # x + x*y # 1
sage: E.coproduct_on_basis((0,1,2))
1 # x*y*z + x # y*z - y # x*z + x*y # z
+ z # x*y - x*z # y + y*z # x + x*y*z # 1
```

```
>>> from sage.all import *
>>> E = ExteriorAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z,) = E._first_
↳ngens(3)
>>> E.coproduct_on_basis((Integer(0),))
1 # x + x # 1
>>> E.coproduct_on_basis((Integer(0), Integer(1)))
1 # x*y + x # y - y # x + x*y # 1
>>> E.coproduct_on_basis((Integer(0), Integer(1), Integer(2)))
1 # x*y*z + x # y*z - y # x*z + x*y # z
+ z # x*y - x*z # y + y*z # x + x*y*z # 1
```

counit (x)

Return the counit of x .

The counit of an element ω of the exterior algebra is its constant coefficient.

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: elt = x*y - 2*x + 3
sage: E.counit(elt)
3
```

```
>>> from sage.all import *
>>> E = ExteriorAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z,) = E._first_
↳ngens(3)
>>> elt = x*y - Integer(2)*x + Integer(3)
>>> E.counit(elt)
3
```

degree_on_basis (m)

Return the degree of the monomial indexed by m .

The degree of m in the $\mathbf{Z}$ -grading of `self` is defined to be the length of m .

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: E.degree_on_basis(())
```

(continues on next page)

(continued from previous page)

```

0
sage: E.degree_on_basis((0,))
1
sage: E.degree_on_basis((0,1))
2

```

```

>>> from sage.all import *
>>> E = ExteriorAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z) = E._first_
↳ngens(3)
>>> E.degree_on_basis(())
0
>>> E.degree_on_basis((Integer(0),))
1
>>> E.degree_on_basis((Integer(0), Integer(1)))
2

```

interior_product_on_basis(a, b)

Return the interior product $\iota_b a$ of a with respect to b .

See `interior_product()` for more information.

In this method, a and b are supposed to be basis elements (see `interior_product()` for a method that computes interior product of arbitrary elements), and to be input as their keys.

This depends on the choice of basis of the vector space whose exterior algebra is `self`.

EXAMPLES:

```

sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: k = list(E.basis().keys())
sage: E.interior_product_on_basis(k[1], k[1])
1
sage: E.interior_product_on_basis(k[5], k[1])
z
sage: E.interior_product_on_basis(k[2], k[5])
0
sage: E.interior_product_on_basis(k[5], k[2])
0
sage: E.interior_product_on_basis(k[7], k[5])
-y

```

```

>>> from sage.all import *
>>> E = ExteriorAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z) = E._first_
↳ngens(3)
>>> k = list(E.basis().keys())
>>> E.interior_product_on_basis(k[Integer(1)], k[Integer(1)])
1
>>> E.interior_product_on_basis(k[Integer(5)], k[Integer(1)])
z
>>> E.interior_product_on_basis(k[Integer(2)], k[Integer(5)])
0
>>> E.interior_product_on_basis(k[Integer(5)], k[Integer(2)])
0

```

(continues on next page)

(continued from previous page)

```
>>> E.interior_product_on_basis(k[Integer(7)], k[Integer(5)])
-y
```

Check Issue #34694:

```
sage: # needs sage.symbolic
sage: E = ExteriorAlgebra(SR, 'e', 3)
sage: E.inject_variables()
Defining e0, e1, e2
sage: a = (e0*e1).interior_product(e0)
sage: a * e0
-e0*e1
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> E = ExteriorAlgebra(SR, 'e', Integer(3))
>>> E.inject_variables()
Defining e0, e1, e2
>>> a = (e0*e1).interior_product(e0)
>>> a * e0
-e0*e1
```

lift_morphism(*phi*, *names=None*)

Lift the matrix *m* to an algebra morphism of exterior algebras.

Given a linear map $\phi : V \rightarrow W$ (here represented by a matrix acting on column vectors over the base ring of V), this method returns the algebra morphism $\Lambda(\phi) : \Lambda(V) \rightarrow \Lambda(W)$. This morphism is defined on generators $v_i \in \Lambda(V)$ by $v_i \mapsto \phi(v_i)$.

Note

This is the map going out of `self` as opposed to `lift_module_morphism()` for general Clifford algebras.

INPUT:

- *phi* – a linear map ϕ from V to W , encoded as a matrix
- *names* – (default: 'e') the names of the generators of the Clifford algebra of the domain of (the map represented by) *phi*

OUTPUT: the algebra morphism $\Lambda(\phi)$ from `self` to $\Lambda(W)$

EXAMPLES:

```
sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: phi = matrix([[0,1],[1,1],[1,2]]); phi
[0 1]
[1 1]
[1 2]
sage: L = E.lift_morphism(phi, ['a','b','c']); L
Generic morphism:
From: The exterior algebra of rank 2 over Rational Field
```

(continues on next page)

(continued from previous page)

```

To: The exterior algebra of rank 3 over Rational Field
sage: L(x)
b + c
sage: L(y)
a + b + 2*c
sage: L.on_basis()((1,))
a + b + 2*c
sage: p = L(E.one()); p
1
sage: p.parent()
The exterior algebra of rank 3 over Rational Field
sage: L(x*y)
-a*b - a*c + b*c
sage: L(x)*L(y)
-a*b - a*c + b*c
sage: L(x + y)
a + 2*b + 3*c
sage: L(x) + L(y)
a + 2*b + 3*c
sage: L(1/2*x + 2)
1/2*b + 1/2*c + 2
sage: L(E(3))
3

sage: psi = matrix([[1, -3/2]]); psi
[ 1 -3/2]
sage: Lp = E.lift_morphism(psi, ['a']); Lp
Generic morphism:
From: The exterior algebra of rank 2 over Rational Field
To: The exterior algebra of rank 1 over Rational Field
sage: Lp(x)
a
sage: Lp(y)
-3/2*a
sage: Lp(x + 2*y + 3)
-2*a + 3

```

```

>>> from sage.all import *
>>> E = ExteriorAlgebra(QQ, names=('x', 'y',)); (x, y,) = E._first_ngens(2)
>>> phi = matrix([[Integer(0), Integer(1)], [Integer(1), Integer(1)], [Integer(1),
↵Integer(2)]]); phi
[0 1]
[1 1]
[1 2]
>>> L = E.lift_morphism(phi, ['a', 'b', 'c']); L
Generic morphism:
From: The exterior algebra of rank 2 over Rational Field
To: The exterior algebra of rank 3 over Rational Field
>>> L(x)
b + c
>>> L(y)
a + b + 2*c

```

(continues on next page)

(continued from previous page)

```

>>> L.on_basis() ((Integer(1),))
a + b + 2*c
>>> p = L(E.one()); p
1
>>> p.parent()
The exterior algebra of rank 3 over Rational Field
>>> L(x*y)
-a*b - a*c + b*c
>>> L(x)*L(y)
-a*b - a*c + b*c
>>> L(x + y)
a + 2*b + 3*c
>>> L(x) + L(y)
a + 2*b + 3*c
>>> L(Integer(1)/Integer(2)*x + Integer(2))
1/2*b + 1/2*c + 2
>>> L(E(Integer(3)))
3

>>> psi = matrix([[Integer(1), -Integer(3)/Integer(2)]]); psi
[ 1 -3/2]
>>> Lp = E.lift_morphism(psi, ['a']); Lp
Generic morphism:
  From: The exterior algebra of rank 2 over Rational Field
  To:   The exterior algebra of rank 1 over Rational Field
>>> Lp(x)
a
>>> Lp(y)
-3/2*a
>>> Lp(x + Integer(2)*y + Integer(3))
-2*a + 3

```

lifted_bilinear_form(M)

Return the bilinear form on the exterior algebra `self =  $\Lambda(V)$` which is obtained by lifting the bilinear form f on V given by the matrix M .

Let V be a module over a commutative ring R , and let $f : V \times V \rightarrow R$ be a bilinear form on V . Then, a bilinear form $\Lambda(f) : \Lambda(V) \times \Lambda(V) \rightarrow R$ on $\Lambda(V)$ can be canonically defined as follows: For every $n \in \mathbf{N}$, $m \in \mathbf{N}$, $v_1, v_2, \dots, v_n, w_1, w_2, \dots, w_m \in V$, we define

$$\Lambda(f)(v_1 \wedge v_2 \wedge \dots \wedge v_n, w_1 \wedge w_2 \wedge \dots \wedge w_m) := \begin{cases} 0, & \text{if } n \neq m; \\ \det G, & \text{if } n = m \end{cases},$$

where G is the $n \times m$ -matrix whose (i, j) -th entry is $f(v_i, w_j)$. This bilinear form $\Lambda(f)$ is known as the bilinear form on $\Lambda(V)$ obtained by lifting the bilinear form f . Its restriction to the 1-st homogeneous component V of $\Lambda(V)$ is f .

The bilinear form $\Lambda(f)$ is symmetric if f is.

INPUT:

- M – a matrix over the same base ring as `self`, whose (i, j) -th entry is $f(e_i, e_j)$, where $(e_1, e_2, \dots, e_N)$ is the standard basis of the module V for which `self =  $\Lambda(V)$` (so that $N = \dim(V)$), and where f is the bilinear form which is to be lifted.

OUTPUT:

A bivariate function which takes two elements p and q of `self` to $\Lambda(f)(p, q)$.

Note

This takes a bilinear form on V as matrix, and returns a bilinear form on `self` as a function in two arguments. We do not return the bilinear form as a matrix since this matrix can be huge and one often needs just a particular value.

Todo

Implement a class for bilinear forms and rewrite this method to use that class.

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: M = Matrix(QQ, [[1, 2, 3], [2, 3, 4], [3, 4, 5]])
sage: Eform = E.lifted_bilinear_form(M)
sage: Eform
Bilinear Form from The exterior algebra of rank 3 over Rational
Field (+) The exterior algebra of rank 3 over Rational Field to
Rational Field
sage: Eform(x*y, y*z)
-1
sage: Eform(x*y, y)
0
sage: Eform(x*(y+z), y*z)
-3
sage: Eform(x*(y+z), y*(z+x))
0
sage: N = Matrix(QQ, [[3, 1, 7], [2, 0, 4], [-1, -3, -1]])
sage: N.determinant()
-8
sage: Eform = E.lifted_bilinear_form(N)
sage: Eform(x, E.one())
0
sage: Eform(x, x*z*y)
0
sage: Eform(E.one(), E.one())
1
sage: Eform(E.zero(), E.one())
0
sage: Eform(x, y)
1
sage: Eform(z, y)
-3
sage: Eform(x*z, y*z)
20
sage: Eform(x+x*y+x*y*z, z+z*y+z*y*x)
11
```

```

>>> from sage.all import *
>>> E = ExteriorAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z) = E._first_
↳ngens(3)
>>> M = Matrix(QQ, [[Integer(1), Integer(2), Integer(3)], [Integer(2),
↳Integer(3), Integer(4)], [Integer(3), Integer(4), Integer(5)]])
>>> Eform = E.lifted_bilinear_form(M)
>>> Eform
Bilinear Form from The exterior algebra of rank 3 over Rational
Field (+) The exterior algebra of rank 3 over Rational Field to
Rational Field
>>> Eform(x*y, y*z)
-1
>>> Eform(x*y, y)
0
>>> Eform(x*(y+z), y*z)
-3
>>> Eform(x*(y+z), y*(z+x))
0
>>> N = Matrix(QQ, [[Integer(3), Integer(1), Integer(7)], [Integer(2),
↳Integer(0), Integer(4)], [-Integer(1), -Integer(3), -Integer(1)]])
>>> N.determinant()
-8
>>> Eform = E.lifted_bilinear_form(N)
>>> Eform(x, E.one())
0
>>> Eform(x, x*z*y)
0
>>> Eform(E.one(), E.one())
1
>>> Eform(E.zero(), E.one())
0
>>> Eform(x, y)
1
>>> Eform(z, y)
-3
>>> Eform(x*z, y*z)
20
>>> Eform(x+x*y+x*y*z, z+z*y+z*y*x)
11

```

Todo

Another way to compute this bilinear form seems to be to map x and y to the appropriate Clifford algebra and there compute $x^t y$, then send the result back to the exterior algebra and return its constant coefficient. Or something like this. Once the maps to the Clifford and back are implemented, check if this is faster.

volume_form()

Return the volume form of `self`.

Given the basis $e_1, e_2, \dots, e_n$ of the underlying R -module, the volume form is defined as $e_1 \wedge e_2 \wedge \dots \wedge e_n$.

This depends on the choice of basis.

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: E.volume_form()
x*y*z
```

```
>>> from sage.all import *
>>> E = ExteriorAlgebra(QQ, names=('x', 'y', 'z',)); (x, y, z,) = E._first_
↳ngens(3)
>>> E.volume_form()
x*y*z
```

class sage.algebras.clifford_algebra.**ExteriorAlgebraBoundary** (*E*, *s_coeff*)

Bases: *ExteriorAlgebraDifferential*

The boundary ∂ of an exterior algebra $\Lambda(L)$ defined by the structure coefficients of L .

Let L be a Lie algebra. We give the exterior algebra $E = \Lambda(L)$ a chain complex structure by considering a differential $\partial : \Lambda^{k+1}(L) \rightarrow \Lambda^k(L)$ defined by

$$\partial(x_1 \wedge x_2 \wedge \cdots \wedge x_{k+1}) = \sum_{i < j} (-1)^{i+j+1} [x_i, x_j] \wedge x_1 \wedge \cdots \wedge \hat{x}_i \wedge \cdots \wedge \hat{x}_j \wedge \cdots \wedge x_{k+1}$$

where $\hat{x}_i$ denotes a missing index. The corresponding homology is the Lie algebra homology.

INPUT:

- *E* – an exterior algebra of a vector space L
- *s_coeff* – dictionary whose keys are in $I \times I$, where I is the index set of the basis of the vector space L , and whose values can be coerced into 1-forms (degree 1 elements) in E ; this dictionary will be used to define the Lie algebra structure on L (indeed, the i -th coordinate of the Lie bracket of the j -th and k -th basis vectors of L for $j < k$ is set to be the value at the key (j, k) if this key appears in *s_coeff*, or otherwise the negated of the value at the key (k, j))

Warning

The values of *s_coeff* are supposed to be coercible into 1-forms in E ; but they can also be dictionaries themselves (in which case they are interpreted as giving the coordinates of vectors in L). In the interest of speed, these dictionaries are not sanitized or checked.

Warning

For any two distinct elements i and j of I , the dictionary *s_coeff* must have only one of the pairs (i, j) and (j, i) as a key. This is not checked.

EXAMPLES:

We consider the differential given by Lie algebra given by the cross product $\times$ of $\mathbf{R}^3$:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: par = E.boundary({(0,1): z, (1,2): x, (2,0): y})
sage: par(x)
0
```

(continues on next page)

(continued from previous page)

```
sage: par(x*y)
z
sage: par(x*y*z)
0
sage: par(x+y-y*z+x*y)
-x + z
sage: par(E.zero())
0
```

```
>>> from sage.all import *
>>> E = ExteriorAlgebra(QQ, names=('x', 'y', 'z',)); (x, y, z,) = E._first_
↳ngens(3)
>>> par = E.boundary({(Integer(0),Integer(1)): z, (Integer(1),Integer(2)): x,
↳(Integer(2),Integer(0)): y})
>>> par(x)
0
>>> par(x*y)
z
>>> par(x*y*z)
0
>>> par(x+y-y*z+x*y)
-x + z
>>> par(E.zero())
0
```

We check that $\partial \circ \partial = 0$:

```
sage: p2 = par * par
sage: all(p2(b) == 0 for b in E.basis())
True
```

```
>>> from sage.all import *
>>> p2 = par * par
>>> all(p2(b) == Integer(0) for b in E.basis())
True
```

Another example: the Lie algebra $\mathfrak{sl}_2$, which has a basis e, f, h satisfying $[h, e] = 2e$, $[h, f] = -2f$, and $[e, f] = h$:

```
sage: E.<e,f,h> = ExteriorAlgebra(QQ)
sage: par = E.boundary({(0,1): h, (2,1): -2*f, (2,0): 2*e})
sage: par(E.zero())
0
sage: par(e)
0
sage: par(e*f)
h
sage: par(f*h)
2*f
sage: par(h*f)
-2*f
sage: C = par.chain_complex(); C
Chain complex with at most 4 nonzero terms over Rational Field
```

(continues on next page)

(continued from previous page)

```

sage: ascii_art(C)
          [ 0 -2  0]          [0]
          [ 0  0  2]          [0]
    [0 0 0]    [ 1  0  0]    [0]
0 <-- C_0 <----- C_1 <----- C_2 <---- C_3 <-- 0
sage: C.homology()
{0: Vector space of dimension 1 over Rational Field,
 1: Vector space of dimension 0 over Rational Field,
 2: Vector space of dimension 0 over Rational Field,
 3: Vector space of dimension 1 over Rational Field}

```

```

>>> from sage.all import *
>>> E = ExteriorAlgebra(QQ, names=('e', 'f', 'h',)); (e, f, h,) = E._first_
↳ngens(3)
>>> par = E.boundary({(Integer(0),Integer(1)): h, (Integer(2),Integer(1)): -
↳Integer(2)*f, (Integer(2),Integer(0)): Integer(2)*e})
>>> par(E.zero())
0
>>> par(e)
0
>>> par(e*f)
h
>>> par(f*h)
2*f
>>> par(h*f)
-2*f
>>> C = par.chain_complex(); C
Chain complex with at most 4 nonzero terms over Rational Field
>>> ascii_art(C)
          [ 0 -2  0]          [0]
          [ 0  0  2]          [0]
    [0 0 0]    [ 1  0  0]    [0]
0 <-- C_0 <----- C_1 <----- C_2 <---- C_3 <-- 0
>>> C.homology()
{0: Vector space of dimension 1 over Rational Field,
 1: Vector space of dimension 0 over Rational Field,
 2: Vector space of dimension 0 over Rational Field,
 3: Vector space of dimension 1 over Rational Field}

```

Over the integers:

```

sage: C = par.chain_complex(R=ZZ); C
Chain complex with at most 4 nonzero terms over Integer Ring
sage: ascii_art(C)
          [ 0 -2  0]          [0]
          [ 0  0  2]          [0]
    [0 0 0]    [ 1  0  0]    [0]
0 <-- C_0 <----- C_1 <----- C_2 <---- C_3 <-- 0
sage: C.homology()
{0: Z, 1: C2 x C2, 2: 0, 3: Z}

```

```

>>> from sage.all import *
>>> C = par.chain_complex(R=ZZ); C
Chain complex with at most 4 nonzero terms over Integer Ring
>>> ascii_art(C)
          [ 0 -2  0]      [0]
          [ 0  0  2]      [0]
[0 0 0]      [ 1  0  0]      [0]
0 <-- C_0 <----- C_1 <----- C_2 <---- C_3 <-- 0
>>> C.homology()
{0: Z, 1: C2 x C2, 2: 0, 3: Z}

```

REFERENCES:

- [Wikipedia article Exterior_algebra#Lie_algebra_homology](#)

chain_complex (*R=None*)

Return the chain complex over *R* determined by *self*.

INPUT:

- *R* – the base ring; the default is the base ring of the exterior algebra

EXAMPLES:

```

sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: par = E.boundary({(0,1): z, (1,2): x, (2,0): y})
sage: C = par.chain_complex(); C
Chain complex with at most 4 nonzero terms over Rational Field
sage: ascii_art(C)
          [ 0  0  1]      [0]
          [ 0 -1  0]      [0]
[0 0 0]      [ 1  0  0]      [0]
0 <-- C_0 <----- C_1 <----- C_2 <---- C_3 <-- 0

```

```

>>> from sage.all import *
>>> E = ExteriorAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z) = E._first_
↳ngens(3)
>>> par = E.boundary({(Integer(0),Integer(1)): z, (Integer(1),Integer(2)): x,
↳ (Integer(2),Integer(0)): y})
>>> C = par.chain_complex(); C
Chain complex with at most 4 nonzero terms over Rational Field
>>> ascii_art(C)
          [ 0  0  1]      [0]
          [ 0 -1  0]      [0]
[0 0 0]      [ 1  0  0]      [0]
0 <-- C_0 <----- C_1 <----- C_2 <---- C_3 <-- 0

```

class sage.algebras.clifford_algebra.**ExteriorAlgebraCoboundary** (*E, s_coeff*)

Bases: [ExteriorAlgebraDifferential](#)

The coboundary d of an exterior algebra $\Lambda(L)$ defined by the structure coefficients of a Lie algebra L .

Let L be a Lie algebra. We endow its exterior algebra $E = \Lambda(L)$ with a cochain complex structure by considering a differential $d : \Lambda^k(L) \rightarrow \Lambda^{k+1}(L)$ defined by

$$dx_i = \sum_{j < k} s_{jk}^i x_j x_k,$$

where $(x_1, x_2, \dots, x_n)$ is a basis of L , and where s_{jk}^i is the x_i -coordinate of the Lie bracket $[x_j, x_k]$.

The corresponding cohomology is the Lie algebra cohomology of L .

This can also be thought of as the exterior derivative, in which case the resulting cohomology is the de Rham cohomology of a manifold whose exterior algebra of differential forms is E .

INPUT:

- E – an exterior algebra of a vector space L
- `s_coeff` – dictionary whose keys are in $I \times I$, where I is the index set of the basis of the vector space L , and whose values can be coerced into 1-forms (degree 1 elements) in E ; this dictionary will be used to define the Lie algebra structure on L (indeed, the i -th coordinate of the Lie bracket of the j -th and k -th basis vectors of L for $j < k$ is set to be the value at the key (j, k) if this key appears in `s_coeff`, or otherwise the negated of the value at the key (k, j))

Warning

For any two distinct elements i and j of I , the dictionary `s_coeff` must have only one of the pairs (i, j) and (j, i) as a key. This is not checked.

EXAMPLES:

We consider the differential coming from the Lie algebra given by the cross product $\times$ of $\mathbf{R}^3$:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: d = E.coboundary({(0,1): z, (1,2): x, (0, 2): -y})
sage: d(x)
y*z
sage: d(y)
-x*z
sage: d(x+y-y*z)
-x*z + y*z
sage: d(x*y)
0
sage: d(E.one())
0
sage: d(E.zero())
0
```

```
>>> from sage.all import *
>>> E = ExteriorAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z) = E._first_
↳ngens(3)
>>> d = E.coboundary({(Integer(0),Integer(1)): z, (Integer(1),Integer(2)): x,
↳ (Integer(0), Integer(2)): -y})
>>> d(x)
y*z
>>> d(y)
-x*z
>>> d(x+y-y*z)
-x*z + y*z
>>> d(x*y)
0
>>> d(E.one())
```

(continues on next page)

(continued from previous page)

```
0
>>> d(E.zero())
0
```

We check that $d \circ d = 0$:

```
sage: d2 = d * d
sage: all(d2(b) == 0 for b in E.basis())
True
```

```
>>> from sage.all import *
>>> d2 = d * d
>>> all(d2(b) == Integer(0) for b in E.basis())
True
```

Another example: the Lie algebra $\mathfrak{sl}_2$, which has a basis e, f, h satisfying $[h, e] = 2e$, $[h, f] = -2f$, and $[e, f] = h$:

```
sage: E.<e,f,h> = ExteriorAlgebra(QQ)
sage: d = E.coboundary({(0,1): h, (2,1): -2*f, (2,0): 2*e})
sage: d(E.zero())
0
sage: d(e)
-2*e*h
sage: d(f)
2*f*h
sage: d(h)
e*f
sage: d(e*f)
0
sage: d(f*h)
0
sage: d(e*h)
0
sage: C = d.chain_complex(); C
Chain complex with at most 4 nonzero terms over Rational Field
sage: ascii_art(C)
          [ 0  0  1]          [0]
          [-2  0  0]          [0]
    [0 0 0]    [ 0  2  0]    [0]
0 <-- C_3 <----- C_2 <----- C_1 <---- C_0 <-- 0
sage: C.homology()
{0: Vector space of dimension 1 over Rational Field,
 1: Vector space of dimension 0 over Rational Field,
 2: Vector space of dimension 0 over Rational Field,
 3: Vector space of dimension 1 over Rational Field}
```

```
>>> from sage.all import *
>>> E = ExteriorAlgebra(QQ, names=('e', 'f', 'h',)); (e, f, h,) = E._first_
    ↪ngens(3)
>>> d = E.coboundary({(Integer(0),Integer(1)): h, (Integer(2),Integer(1)): -
    ↪Integer(2)*f, (Integer(2),Integer(0)): Integer(2)*e})
>>> d(E.zero())
```

(continues on next page)

(continued from previous page)

```

0
>>> d(e)
-2*e*h
>>> d(f)
2*f*h
>>> d(h)
e*f
>>> d(e*f)
0
>>> d(f*h)
0
>>> d(e*h)
0
>>> C = d.chain_complex(); C
Chain complex with at most 4 nonzero terms over Rational Field
>>> ascii_art(C)
           [ 0  0  1]           [0]
           [-2  0  0]           [0]
    [0 0 0]   [ 0  2  0]           [0]
0 <-- C_3 <----- C_2 <----- C_1 <---- C_0 <-- 0
>>> C.homology()
{0: Vector space of dimension 1 over Rational Field,
 1: Vector space of dimension 0 over Rational Field,
 2: Vector space of dimension 0 over Rational Field,
 3: Vector space of dimension 1 over Rational Field}

```

Over the integers:

```

sage: C = d.chain_complex(R=ZZ); C
Chain complex with at most 4 nonzero terms over Integer Ring
sage: ascii_art(C)
           [ 0  0  1]           [0]
           [-2  0  0]           [0]
    [0 0 0]   [ 0  2  0]           [0]
0 <-- C_3 <----- C_2 <----- C_1 <---- C_0 <-- 0
sage: C.homology()
{0: Z, 1: 0, 2: C2 x C2, 3: Z}

```

```

>>> from sage.all import *
>>> C = d.chain_complex(R=ZZ); C
Chain complex with at most 4 nonzero terms over Integer Ring
>>> ascii_art(C)
           [ 0  0  1]           [0]
           [-2  0  0]           [0]
    [0 0 0]   [ 0  2  0]           [0]
0 <-- C_3 <----- C_2 <----- C_1 <---- C_0 <-- 0
>>> C.homology()
{0: Z, 1: 0, 2: C2 x C2, 3: Z}

```

REFERENCES:

- [Wikipedia article Exterior_algebra#Differential_geometry](#)

chain_complex (*R=None*)

Return the chain complex over *R* determined by *self*.

INPUT:

- *R* – the base ring; the default is the base ring of the exterior algebra

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: d = E.coboundary({(0,1): z, (1,2): x, (2,0): y})
sage: C = d.chain_complex(); C
Chain complex with at most 4 nonzero terms over Rational Field
sage: ascii_art(C)

          [ 0  0  1]          [0]
          [ 0 -1  0]          [0]
[0 0 0]    [ 1  0  0]          [0]
0 <-- C_3 <----- C_2 <----- C_1 <---- C_0 <-- 0
```

```
>>> from sage.all import *
>>> E = ExteriorAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z) = E._first_
↳ngens(3)
>>> d = E.coboundary({(Integer(0),Integer(1)): z, (Integer(1),Integer(2)): x,
↳ (Integer(2),Integer(0)): y})
>>> C = d.chain_complex(); C
Chain complex with at most 4 nonzero terms over Rational Field
>>> ascii_art(C)

          [ 0  0  1]          [0]
          [ 0 -1  0]          [0]
[0 0 0]    [ 1  0  0]          [0]
0 <-- C_3 <----- C_2 <----- C_1 <---- C_0 <-- 0
```

class sage.algebras.clifford_algebra.**ExteriorAlgebraDifferential** (*E, s_coeff*)

Bases: [ModuleMorphismByLinearity](#), [UniqueRepresentation](#)

Internal class to store the data of a boundary or coboundary of an exterior algebra $\Lambda(L)$ defined by the structure coefficients of a Lie algebra *L*.

See [ExteriorAlgebraBoundary](#) and [ExteriorAlgebraCoboundary](#) for the actual classes, which inherit from this.

Warning

This is not a general class for differentials on the exterior algebra.

homology (*deg=None, **kws*)

Return the homology determined by *self*.

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: par = E.boundary({(0,1): z, (1,2): x, (2,0): y})
sage: par.homology()
{0: Vector space of dimension 1 over Rational Field,
 1: Vector space of dimension 0 over Rational Field,
```

(continues on next page)

(continued from previous page)

```

2: Vector space of dimension 0 over Rational Field,
3: Vector space of dimension 1 over Rational Field}
sage: d = E.coboundary({(0,1): z, (1,2): x, (2,0): y})
sage: d.homology()
{0: Vector space of dimension 1 over Rational Field,
1: Vector space of dimension 0 over Rational Field,
2: Vector space of dimension 0 over Rational Field,
3: Vector space of dimension 1 over Rational Field}

```

```

>>> from sage.all import *
>>> E = ExteriorAlgebra(QQ, names=('x', 'y', 'z',)); (x, y, z,) = E._first_
↳ngens(3)
>>> par = E.boundary({(Integer(0),Integer(1)): z, (Integer(1),Integer(2)): x,
↳(Integer(2),Integer(0)): y})
>>> par.homology()
{0: Vector space of dimension 1 over Rational Field,
1: Vector space of dimension 0 over Rational Field,
2: Vector space of dimension 0 over Rational Field,
3: Vector space of dimension 1 over Rational Field}
>>> d = E.coboundary({(Integer(0),Integer(1)): z, (Integer(1),Integer(2)): x,
↳(Integer(2),Integer(0)): y})
>>> d.homology()
{0: Vector space of dimension 1 over Rational Field,
1: Vector space of dimension 0 over Rational Field,
2: Vector space of dimension 0 over Rational Field,
3: Vector space of dimension 1 over Rational Field}

```

```

class sage.algebras.clifford_algebra.ExteriorAlgebraIdeal (ring, gens, coerce=True,
side='twosided')

```

Bases: `Ideal_nc`

An ideal of the exterior algebra.

EXAMPLES:

```

sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: I = E.ideal(x*y); I
Twosided Ideal (x*y) of The exterior algebra of rank 3 over Rational Field

```

```

>>> from sage.all import *
>>> E = ExteriorAlgebra(QQ, names=('x', 'y', 'z',)); (x, y, z,) = E._first_
↳ngens(3)
>>> I = E.ideal(x*y); I
Twosided Ideal (x*y) of The exterior algebra of rank 3 over Rational Field

```

We can also use it to build a quotient:

```

sage: Q = E.quotient(I); Q
Quotient of The exterior algebra of rank 3 over Rational Field by the ideal (x*y)
sage: Q.inject_variables()
Defining xbar, ybar, zbar
sage: xbar * ybar
0

```

```

>>> from sage.all import *
>>> Q = E.quotient(I); Q
Quotient of The exterior algebra of rank 3 over Rational Field by the ideal (x*y)
>>> Q.inject_variables()
Defining xbar, ybar, zbar
>>> xbar * ybar
0

```

groebner_basis (*term_order=None, reduced=True*)

Return the (reduced) Gröbner basis of *self*.

INPUT:

- *term_order* – the term order used to compute the Gröbner basis; must be one of the following:
 - 'neglex' – (default) negative (read right-to-left) lex order
 - 'degrevlex' – degree reverse lex order
 - 'deglex' – degree lex order
- *reduced* – boolean (default: True); whether or not to return the reduced Gröbner basis

EXAMPLES:

We compute an example:

```

sage: E.<a,b,c,d,e> = ExteriorAlgebra(QQ)
sage: rels = [c*d*e - b*d*e + b*c*e - b*c*d,
....:         c*d*e - a*d*e + a*c*e - a*c*d,
....:         b*d*e - a*d*e + a*b*e - a*b*d,
....:         b*c*e - a*c*e + a*b*e - a*b*c,
....:         b*c*d - a*c*d + a*b*d - a*b*c]
sage: I = E.ideal(rels)
sage: I.groebner_basis()
(-a*b*c + a*b*d - a*c*d + b*c*d,
 -a*b*c + a*b*e - a*c*e + b*c*e,
 -a*b*d + a*b*e - a*d*e + b*d*e,
 -a*c*d + a*c*e - a*d*e + c*d*e)

```

```

>>> from sage.all import *
>>> E = ExteriorAlgebra(QQ, names=('a', 'b', 'c', 'd', 'e')); (a, b, c, d, e,
↪) = E._first_ngens(5)
>>> rels = [c*d*e - b*d*e + b*c*e - b*c*d,
...         c*d*e - a*d*e + a*c*e - a*c*d,
...         b*d*e - a*d*e + a*b*e - a*b*d,
...         b*c*e - a*c*e + a*b*e - a*b*c,
...         b*c*d - a*c*d + a*b*d - a*b*c]
>>> I = E.ideal(rels)
>>> I.groebner_basis()
(-a*b*c + a*b*d - a*c*d + b*c*d,
 -a*b*c + a*b*e - a*c*e + b*c*e,
 -a*b*d + a*b*e - a*d*e + b*d*e,
 -a*c*d + a*c*e - a*d*e + c*d*e)

```

With different term orders:


```
sage: I.groebner_basis("degrevlex")
(b*c*d - b*c*e + b*d*e - c*d*e,
 a*c*d - a*c*e + a*d*e - c*d*e,
 a*b*d - a*b*e + a*d*e - b*d*e,
 a*b*c - a*b*e + a*c*e - b*c*e)
```

```
sage: I.groebner_basis("deglex")
(-a*b*c + a*b*d - a*c*d + b*c*d,
 -a*b*c + a*b*e - a*c*e + b*c*e,
 -a*b*d + a*b*e - a*d*e + b*d*e,
 -a*c*d + a*c*e - a*d*e + c*d*e)
```

```
>>> from sage.all import *
>>> I.groebner_basis("degrevlex")
(b*c*d - b*c*e + b*d*e - c*d*e,
 a*c*d - a*c*e + a*d*e - c*d*e,
 a*b*d - a*b*e + a*d*e - b*d*e,
 a*b*c - a*b*e + a*c*e - b*c*e)

>>> I.groebner_basis("deglex")
(-a*b*c + a*b*d - a*c*d + b*c*d,
 -a*b*c + a*b*e - a*c*e + b*c*e,
 -a*b*d + a*b*e - a*d*e + b*d*e,
 -a*c*d + a*c*e - a*d*e + c*d*e)
```

The example above was computed first using M2, which agrees with the 'degrevlex' ordering:

```
E = QQ[a..e, SkewCommutative => true]
I = ideal( c*d*e - b*d*e + b*c*e - b*c*d,
           c*d*e - a*d*e + a*c*e - a*c*d,
           b*d*e - a*d*e + a*b*e - a*b*d,
           b*c*e - a*c*e + a*b*e - a*b*c,
           b*c*d - a*c*d + a*b*d - a*b*c)
groebnerBasis(I)

returns:
o3 = | bcd-bce+bde-cde acd-ace+ade-cde abd-abe+ade-bde abc-abe+ace-bce |
```

By default, the Gröbner basis is reduced, but we can get non-reduced Gröber bases (which are not unique):

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: I = E.ideal([x+y*z])
sage: I.groebner_basis(reduced=False)
(x*y, x*z, y*z + x, x*y*z)
sage: I.groebner_basis(reduced=True)
(x*y, x*z, y*z + x)
```

```
>>> from sage.all import *
>>> E = ExteriorAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z) = E._first_
↳ngens(3)
>>> I = E.ideal([x+y*z])
>>> I.groebner_basis(reduced=False)
(x*y, x*z, y*z + x, x*y*z)
```

(continues on next page)

(continued from previous page)

```
>>> I.groebner_basis(reduced=True)
(x*y, x*z, y*z + x)
```

However, if we have already computed a reduced Gröbner basis (with a given term order), then we return that:

```
sage: I = E.ideal([x+y*z]) # A fresh ideal
sage: I.groebner_basis()
(x*y, x*z, y*z + x)
sage: I.groebner_basis(reduced=False)
(x*y, x*z, y*z + x)
```

```
>>> from sage.all import *
>>> I = E.ideal([x+y*z]) # A fresh ideal
>>> I.groebner_basis()
(x*y, x*z, y*z + x)
>>> I.groebner_basis(reduced=False)
(x*y, x*z, y*z + x)
```

reduce(*f*)

Reduce *f* modulo *self*.

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: I = E.ideal(x*y);
sage: I.reduce(x*y + x*y*z + z)
z
sage: I.reduce(x*y + x + y)
x + y
sage: I.reduce(x*y + x*y*z)
0

sage: E.<a,b,c,d> = ExteriorAlgebra(QQ)
sage: I = E.ideal([a+b*c])
sage: I.reduce(I.gen(0) * d)
0
```

```
>>> from sage.all import *
>>> E = ExteriorAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z) = E._first_
↳ngens(3)
>>> I = E.ideal(x*y);
>>> I.reduce(x*y + x*y*z + z)
z
>>> I.reduce(x*y + x + y)
x + y
>>> I.reduce(x*y + x*y*z)
0

>>> E = ExteriorAlgebra(QQ, names=('a', 'b', 'c', 'd')); (a, b, c, d) = E._
↳first_ngens(4)
>>> I = E.ideal([a+b*c])
```

(continues on next page)

(continued from previous page)

```
>>> I.reduce(I.gen(Integer(0)) * d)
0
```

5.5 Cluster algebras

This file constructs cluster algebras using the Parent-Element framework. The implementation mainly utilizes structural theorems from [FZ2007].

The key points being used here are these:

- cluster variables are parametrized by their g-vectors;
- g-vectors (together with c-vectors) provide a self-standing model for the combinatorics behind any cluster algebra;
- each cluster variable in any cluster algebra can be computed, by the separation of additions formula, from its g-vector and F-polynomial.

Accordingly this file provides three classes:

- *ClusterAlgebra*
- *ClusterAlgebraSeed*
- *ClusterAlgebraElement*

ClusterAlgebra, constructed as a subobject of `sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_generic`, is the frontend of this implementation. It provides all the algebraic features (like ring morphisms), it computes cluster variables, it is responsible for controlling the exploration of the exchange graph and serves as the repository for all the data recursively computed so far. In particular, all g-vectors and all F-polynomials of known cluster variables as well as a mutation path by which they can be obtained are recorded. In the optic of efficiency, this implementation does not store directly the exchange graph nor the exchange relations. Both of these could be added to *ClusterAlgebra* with minimal effort.

ClusterAlgebraSeed provides the combinatorial backbone for *ClusterAlgebra*. It is an auxiliary class and therefore its instances should **not** be directly created by the user. Rather it should be accessed via *ClusterAlgebra.current_seed()* and *ClusterAlgebra.initial_seed()*. The task of performing current seed mutations is delegated to this class. Seeds are considered equal if they have the same parent cluster algebra and they can be obtained from each other by a permutation of their data (i.e. if they coincide as unlabelled seeds). Cluster algebras whose initial seeds are equal in the above sense are not considered equal but are endowed with coercion maps to each other. More generally, a cluster algebra is endowed with coercion maps from any cluster algebra which is obtained by freezing a collection of initial cluster variables and/or permuting both cluster variables and coefficients.

ClusterAlgebraElement is a thin wrapper around `sage.rings.polynomial.laurent_polynomial.LaurentPolynomial` providing all the functions specific to cluster variables. Elements of a cluster algebra with principal coefficients have special methods and these are grouped in the subclass *PrincipalClusterAlgebraElement*.

One more remark about this implementation. Instances of *ClusterAlgebra* are built by identifying the initial cluster variables with the generators of *ClusterAlgebra.ambient()*. In particular, this forces a specific embedding into the ambient field of rational expressions. In view of this, although cluster algebras themselves are independent of the choice of initial seed, *ClusterAlgebra.mutate_initial()* is forced to return a different instance of *ClusterAlgebra*. At the moment there is no coercion implemented among the two instances but this could in principle be added to *ClusterAlgebra.mutate_initial()*.

REFERENCES:

- [FZ2007]
- [LLZ2014]

- [NZ2012]

AUTHORS:

- Dylan Rupel (2015-06-15): initial version
- Salvatore Stella (2015-06-15): initial version

EXAMPLES:

We begin by creating a simple cluster algebra and printing its initial exchange matrix:

```
sage: A = ClusterAlgebra(['A', 2]); A
A Cluster Algebra with cluster variables x0, x1 and no coefficients over Integer Ring
sage: A.b_matrix()
[ 0  1]
[-1  0]
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)]); A
A Cluster Algebra with cluster variables x0, x1 and no coefficients over Integer Ring
>>> A.b_matrix()
[ 0  1]
[-1  0]
```

A is of finite type so we can explore all its exchange graph:

```
sage: A.explore_to_depth(infinity)
```

```
>>> from sage.all import *
>>> A.explore_to_depth(infinity)
```

and get all its g-vectors, F-polynomials, and cluster variables:

```
sage: sorted(A.g_vectors_so_far())
[(-1, 0), (-1, 1), (0, -1), (0, 1), (1, 0)]
sage: sorted(A.F_polynomials_so_far(), key=str)
[1, 1, u0 + 1, u0*u1 + u0 + 1, u1 + 1]
sage: sorted(A.cluster_variables_so_far(), key=str)
[(x0 + 1)/x1, (x0 + x1 + 1)/(x0*x1), (x1 + 1)/x0, x0, x1]
```

```
>>> from sage.all import *
>>> sorted(A.g_vectors_so_far())
[(-1, 0), (-1, 1), (0, -1), (0, 1), (1, 0)]
>>> sorted(A.F_polynomials_so_far(), key=str)
[1, 1, u0 + 1, u0*u1 + u0 + 1, u1 + 1]
>>> sorted(A.cluster_variables_so_far(), key=str)
[(x0 + 1)/x1, (x0 + x1 + 1)/(x0*x1), (x1 + 1)/x0, x0, x1]
```

Simple operations among cluster variables behave as expected:

```
sage: s = A.cluster_variable((0, -1)); s
(x0 + 1)/x1
sage: t = A.cluster_variable((-1, 1)); t
(x1 + 1)/x0
sage: t + s
```

(continues on next page)

(continued from previous page)

```

(x0^2 + x1^2 + x0 + x1)/(x0*x1)
sage: _.parent() == A
True
sage: t - s
(-x0^2 + x1^2 - x0 + x1)/(x0*x1)
sage: _.parent() == A
True
sage: t*s
(x0*x1 + x0 + x1 + 1)/(x0*x1)
sage: _.parent() == A
True
sage: t/s
(x1^2 + x1)/(x0^2 + x0)
sage: _.parent() == A
False

```

```

>>> from sage.all import *
>>> s = A.cluster_variable((Integer(0), -Integer(1))); s
(x0 + 1)/x1
>>> t = A.cluster_variable((-Integer(1), Integer(1))); t
(x1 + 1)/x0
>>> t + s
(x0^2 + x1^2 + x0 + x1)/(x0*x1)
>>> _.parent() == A
True
>>> t - s
(-x0^2 + x1^2 - x0 + x1)/(x0*x1)
>>> _.parent() == A
True
>>> t*s
(x0*x1 + x0 + x1 + 1)/(x0*x1)
>>> _.parent() == A
True
>>> t/s
(x1^2 + x1)/(x0^2 + x0)
>>> _.parent() == A
False

```

Division is not guaranteed to yield an element of A so it returns an element of $A.ambient().fraction\_field()$ instead:

```

sage: (t/s).parent() == A.ambient().fraction_field()
True

```

```

>>> from sage.all import *
>>> (t/s).parent() == A.ambient().fraction_field()
True

```

We can compute denominator vectors of any element of A :

```

sage: (t*s).d_vector()
(1, 1)

```

```
>>> from sage.all import *
>>> (t*s).d_vector()
(1, 1)
```

Since we are in rank 2 and we do not have coefficients we can compute the greedy element associated to any denominator vector:

```
sage: A.rank() == 2 and A.coefficients() == ()
True
sage: A.greedy_element((1, 1))
(x0 + x1 + 1)/(x0*x1)
sage: _ == t*s
False
```

```
>>> from sage.all import *
>>> A.rank() == Integer(2) and A.coefficients() == ()
True
>>> A.greedy_element((Integer(1), Integer(1)))
(x0 + x1 + 1)/(x0*x1)
>>> _ == t*s
False
```

not surprising since there is no cluster in A containing both t and s:

```
sage: seeds = A.seeds(mutating_F=false)
sage: [ S for S in seeds if (0, -1) in S and (-1, 1) in S ]
[]
```

```
>>> from sage.all import *
>>> seeds = A.seeds(mutating_F=false)
>>> [ S for S in seeds if (Integer(0), -Integer(1)) in S and (-Integer(1),
↪Integer(1)) in S ]
[]
```

indeed:

```
sage: A.greedy_element((1, 1)) == A.cluster_variable((-1, 0))
True
```

```
>>> from sage.all import *
>>> A.greedy_element((Integer(1), Integer(1))) == A.cluster_variable((-Integer(1),
↪Integer(0)))
True
```

Disabling F-polynomials in the computation just done was redundant because we already explored the whole exchange graph before. Though in different circumstances it could have saved us considerable time.

g-vectors and F-polynomials can be computed from elements of A only if A has principal coefficients at the initial seed:

```
sage: (t*s).g_vector()
Traceback (most recent call last):
...
AttributeError: 'ClusterAlgebra_with_category.element_class' object has no attribute
```

(continues on next page)

(continued from previous page)

```

↪ 'g_vector'...
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True)
sage: A.explore_to_depth(infinity)
sage: s = A.cluster_variable((0, -1)); s
(x0*y1 + 1)/x1
sage: t = A.cluster_variable((-1, 1)); t
(x1 + y0)/x0
sage: (t*s).g_vector()
(-1, 0)
sage: (t*s).F_polynomial()
u0*u1 + u0 + u1 + 1
sage: (t*s).is_homogeneous()
True
sage: (t+s).is_homogeneous()
False
sage: (t+s).homogeneous_components()
{(-1, 1): (x1 + y0)/x0, (0, -1): (x0*y1 + 1)/x1}

```

```

>>> from sage.all import *
>>> (t*s).g_vector()
Traceback (most recent call last):
...
AttributeError: 'ClusterAlgebra_with_category.element_class' object has no attribute
↪ 'g_vector'...
>>> A = ClusterAlgebra(['A', Integer(2)], principal_coefficients=True)
>>> A.explore_to_depth(infinity)
>>> s = A.cluster_variable((Integer(0), -Integer(1))); s
(x0*y1 + 1)/x1
>>> t = A.cluster_variable((-Integer(1), Integer(1))); t
(x1 + y0)/x0
>>> (t*s).g_vector()
(-1, 0)
>>> (t*s).F_polynomial()
u0*u1 + u0 + u1 + 1
>>> (t*s).is_homogeneous()
True
>>> (t+s).is_homogeneous()
False
>>> (t+s).homogeneous_components()
{(-1, 1): (x1 + y0)/x0, (0, -1): (x0*y1 + 1)/x1}

```

Each cluster algebra is endowed with a reference to a current seed; it could be useful to assign a name to it:

```

sage: A = ClusterAlgebra(['F', 4])
sage: len(A.g_vectors_so_far())
4
sage: A.current_seed()
The initial seed of a Cluster Algebra with cluster variables x0, x1, x2, x3
and no coefficients over Integer Ring
sage: A.current_seed() == A.initial_seed()
True
sage: S = A.current_seed()

```

(continues on next page)

(continued from previous page)

```

sage: S.b_matrix()
[ 0  1  0  0]
[-1  0 -1  0]
[ 0  2  0  1]
[ 0  0 -1  0]
sage: S.g_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
sage: S.cluster_variables()
[x0, x1, x2, x3]

```

```

>>> from sage.all import *
>>> A = ClusterAlgebra(['F', Integer(4)])
>>> len(A.g_vectors_so_far())
4
>>> A.current_seed()
The initial seed of a Cluster Algebra with cluster variables x0, x1, x2, x3
and no coefficients over Integer Ring
>>> A.current_seed() == A.initial_seed()
True
>>> S = A.current_seed()
>>> S.b_matrix()
[ 0  1  0  0]
[-1  0 -1  0]
[ 0  2  0  1]
[ 0  0 -1  0]
>>> S.g_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
>>> S.cluster_variables()
[x0, x1, x2, x3]

```

and use `S` to walk around the exchange graph of `A`:

```

sage: S.mutate(0); S
The seed of a Cluster Algebra with cluster variables x0, x1, x2, x3
and no coefficients over Integer Ring obtained from the initial
by mutating in direction 0
sage: S.b_matrix()
[ 0 -1  0  0]
[ 1  0 -1  0]
[ 0  2  0  1]
[ 0  0 -1  0]
sage: S.g_matrix()
[-1  0  0  0]
[ 1  1  0  0]
[ 0  0  1  0]
[ 0  0  0  1]

```

(continues on next page)

(continued from previous page)

```

sage: S.cluster_variables()
[(x1 + 1)/x0, x1, x2, x3]
sage: S.mutate('sinks'); S
The seed of a Cluster Algebra with cluster variables x0, x1, x2, x3
and no coefficients over Integer Ring obtained from the initial
by mutating along the sequence [0, 2]
sage: S.mutate([2, 3, 2, 1, 0]); S
The seed of a Cluster Algebra with cluster variables x0, x1, x2, x3
and no coefficients over Integer Ring obtained from the initial
by mutating along the sequence [0, 3, 2, 1, 0]
sage: S.g_vectors()
[(0, 1, -2, 0), (-1, 2, -2, 0), (0, 1, -1, 0), (0, 0, 0, -1)]
sage: S.cluster_variable(3)
(x2 + 1)/x3

```

```

>>> from sage.all import *
>>> S.mutate(Integer(0)); S
The seed of a Cluster Algebra with cluster variables x0, x1, x2, x3
and no coefficients over Integer Ring obtained from the initial
by mutating in direction 0
>>> S.b_matrix()
[ 0 -1  0  0]
[ 1  0 -1  0]
[ 0  2  0  1]
[ 0  0 -1  0]
>>> S.g_matrix()
[-1  0  0  0]
[ 1  1  0  0]
[ 0  0  1  0]
[ 0  0  0  1]
>>> S.cluster_variables()
[(x1 + 1)/x0, x1, x2, x3]
>>> S.mutate('sinks'); S
The seed of a Cluster Algebra with cluster variables x0, x1, x2, x3
and no coefficients over Integer Ring obtained from the initial
by mutating along the sequence [0, 2]
>>> S.mutate([Integer(2), Integer(3), Integer(2), Integer(1), Integer(0)]); S
The seed of a Cluster Algebra with cluster variables x0, x1, x2, x3
and no coefficients over Integer Ring obtained from the initial
by mutating along the sequence [0, 3, 2, 1, 0]
>>> S.g_vectors()
[(0, 1, -2, 0), (-1, 2, -2, 0), (0, 1, -1, 0), (0, 0, 0, -1)]
>>> S.cluster_variable(Integer(3))
(x2 + 1)/x3

```

Walking around by mutating S updates the information stored in A :

```

sage: len(A.g_vectors_so_far())
10
sage: A.current_seed().path_from_initial_seed()
[0, 3, 2, 1, 0]
sage: A.current_seed() == S

```

(continues on next page)

(continued from previous page)

```
True
```

```
>>> from sage.all import *
>>> len(A.g_vectors_so_far())
10
>>> A.current_seed().path_from_initial_seed()
[0, 3, 2, 1, 0]
>>> A.current_seed() == S
True
```

Starting from `A.initial_seed()` still records data in `A` but does not update `A.current_seed()`:

```
sage: S1 = A.initial_seed()
sage: S1.mutate([2, 1, 3])
sage: len(A.g_vectors_so_far())
11
sage: S1 == A.current_seed()
False
```

```
>>> from sage.all import *
>>> S1 = A.initial_seed()
>>> S1.mutate([Integer(2), Integer(1), Integer(3)])
>>> len(A.g_vectors_so_far())
11
>>> S1 == A.current_seed()
False
```

Since `ClusterAlgebra` inherits from `UniqueRepresentation`, computed data is shared across instances:

```
sage: A1 = ClusterAlgebra(['F', 4])
sage: A1 is A
True
sage: len(A1.g_vectors_so_far())
11
```

```
>>> from sage.all import *
>>> A1 = ClusterAlgebra(['F', Integer(4)])
>>> A1 is A
True
>>> len(A1.g_vectors_so_far())
11
```

It can be useful, at times to forget all computed data. Because of `UniqueRepresentation` this cannot be achieved by simply creating a new instance; instead it has to be manually triggered by:

```
sage: A.clear_computed_data()
sage: len(A.g_vectors_so_far())
4
```

```
>>> from sage.all import *
>>> A.clear_computed_data()
```

(continues on next page)

(continued from previous page)

```
>>> len(A.g_vectors_so_far())
4
```

Given a cluster algebra A we may be looking for a specific cluster variable:

```
sage: A = ClusterAlgebra(['E', 8, 1])
sage: v = (-1, 1, -1, 1, -1, 1, 0, 0, 1)
sage: A.find_g_vector(v, depth=2)
sage: seq = A.find_g_vector(v); seq # random
[0, 1, 2, 4, 3]
sage: v in A.initial_seed().mutate(seq, inplace=False).g_vectors()
True
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['E', Integer(8), Integer(1)])
>>> v = (-Integer(1), Integer(1), -Integer(1), Integer(1), -Integer(1), Integer(1),
↪ Integer(0), Integer(0), Integer(1))
>>> A.find_g_vector(v, depth=Integer(2))
>>> seq = A.find_g_vector(v); seq # random
[0, 1, 2, 4, 3]
>>> v in A.initial_seed().mutate(seq, inplace=False).g_vectors()
True
```

This also performs mutations of F-polynomials:

```
sage: A.F_polynomial((-1, 1, -1, 1, -1, 1, 0, 0, 1))
u0*u1*u2*u3*u4 + u0*u1*u2*u4 + u0*u2*u3*u4 + u0*u1*u2 + u0*u2*u4
+ u2*u3*u4 + u0*u2 + u0*u4 + u2*u4 + u0 + u2 + u4 + 1
```

```
>>> from sage.all import *
>>> A.F_polynomial((-Integer(1), Integer(1), -Integer(1), Integer(1), -Integer(1),
↪ Integer(1), Integer(0), Integer(0), Integer(1)))
u0*u1*u2*u3*u4 + u0*u1*u2*u4 + u0*u2*u3*u4 + u0*u1*u2 + u0*u2*u4
+ u2*u3*u4 + u0*u2 + u0*u4 + u2*u4 + u0 + u2 + u4 + 1
```

which might not be a good idea in algebras that are too big. One workaround is to first disable F-polynomials and then recompute only the desired mutations:

```
sage: # long time
sage: A.reset_exploring_iterator(mutating_F=False)
sage: v = (-1, 1, -2, 2, -1, 1, -1, 1, 1)
sage: seq = A.find_g_vector(v); seq # random
[1, 0, 2, 6, 5, 4, 3, 8, 1]
sage: S = A.initial_seed().mutate(seq, inplace=False)
sage: v in S.g_vectors()
True
sage: A.current_seed().mutate(seq)
sage: A.F_polynomial((-1, 1, -2, 2, -1, 1, -1, 1, 1))
u0*u1^2*u2^2*u3*u4*u5*u6*u8 +
...
2*u2 + u4 + u6 + 1
```

```

>>> from sage.all import *
>>> # long time
>>> A.reset_exploring_iterator(mutating_F=False)
>>> v = (-Integer(1), Integer(1), -Integer(2), Integer(2), -Integer(1), Integer(1), -
↳ Integer(1), Integer(1), Integer(1))
>>> seq = A.find_g_vector(v); seq # random
[1, 0, 2, 6, 5, 4, 3, 8, 1]
>>> S = A.initial_seed().mutate(seq, inplace=False)
>>> v in S.g_vectors()
True
>>> A.current_seed().mutate(seq)
>>> A.F_polynomial((-Integer(1), Integer(1), -Integer(2), Integer(2), -Integer(1), -
↳ Integer(1), -Integer(1), Integer(1), Integer(1)))
u0*u1^2*u2^2*u3*u4*u5*u6*u8 +
...
2*u2 + u4 + u6 + 1

```

We can manually freeze cluster variables and get coercions in between the two algebras:

```

sage: A = ClusterAlgebra(['F', 4]); A
A Cluster Algebra with cluster variables x0, x1, x2, x3 and no coefficients
over Integer Ring
sage: A1 = ClusterAlgebra(A.b_matrix().matrix_from_columns([0, 1, 2]), coefficient_
↳ prefix='x'); A1
A Cluster Algebra with cluster variables x0, x1, x2 and coefficient x3
over Integer Ring
sage: A.has_coerce_map_from(A1)
True

```

```

>>> from sage.all import *
>>> A = ClusterAlgebra(['F', Integer(4)]); A
A Cluster Algebra with cluster variables x0, x1, x2, x3 and no coefficients
over Integer Ring
>>> A1 = ClusterAlgebra(A.b_matrix().matrix_from_columns([Integer(0), Integer(1), -
↳ Integer(2)]), coefficient_prefix='x'); A1
A Cluster Algebra with cluster variables x0, x1, x2 and coefficient x3
over Integer Ring
>>> A.has_coerce_map_from(A1)
True

```

and we also have an immersion of $A.\text{base}()$ into A and of A into $A.\text{ambient}()$:

```

sage: A.has_coerce_map_from(A.base())
True
sage: A.ambient().has_coerce_map_from(A)
True

```

```

>>> from sage.all import *
>>> A.has_coerce_map_from(A.base())
True
>>> A.ambient().has_coerce_map_from(A)
True

```

but there is currently no coercion in between algebras obtained by mutating at the initial seed:

```
sage: A1 = A.mutate_initial(0); A1
A Cluster Algebra with cluster variables x4, x1, x2, x3 and no coefficients
over Integer Ring
sage: A.b_matrix() == A1.b_matrix()
False
sage: [X.has_coerce_map_from(Y) for X, Y in [(A, A1), (A1, A)]]
[False, False]
```

```
>>> from sage.all import *
>>> A1 = A.mutate_initial(Integer(0)); A1
A Cluster Algebra with cluster variables x4, x1, x2, x3 and no coefficients
over Integer Ring
>>> A.b_matrix() == A1.b_matrix()
False
>>> [X.has_coerce_map_from(Y) for X, Y in [(A, A1), (A1, A)]]
[False, False]
```

```
class sage.algebras.cluster_algebra.ClusterAlgebra(B, **kwargs)
```

Bases: `Parent`, `UniqueRepresentation`

A Cluster Algebra.

INPUT:

- `data` – some data defining a cluster algebra; it can be anything that can be parsed by `ClusterQuiver`
- `scalars` – a ring (default: $\mathbb{Z}$); the scalars over which the cluster algebra is defined
- `cluster_variable_prefix` – string (default: 'x'); it needs to be a valid variable name
- `cluster_variable_names` – list of strings; each element needs to be a valid variable name; supersedes `cluster_variable_prefix`
- `coefficient_prefix` – string (default: 'y'); it needs to be a valid variable name
- `coefficient_names` – list of strings; each element needs to be a valid variable name; supersedes `cluster_variable_prefix`
- `principal_coefficients` – boolean (default: False); supersedes any coefficient defined by `data`

ALGORITHM:

The implementation is mainly based on [FZ2007] and [NZ2012].

EXAMPLES:

```
sage: B = matrix([(0, 1, 0, 0), (-1, 0, -1, 0), (0, 1, 0, 1), (0, 0, -2, 0), (-1, 0, 0, 0), (0, -1, 0, 0)])
sage: A = ClusterAlgebra(B); A
A Cluster Algebra with cluster variables x0, x1, x2, x3
and coefficients y0, y1 over Integer Ring
sage: A.gens()
(x0, x1, x2, x3, y0, y1)
sage: A = ClusterAlgebra(['A', 2]); A
A Cluster Algebra with cluster variables x0, x1 and no coefficients
over Integer Ring
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True); A.gens()
```

(continues on next page)

(continued from previous page)

```

(x0, x1, y0, y1)
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True, coefficient_
↳ prefix='x'); A.gens()
(x0, x1, x2, x3)
sage: A = ClusterAlgebra(['A', 3], principal_coefficients=True, cluster_variable_
↳ names=['a', 'b', 'c']); A.gens()
(a, b, c, y0, y1, y2)
sage: A = ClusterAlgebra(['A', 3], principal_coefficients=True, cluster_variable_
↳ names=['a', 'b'])
Traceback (most recent call last):
...
ValueError: cluster_variable_names should be an iterable of 3 valid variable names
sage: A = ClusterAlgebra(['A', 3], principal_coefficients=True, coefficient_
↳ names=['a', 'b', 'c']); A.gens()
(x0, x1, x2, a, b, c)
sage: A = ClusterAlgebra(['A', 3], principal_coefficients=True, coefficient_
↳ names=['a', 'b'])
Traceback (most recent call last):
...
ValueError: coefficient_names should be an iterable of 3 valid variable names

```

```

>>> from sage.all import *
>>> B = matrix([(Integer(0), Integer(1), Integer(0), Integer(0)), (-Integer(1),
↳ Integer(0), -Integer(1), Integer(0)), (Integer(0), Integer(1), Integer(0),
↳ Integer(1)), (Integer(0), Integer(0), -Integer(2), Integer(0)), (-Integer(1),
↳ Integer(0), Integer(0), Integer(0)), (Integer(0), -Integer(1), Integer(0),
↳ Integer(0))])
>>> A = ClusterAlgebra(B); A
A Cluster Algebra with cluster variables x0, x1, x2, x3
and coefficients y0, y1 over Integer Ring
>>> A.gens()
(x0, x1, x2, x3, y0, y1)
>>> A = ClusterAlgebra(['A', Integer(2)]); A
A Cluster Algebra with cluster variables x0, x1 and no coefficients
over Integer Ring
>>> A = ClusterAlgebra(['A', Integer(2)], principal_coefficients=True); A.gens()
(x0, x1, y0, y1)
>>> A = ClusterAlgebra(['A', Integer(2)], principal_coefficients=True,
↳ coefficient_prefix='x'); A.gens()
(x0, x1, x2, x3)
>>> A = ClusterAlgebra(['A', Integer(3)], principal_coefficients=True, cluster_
↳ variable_names=['a', 'b', 'c']); A.gens()
(a, b, c, y0, y1, y2)
>>> A = ClusterAlgebra(['A', Integer(3)], principal_coefficients=True, cluster_
↳ variable_names=['a', 'b'])
Traceback (most recent call last):
...
ValueError: cluster_variable_names should be an iterable of 3 valid variable names
>>> A = ClusterAlgebra(['A', Integer(3)], principal_coefficients=True,
↳ coefficient_names=['a', 'b', 'c']); A.gens()
(x0, x1, x2, a, b, c)
>>> A = ClusterAlgebra(['A', Integer(3)], principal_coefficients=True,

```

(continues on next page)

(continued from previous page)

```

↪coefficient_names=['a', 'b'])
Traceback (most recent call last):
...
ValueError: coefficient_names should be an iterable of 3 valid variable names

```

F_polynomial(g_vector)

Return the F-polynomial with g-vector g_vector if it has been found.

INPUT:

- g_vector – tuple; the g-vector of the F-polynomial to return

EXAMPLES:

```

sage: A = ClusterAlgebra(['A', 2])
sage: A.clear_computed_data()
sage: A.F_polynomial((-1, 1))
Traceback (most recent call last):
...
KeyError: 'the g-vector (-1, 1) has not been found yet'
sage: A.initial_seed().mutate(0, mutating_F=False)
sage: A.F_polynomial((-1, 1))
Traceback (most recent call last):
...
KeyError: 'the F-polynomial with g-vector (-1, 1) has not been computed yet;
you can compute it by mutating from the initial seed along the sequence [0]'
sage: A.initial_seed().mutate(0)
sage: A.F_polynomial((-1, 1))
u0 + 1

```

```

>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)])
>>> A.clear_computed_data()
>>> A.F_polynomial((-Integer(1), Integer(1)))
Traceback (most recent call last):
...
KeyError: 'the g-vector (-1, 1) has not been found yet'
>>> A.initial_seed().mutate(Integer(0), mutating_F=False)
>>> A.F_polynomial((-Integer(1), Integer(1)))
Traceback (most recent call last):
...
KeyError: 'the F-polynomial with g-vector (-1, 1) has not been computed yet;
you can compute it by mutating from the initial seed along the sequence [0]'
>>> A.initial_seed().mutate(Integer(0))
>>> A.F_polynomial((-Integer(1), Integer(1)))
u0 + 1

```

F_polynomials()

Return an iterator producing all the F-polynomials of self.

ALGORITHM:

This method does not use the caching framework provided by self, but recomputes all the F-polynomials from scratch. On the other hand it stores the results so that other methods like `F_polynomials_so_far()` can access them afterwards.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: len(list(A.F_polynomials()))
9
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(3)])
>>> len(list(A.F_polynomials()))
9
```

F_polynomials_so_far()

Return a list of the F-polynomials encountered so far.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2])
sage: A.clear_computed_data()
sage: A.current_seed().mutate(0)
sage: sorted(A.F_polynomials_so_far(), key=str)
[1, 1, u0 + 1]
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)])
>>> A.clear_computed_data()
>>> A.current_seed().mutate(Integer(0))
>>> sorted(A.F_polynomials_so_far(), key=str)
[1, 1, u0 + 1]
```

ambient()

Return the Laurent polynomial ring containing self.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True)
sage: A.ambient()
Multivariate Laurent Polynomial Ring in x0, x1, y0, y1 over Integer Ring
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)], principal_coefficients=True)
>>> A.ambient()
Multivariate Laurent Polynomial Ring in x0, x1, y0, y1 over Integer Ring
```

b_matrix()

Return the initial exchange matrix of self.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2])
sage: A.b_matrix()
[ 0  1]
[-1  0]
```



```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)])
>>> A.b_matrix()
[ 0  1]
[-1  0]
```

clear_computed_data()

Clear the cache of computed g-vectors and F-polynomials and reset both the current seed and the exploring iterator.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2])
sage: A.clear_computed_data()
sage: sorted(A.g_vectors_so_far())
[(0, 1), (1, 0)]
sage: A.current_seed().mutate([1, 0])
sage: sorted(A.g_vectors_so_far())
[(-1, 0), (0, -1), (0, 1), (1, 0)]
sage: A.clear_computed_data()
sage: sorted(A.g_vectors_so_far())
[(0, 1), (1, 0)]
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)])
>>> A.clear_computed_data()
>>> sorted(A.g_vectors_so_far())
[(0, 1), (1, 0)]
>>> A.current_seed().mutate([Integer(1), Integer(0)])
>>> sorted(A.g_vectors_so_far())
[(-1, 0), (0, -1), (0, 1), (1, 0)]
>>> A.clear_computed_data()
>>> sorted(A.g_vectors_so_far())
[(0, 1), (1, 0)]
```

cluster_fan(depth=+Infinity)

Return the cluster fan (the fan of g-vectors) of self.

INPUT:

- depth – positive integer or infinity (default: infinity); the maximum depth at which to compute

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2])
sage: A.cluster_fan()
↳needs sage.geometry.polyhedron
Rational polyhedral fan in 2-d lattice N
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)])
>>> A.cluster_fan()
↳needs sage.geometry.polyhedron
Rational polyhedral fan in 2-d lattice N
```

cluster_variable(*g_vector*)

Return the cluster variable with g-vector *g_vector* if it has been found.

INPUT:

- *g_vector* – tuple; the g-vector of the cluster variable to return

ALGORITHM:

This function computes cluster variables from their g-vectors and F-polynomials using the “separation of additions” formula of Theorem 3.7 in [FZ2007].

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2])
sage: A.initial_seed().mutate(0)
sage: A.cluster_variable((-1, 1))
(x1 + 1)/x0
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)])
>>> A.initial_seed().mutate(Integer(0))
>>> A.cluster_variable((-Integer(1), Integer(1)))
(x1 + 1)/x0
```

cluster_variables()

Return an iterator producing all the cluster variables of *self*.

ALGORITHM:

This method does not use the caching framework provided by *self*, but recomputes all the cluster variables from scratch. On the other hand it stores the results so that other methods like *cluster_variables_so_far()* can access them afterwards.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: len(list(A.cluster_variables()))
9
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(3)])
>>> len(list(A.cluster_variables()))
9
```

cluster_variables_so_far()

Return a list of the cluster variables encountered so far.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2])
sage: A.clear_computed_data()
sage: A.current_seed().mutate(0)
sage: sorted(A.cluster_variables_so_far(), key=str)
[(x1 + 1)/x0, x0, x1]
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)])
>>> A.clear_computed_data()
>>> A.current_seed().mutate(Integer(0))
>>> sorted(A.cluster_variables_so_far(), key=str)
[(x1 + 1)/x0, x0, x1]
```

coefficient(j)

Return the j -th coefficient of `self`.

INPUT:

- j – integer in `range(self.parent().rank())`; the index of the coefficient to return

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True)
sage: A.coefficient(0)
y0
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)], principal_coefficients=True)
>>> A.coefficient(Integer(0))
y0
```

coefficient_names()

Return the list of coefficient names.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: A.coefficient_names()
()
sage: A1 = ClusterAlgebra(['B', 2], principal_coefficients=True)
sage: A1.coefficient_names()
('y0', 'y1')
sage: A2 = ClusterAlgebra(['C', 3], principal_coefficients=True, coefficient_
↪ prefix='x')
sage: A2.coefficient_names()
('x3', 'x4', 'x5')
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(3)])
>>> A.coefficient_names()
()
>>> A1 = ClusterAlgebra(['B', Integer(2)], principal_coefficients=True)
>>> A1.coefficient_names()
('y0', 'y1')
>>> A2 = ClusterAlgebra(['C', Integer(3)], principal_coefficients=True, ↪
↪ coefficient_prefix='x')
>>> A2.coefficient_names()
('x3', 'x4', 'x5')
```

coefficients()

Return the list of coefficients of `self`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True)
sage: A.coefficients()
(y0, y1)
sage: A1 = ClusterAlgebra(['B', 2])
sage: A1.coefficients()
()
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)], principal_coefficients=True)
>>> A.coefficients()
(y0, y1)
>>> A1 = ClusterAlgebra(['B', Integer(2)])
>>> A1.coefficients()
()
```

contains_seed(seed)

Test if seed is a seed of self.

INPUT:

- seed – a *ClusterAlgebraSeed*

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True); A
A Cluster Algebra with cluster variables x0, x1 and coefficients y0, y1 over
↳ Integer Ring
sage: S = copy(A.current_seed())
sage: A.contains_seed(S)
True
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)], principal_coefficients=True); A
A Cluster Algebra with cluster variables x0, x1 and coefficients y0, y1 over
↳ Integer Ring
>>> S = copy(A.current_seed())
>>> A.contains_seed(S)
True
```

coxeter_element()

Return the Coxeter element associated to the initial exchange matrix, if acyclic.

EXAMPLES:

```
sage: A = ClusterAlgebra(matrix([[0, 1, 1], [-1, 0, 1], [-1, -1, 0]]))
sage: A.coxeter_element()
[0, 1, 2]
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(matrix([[Integer(0), Integer(1), Integer(1)], [-
↳ Integer(1), Integer(0), Integer(1)], [-Integer(1), -Integer(1), Integer(0)]]))
>>> A.coxeter_element()
[0, 1, 2]
```

Raise an error if the initial exchange matrix is not acyclic:

```
sage: A = ClusterAlgebra(matrix([[0,1,-1],[-1,0,1],[1,-1,0]]))
sage: A.coxeter_element()
Traceback (most recent call last):
...
ValueError: the initial exchange matrix is not acyclic
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(matrix([[Integer(0),Integer(1),-Integer(1)],[-
↪Integer(1),Integer(0),Integer(1)], [Integer(1),-Integer(1),Integer(0)]]))
>>> A.coxeter_element()
Traceback (most recent call last):
...
ValueError: the initial exchange matrix is not acyclic
```

current_seed()

Return the current seed of self.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2])
sage: A.clear_computed_data()
sage: A.current_seed()
The initial seed of a Cluster Algebra with cluster variables x0, x1
and no coefficients over Integer Ring
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)])
>>> A.clear_computed_data()
>>> A.current_seed()
The initial seed of a Cluster Algebra with cluster variables x0, x1
and no coefficients over Integer Ring
```

d_vector_to_g_vector(d)

Return the g-vector of an element of self having d-vector d

INPUT:

- d – the d-vector

ALGORITHM:

This method implements the piecewise-linear map nu_c introduced in Section 9.1 of [ReSt2020].

EXAMPLES:

```
sage: A = ClusterAlgebra(matrix([[0,1,1],[-1,0,1],[-1,-1,0]]))
sage: A.d_vector_to_g_vector((1,0,-1))
(-1, 1, 2)
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(matrix([[Integer(0),Integer(1),Integer(1)],[-
↪Integer(1),Integer(0),Integer(1)], [-Integer(1),-Integer(1),Integer(0)]]))
```

(continues on next page)

(continued from previous page)

```
>>> A.d_vector_to_g_vector((Integer(1), Integer(0), -Integer(1)))
(-1, 1, 2)
```

euler_matrix()

Return the Euler matrix associated to `self`.

ALGORITHM:

This method returns the matrix of the bilinear form defined in Equation (2.1) of [ReSt2020].

EXAMPLES:

```
sage: A = ClusterAlgebra(matrix([[0, 1, 1], [-1, 0, 1], [-1, -1, 0]]))
sage: A.euler_matrix()
[ 1  0  0]
[-1  1  0]
[-1 -1  1]
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(matrix([[Integer(0), Integer(1), Integer(1)], [-
↪ Integer(1), Integer(0), Integer(1)], [-Integer(1), -Integer(1), Integer(0)]]))
>>> A.euler_matrix()
[ 1  0  0]
[-1  1  0]
[-1 -1  1]
```

Raise an error if the initial exchange matrix is not acyclic:

```
sage: A = ClusterAlgebra(matrix([[0, 1, -1], [-1, 0, 1], [1, -1, 0]]))
sage: A.euler_matrix()
Traceback (most recent call last):
...
ValueError: the initial exchange matrix is not acyclic
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(matrix([[Integer(0), Integer(1), -Integer(1)], [-
↪ Integer(1), Integer(0), Integer(1)], [Integer(1), -Integer(1), Integer(0)]]))
>>> A.euler_matrix()
Traceback (most recent call last):
...
ValueError: the initial exchange matrix is not acyclic
```

explore_to_depth(depth)

Explore the exchange graph of `self` up to distance `depth` from the initial seed.

INPUT:

- `depth` – positive integer or infinity; the maximum depth at which to stop searching

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 4])
sage: A.explore_to_depth(infinity)
sage: len(A.g_vectors_so_far())
14
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(4)])
>>> A.explore_to_depth(infinity)
>>> len(A.g_vectors_so_far())
14
```

find_g_vector(*g_vector*, *depth*=+Infinity)

Return a mutation sequence to obtain a seed containing the g-vector *g_vector* from the initial seed.

INPUT:

- *g_vector* – tuple; the g-vector to find
- *depth* – positive integer or infinity (default: `infinity`); the maximum distance from `self.current_seed` to reach

OUTPUT:

This function returns a list of integers if it can find *g_vector*, otherwise it returns `None`. If the exploring iterator stops, it means that the algebra is of finite type and *g_vector* is not the g-vector of any cluster variable. In this case the function resets the iterator and raises an error.

EXAMPLES:

```
sage: A = ClusterAlgebra(['G', 2], principal_coefficients=True)
sage: A.clear_computed_data()
sage: A.find_g_vector((-2, 3), depth=2)
sage: A.find_g_vector((-2, 3), depth=3)
[0, 1, 0]
sage: A.find_g_vector((1, 1), depth=3)
sage: A.find_g_vector((1, 1), depth=4)
Traceback (most recent call last):
...
ValueError: (1, 1) is not the g-vector of any cluster variable of a
Cluster Algebra with cluster variables x0, x1 and coefficients y0, y1
over Integer Ring
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['G', Integer(2)], principal_coefficients=True)
>>> A.clear_computed_data()
>>> A.find_g_vector((-Integer(2), Integer(3)), depth=Integer(2))
>>> A.find_g_vector((-Integer(2), Integer(3)), depth=Integer(3))
[0, 1, 0]
>>> A.find_g_vector((Integer(1), Integer(1)), depth=Integer(3))
>>> A.find_g_vector((Integer(1), Integer(1)), depth=Integer(4))
Traceback (most recent call last):
...
ValueError: (1, 1) is not the g-vector of any cluster variable of a
Cluster Algebra with cluster variables x0, x1 and coefficients y0, y1
over Integer Ring
```

g_vector_to_d_vector(*g*)

Return the d-vector of an element of `self` having g-vector *g*

INPUT:

- *g* – the g-vector

ALGORITHM:

This method implements the inverse of the piecewise-linear map nu_c introduced in Section 9.1 of [ReSt2020].

EXAMPLES:

```
sage: A = ClusterAlgebra(matrix([[0, 1, 1], [-1, 0, 1], [-1, -1, 0]]))
sage: A.g_vector_to_d_vector((-1, 1, 2))
(1, 0, -1)
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(matrix([[Integer(0), Integer(1), Integer(1)], [-
↪ Integer(1), Integer(0), Integer(1)], [-Integer(1), -Integer(1), Integer(0)]]))
>>> A.g_vector_to_d_vector((-Integer(1), Integer(1), Integer(2)))
(1, 0, -1)
```

g_vectors (*mutating_F=True*)

Return an iterator producing all the g-vectors of *self*.

INPUT:

- *mutating_F* – boolean (default: *True*); whether to compute F-polynomials; disable this for speed considerations

ALGORITHM:

This method does not use the caching framework provided by *self*, but recomputes all the g-vectors from scratch. On the other hand it stores the results so that other methods like *g_vectors_so_far()* can access them afterwards.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: len(list(A.g_vectors()))
9
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(3)])
>>> len(list(A.g_vectors()))
9
```

g_vectors_so_far()

Return a list of the g-vectors of cluster variables encountered so far.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2])
sage: A.clear_computed_data()
sage: A.current_seed().mutate(0)
sage: sorted(A.g_vectors_so_far())
[(-1, 1), (0, 1), (1, 0)]
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)])
>>> A.clear_computed_data()
```

(continues on next page)

(continued from previous page)

```
>>> A.current_seed().mutate(Integer(0))
>>> sorted(A.g_vectors_so_far())
[(-1, 1), (0, 1), (1, 0)]
```

gens()

Return the list of initial cluster variables and coefficients of `self`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True)
sage: A.gens()
(x0, x1, y0, y1)
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True, coefficient_
↳ prefix='x')
sage: A.gens()
(x0, x1, x2, x3)
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)], principal_coefficients=True)
>>> A.gens()
(x0, x1, y0, y1)
>>> A = ClusterAlgebra(['A', Integer(2)], principal_coefficients=True,
↳ coefficient_prefix='x')
>>> A.gens()
(x0, x1, x2, x3)
```

greedy_element(d_vector)

Return the greedy element with denominator vector `d_vector`.

INPUT:

- `d_vector` – tuple of 2 integers; the denominator vector of the element to compute

ALGORITHM:

This implements greedy elements of a rank 2 cluster algebra using Equation (1.5) from [LLZ2014].

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', [1, 1], 1])
sage: A.greedy_element((1, 1))
(x0^2 + x1^2 + 1)/(x0*x1)
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', [Integer(1), Integer(1)], Integer(1)])
>>> A.greedy_element((Integer(1), Integer(1)))
(x0^2 + x1^2 + 1)/(x0*x1)
```

initial_cluster_variable(j)

Return the `j`-th initial cluster variable of `self`.

INPUT:

- `j` – integer in `range(self.parent().rank())`; the index of the cluster variable to return

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True)
sage: A.initial_cluster_variable(0)
x0
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)], principal_coefficients=True)
>>> A.initial_cluster_variable(Integer(0))
x0
```

initial_cluster_variable_names()

Return the list of initial cluster variable names.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True)
sage: A.initial_cluster_variable_names()
('x0', 'x1')
sage: A1 = ClusterAlgebra(['B', 2], cluster_variable_prefix='a')
sage: A1.initial_cluster_variable_names()
('a0', 'a1')
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)], principal_coefficients=True)
>>> A.initial_cluster_variable_names()
('x0', 'x1')
>>> A1 = ClusterAlgebra(['B', Integer(2)], cluster_variable_prefix='a')
>>> A1.initial_cluster_variable_names()
('a0', 'a1')
```

initial_cluster_variables()

Return the list of initial cluster variables of self.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True)
sage: A.initial_cluster_variables()
(x0, x1)
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)], principal_coefficients=True)
>>> A.initial_cluster_variables()
(x0, x1)
```

initial_seed()

Return the initial seed of self.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2])
sage: A.initial_seed()
The initial seed of a Cluster Algebra with cluster variables x0, x1 and no_
↪coefficients over Integer Ring
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)])
>>> A.initial_seed()
The initial seed of a Cluster Algebra with cluster variables x0, x1 and no
↪coefficients over Integer Ring
```

is_acyclic()

Return True if the exchange matrix in the initial seed is acyclic, False otherwise.

EXAMPLES:

```
sage: A = ClusterAlgebra(matrix([[0,1,1],[-1,0,1],[-1,-1,0]]))
sage: A.is_acyclic()
True
sage: A = ClusterAlgebra(matrix([[0,1,-1],[-1,0,1],[1,-1,0]]))
sage: A.is_acyclic()
False
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(matrix([[Integer(0),Integer(1),Integer(1)],[-
↪Integer(1),Integer(0),Integer(1)],[-Integer(1),-Integer(1),Integer(0)]]))
>>> A.is_acyclic()
True
>>> A = ClusterAlgebra(matrix([[Integer(0),Integer(1),-Integer(1)],[-
↪Integer(1),Integer(0),Integer(1)],[Integer(1),-Integer(1),Integer(0)]]))
>>> A.is_acyclic()
False
```

lift(x)

Return x as an element of *ambient()*.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True)
sage: x = A.cluster_variable((1, 0))
sage: A.lift(x).parent()
Multivariate Laurent Polynomial Ring in x0, x1, y0, y1 over Integer Ring
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)], principal_coefficients=True)
>>> x = A.cluster_variable((Integer(1), Integer(0)))
>>> A.lift(x).parent()
Multivariate Laurent Polynomial Ring in x0, x1, y0, y1 over Integer Ring
```

lower_bound()

Return the lower bound associated to self.

EXAMPLES:

```
sage: A = ClusterAlgebra(['F', 4])
sage: A.lower_bound()
Traceback (most recent call last):
...
NotImplementedError: not implemented yet
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['F', Integer(4)])
>>> A.lower_bound()
Traceback (most recent call last):
...
NotImplementedError: not implemented yet
```

mutate_initial(*direction*, ***kwargs*)

Return the cluster algebra obtained by mutating *self* at the initial seed.

Warning

This method is significantly slower than `ClusterAlgebraSeed.mutate()`. It is therefore advisable to use the latter for exploration purposes.

INPUT:

- *direction* – in which direction(s) to mutate, it can be:
 - an integer in `range(self.rank())` to mutate in one direction only
 - an iterable of such integers to mutate along a sequence
 - a string “sinks” or “sources” to mutate at all sinks or sources simultaneously
- *mutating_F* – boolean (default: `True`); whether to compute F-polynomials while mutating

Note

While knowing F-polynomials is essential to computing cluster variables, the process of mutating them is quite slow. If you care only about combinatorial data like g-vectors and c-vectors, setting `mutating_F=False` yields significant benefits in terms of speed.

ALGORITHM:

This function computes data for the new algebra from known data for the old algebra using Equation (4.2) from [NZ2012] for g-vectors, and Equation (6.21) from [FZ2007] for F-polynomials. The exponent h in the formula for F-polynomials is $-\min(0, \text{old\_g\_vect}[k])$ due to [NZ2012] Proposition 4.2.

EXAMPLES:

```
sage: A = ClusterAlgebra(['F', 4])
sage: A.explore_to_depth(infinity)
sage: B = A.b_matrix()
sage: B.mutate(0)
sage: A1 = ClusterAlgebra(B)
sage: A1.explore_to_depth(infinity)
sage: A2 = A1.mutate_initial(0)
sage: A2._F_poly_dict == A._F_poly_dict
True
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['F', Integer(4)])
>>> A.explore_to_depth(infinity)
```

(continues on next page)

(continued from previous page)

```

>>> B = A.b_matrix()
>>> B.mutate(Integer(0))
>>> A1 = ClusterAlgebra(B)
>>> A1.explore_to_depth(infinity)
>>> A2 = A1.mutate_initial(Integer(0))
>>> A2._F_poly_dict == A._F_poly_dict
True

```

Check that we did not mess up the original algebra because of `UniqueRepresentation`:

```

sage: A = ClusterAlgebra(['A', 2])
sage: A.mutate_initial(0) is A
False

```

```

>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)])
>>> A.mutate_initial(Integer(0)) is A
False

```

A faster example without recomputing F-polynomials:

```

sage: A = ClusterAlgebra(matrix([[0, 5], [-5, 0]]))
sage: A.mutate_initial([0, 1]*10, mutating_F=False)
A Cluster Algebra with cluster variables x20, x21 and no coefficients over_
↪ Integer Ring

```

```

>>> from sage.all import *
>>> A = ClusterAlgebra(matrix([[Integer(0), Integer(5)], [-Integer(5),
↪ Integer(0)]]))
>>> A.mutate_initial([Integer(0), Integer(1)]*Integer(10), mutating_F=False)
A Cluster Algebra with cluster variables x20, x21 and no coefficients over_
↪ Integer Ring

```

Check that [Issue #28176](#) is fixed:

```

sage: A = ClusterAlgebra(matrix(5, [0, 1, -1, 1, -1]), cluster_variable_names=[
↪ 'p13'], coefficient_names=['p12', 'p23', 'p34', 'p41']); A
A Cluster Algebra with cluster variable p13 and coefficients p12, p23, p34,
↪ p41 over Integer Ring
sage: A.mutate_initial(0)
A Cluster Algebra with cluster variable x0 and coefficients p12, p23, p34,
↪ p41 over Integer Ring

sage: A1 = ClusterAlgebra(['A', [2, 1], 1])
sage: A2 = A1.mutate_initial([0, 1, 0])
sage: len(A2.g_vectors_so_far()) == len(A2.F_polynomials_so_far())
True
sage: all(parent(f) == A2._U for f in A2.F_polynomials_so_far())
True
sage: A2.find_g_vector((0, 0, 1)) == []
True

```

```

>>> from sage.all import *
>>> A = ClusterAlgebra(matrix(Integer(5), [Integer(0), Integer(1), -Integer(1),
↳ Integer(1), -Integer(1)]), cluster_variable_names=['p13'], coefficient_
↳ names=['p12', 'p23', 'p34', 'p41']); A
A Cluster Algebra with cluster variable p13 and coefficients p12, p23, p34,
↳ p41 over Integer Ring
>>> A.mutate_initial(Integer(0))
A Cluster Algebra with cluster variable x0 and coefficients p12, p23, p34,
↳ p41 over Integer Ring

>>> A1 = ClusterAlgebra(['A', [Integer(2), Integer(1)], Integer(1)])
>>> A2 = A1.mutate_initial([Integer(0), Integer(1), Integer(0)])
>>> len(A2.g_vectors_so_far()) == len(A2.F_polynomials_so_far())
True
>>> all(parent(f) == A2._U for f in A2.F_polynomials_so_far())
True
>>> A2.find_g_vector((Integer(0), Integer(0), Integer(1))) == []
True

```

rank()

Return the rank of self, i.e. the number of cluster variables in any seed.

EXAMPLES:

```

sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True); A
A Cluster Algebra with cluster variables x0, x1
and coefficients y0, y1 over Integer Ring
sage: A.rank()
2

```

```

>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)], principal_coefficients=True); A
A Cluster Algebra with cluster variables x0, x1
and coefficients y0, y1 over Integer Ring
>>> A.rank()
2

```

reset_current_seed()

Reset the value reported by *current_seed()* to *initial_seed()*.

EXAMPLES:

```

sage: A = ClusterAlgebra(['A', 2])
sage: A.clear_computed_data()
sage: A.current_seed().mutate([1, 0])
sage: A.current_seed() == A.initial_seed()
False
sage: A.reset_current_seed()
sage: A.current_seed() == A.initial_seed()
True

```

```

>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)])

```

(continues on next page)

(continued from previous page)

```

>>> A.clear_computed_data()
>>> A.current_seed().mutate([Integer(1), Integer(0)])
>>> A.current_seed() == A.initial_seed()
False
>>> A.reset_current_seed()
>>> A.current_seed() == A.initial_seed()
True

```

reset_exploring_iterator (*mutating_F=True*)

Reset the iterator used to explore *self*.

INPUT:

- *mutating_F* – boolean (default: *True*); whether to also compute F-polynomials; disable this for speed considerations

EXAMPLES:

```

sage: A = ClusterAlgebra(['A', 4])
sage: A.clear_computed_data()
sage: A.reset_exploring_iterator(mutating_F=False)
sage: A.explore_to_depth(infinity)
sage: len(A.g_vectors_so_far())
14
sage: len(A.F_polynomials_so_far())
4

```

```

>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(4)])
>>> A.clear_computed_data()
>>> A.reset_exploring_iterator(mutating_F=False)
>>> A.explore_to_depth(infinity)
>>> len(A.g_vectors_so_far())
14
>>> len(A.F_polynomials_so_far())
4

```

retract (*x*)

Return *x* as an element of *self*.

EXAMPLES:

```

sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True)
sage: L = A.ambient()
sage: x = L.gen(0)
sage: A.retract(x).parent()
A Cluster Algebra with cluster variables x0, x1 and coefficients y0, y1 over
↳ Integer Ring

```

```

>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)], principal_coefficients=True)
>>> L = A.ambient()
>>> x = L.gen(Integer(0))

```

(continues on next page)

(continued from previous page)

```
>>> A.retract(x).parent()
A Cluster Algebra with cluster variables x0, x1 and coefficients y0, y1 over
↳ Integer Ring
```

scalars()

Return the ring of scalars over which `self` is defined.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2])
sage: A.scalars()
Integer Ring
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)])
>>> A.scalars()
Integer Ring
```

seeds(kwargs)**

Return an iterator running over seeds of `self`.

INPUT:

- `from_current_seed` – boolean (default: `False`); whether to start the iterator from `current_seed()` or `initial_seed()`
- `mutating_F` – boolean (default: `True`); whether to compute F-polynomials also; disable this for speed considerations
- `allowed_directions` – iterable of integers (default: `range(self.rank())`); the directions in which to mutate
- `depth` – positive integer or infinity (default: `infinity`); the maximum depth at which to stop searching
- `catch_KeyboardInterrupt` – boolean (default: `False`); whether to catch `KeyboardInterrupt` and return it rather than raising an exception – this allows the iterator returned by this method to be resumed after being interrupted

ALGORITHM:

This function traverses the exchange graph in a breadth-first search.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 4])
sage: A.clear_computed_data()
sage: seeds = A.seeds(allowed_directions=[3, 0, 1])
sage: _ = list(seeds)
sage: sorted(A.g_vectors_so_far())
[(-1, 0, 0, 0),
 (-1, 1, 0, 0),
 (0, -1, 0, 0),
 (0, 0, 0, -1),
 (0, 0, 0, 1),
 (0, 0, 1, 0),
 (0, 1, 0, 0),
 (1, 0, 0, 0)]
```



```

>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(4)])
>>> A.clear_computed_data()
>>> seeds = A.seeds(allowed_directions=[Integer(3), Integer(0), Integer(1)])
>>> _ = list(seeds)
>>> sorted(A.g_vectors_so_far())
[(-1, 0, 0, 0),
 (-1, 1, 0, 0),
 (0, -1, 0, 0),
 (0, 0, 0, -1),
 (0, 0, 0, 1),
 (0, 0, 1, 0),
 (0, 1, 0, 0),
 (1, 0, 0, 0)]

```

set_current_seed(*seed*)

Set the value reported by `current_seed()` to *seed*, if it makes sense.

INPUT:

- *seed* – a `ClusterAlgebraSeed`

EXAMPLES:

```

sage: A = ClusterAlgebra(['A', 2])
sage: A.clear_computed_data()
sage: S = copy(A.current_seed())
sage: S.mutate([0, 1, 0])
sage: A.current_seed() == S
False
sage: A.set_current_seed(S)
sage: A.current_seed() == S
True
sage: A1 = ClusterAlgebra(['B', 2])
sage: A.set_current_seed(A1.initial_seed())
Traceback (most recent call last):
...
ValueError: this is not a seed in this cluster algebra

```

```

>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)])
>>> A.clear_computed_data()
>>> S = copy(A.current_seed())
>>> S.mutate([Integer(0), Integer(1), Integer(0)])
>>> A.current_seed() == S
False
>>> A.set_current_seed(S)
>>> A.current_seed() == S
True
>>> A1 = ClusterAlgebra(['B', Integer(2)])
>>> A.set_current_seed(A1.initial_seed())
Traceback (most recent call last):
...
ValueError: this is not a seed in this cluster algebra

```

theta_basis_F_polynomial (*g_vector*)

Return the F-polynomial of the element of the theta basis of *self* with g-vector *g_vector*.

INPUT:

- *g_vector* – tuple; the g-vector of the F-polynomial to compute

Warning

Elements of the theta basis do not satisfy a separation of additions formula. See the implementation of `sage.algebras.cluster_algebra.theta_basis_F_polynomial()` for further details.

ALGORITHM:

This method uses the fact that the greedy basis and the theta basis coincide in rank 2 and uses the former defining recursion (Equation (1.5) from [LLZ2014]) to compute.

EXAMPLES:

```
sage: A = ClusterAlgebra(matrix([[0,-3],[2,0]]), principal_coefficients=True)
sage: A.theta_basis_F_polynomial((-1,-1))
u0^4*u1 + 4*u0^3*u1 + 6*u0^2*u1 + 4*u0*u1 + u0 + u1 + 1

sage: A = ClusterAlgebra(['F', 4])
sage: A.theta_basis_F_polynomial((1, 0, 0, 0))
Traceback (most recent call last):
...
NotImplementedError: currently only implemented for cluster algebras of rank 2
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(matrix([[Integer(0),-Integer(3)],[Integer(2),
↳Integer(0)]]), principal_coefficients=True)
>>> A.theta_basis_F_polynomial((-Integer(1),-Integer(1)))
u0^4*u1 + 4*u0^3*u1 + 6*u0^2*u1 + 4*u0*u1 + u0 + u1 + 1

>>> A = ClusterAlgebra(['F', Integer(4)])
>>> A.theta_basis_F_polynomial(Integer(1), Integer(0), Integer(0),
↳Integer(0))
Traceback (most recent call last):
...
NotImplementedError: currently only implemented for cluster algebras of rank 2
```

theta_basis_element (*g_vector*)

Return the element of the theta basis of *self* with g-vector *g_vector*.

INPUT:

- *g_vector* – tuple; the g-vector of the element to compute

EXAMPLES:

```
sage: A = ClusterAlgebra(matrix([[0,-3],[2,0]]), principal_coefficients=True)
sage: A.theta_basis_element((-1,-1))
(x1^8*y0^4*y1 + 4*x1^6*y0^3*y1 + 6*x1^4*y0^2*y1 + x0^3*x1^2*y0 + 4*x1^2*y0*y1
↳+ x0^3 + y1)/(x0^4*x1)
```

(continues on next page)

(continued from previous page)

```
sage: A = ClusterAlgebra(['F', 4])
sage: A.theta_basis_element((1, 0, 0, 0))
Traceback (most recent call last):
...
NotImplementedError: currently only implemented for cluster algebras of rank 2
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(matrix([[Integer(0), -Integer(3)], [Integer(2),
↳ Integer(0)]]), principal_coefficients=True)
>>> A.theta_basis_element((-Integer(1), -Integer(1)))
(x1^8*y0^4*y1 + 4*x1^6*y0^3*y1 + 6*x1^4*y0^2*y1 + x0^3*x1^2*y0 + 4*x1^2*y0*y1
↳ + x0^3 + y1)/(x0^4*x1)

>>> A = ClusterAlgebra(['F', Integer(4)])
>>> A.theta_basis_element((Integer(1), Integer(0), Integer(0), Integer(0)))
Traceback (most recent call last):
...
NotImplementedError: currently only implemented for cluster algebras of rank 2
```

Note

Elements of the theta basis correspond with the associated cluster monomial only for appropriate coefficient choices. For example:

```
sage: A = ClusterAlgebra(matrix([[0, -1], [1, 0], [-1, 0]]))
sage: A.theta_basis_element((-1, 0))
(x1 + y0)/(x0*y0)
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(matrix([[Integer(0), -Integer(1)], [Integer(1),
↳ Integer(0)], [-Integer(1), Integer(0)]]))
>>> A.theta_basis_element((-Integer(1), Integer(0)))
(x1 + y0)/(x0*y0)
```

while:

```
sage: _ = A.find_g_vector((-1, 0));
sage: A.cluster_variable((-1, 0))
(x1 + y0)/x0
```

```
>>> from sage.all import *
>>> _ = A.find_g_vector((-Integer(1), Integer(0)));
>>> A.cluster_variable((-Integer(1), Integer(0)))
(x1 + y0)/x0
```

In particular theta basis elements do not satisfy a separation of additions formula.

Warning

Currently only cluster algebras of rank 2 are supported

See also

```
sage.algebras.cluster_algebra.theta_basis_F_polynomial()
```

upper_bound()

Return the upper bound associated to `self`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['F', 4])
sage: A.upper_bound()
Traceback (most recent call last):
...
NotImplementedError: not implemented yet
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['F', Integer(4)])
>>> A.upper_bound()
Traceback (most recent call last):
...
NotImplementedError: not implemented yet
```

upper_cluster_algebra()

Return the upper cluster algebra associated to `self`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['F', 4])
sage: A.upper_cluster_algebra()
Traceback (most recent call last):
...
NotImplementedError: not implemented yet
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['F', Integer(4)])
>>> A.upper_cluster_algebra()
Traceback (most recent call last):
...
NotImplementedError: not implemented yet
```

class `sage.algebras.cluster_algebra.ClusterAlgebraElement`

Bases: `ElementWrapper`

An element of a cluster algebra.

d_vector()

Return the denominator vector of `self` as a tuple of integers.

EXAMPLES:

```
sage: A = ClusterAlgebra(['F', 4], principal_coefficients=True)
sage: A.current_seed().mutate([0, 2, 1])
sage: x = A.cluster_variable((-1, 2, -2, 2)) * A.cluster_variable((0, 0, 0, 1))
sage: x**2
```

(continues on next page)

(continued from previous page)

```
sage: x.d_vector()
(1, 1, 2, -2)
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['F', Integer(4)], principal_coefficients=True)
>>> A.current_seed().mutate([Integer(0), Integer(2), Integer(1)])
>>> x = A.cluster_variable((-Integer(1), Integer(2), -Integer(2),
↪ Integer(2))) * A.cluster_variable((Integer(0), Integer(0), Integer(0),
↪ Integer(1)))**Integer(2)
>>> x.d_vector()
(1, 1, 2, -2)
```

```
class sage.algebras.cluster_algebra.ClusterAlgebraSeed(B, C, G, parent, **kwargs)
```

Bases: `SageObject`

A seed in a Cluster Algebra.

INPUT:

- `B` – a skew-symmetrizable integer matrix
- `C` – the matrix of `c`-vectors of `self`
- `G` – the matrix of `g`-vectors of `self`
- `parent` – `ClusterAlgebra`; the algebra to which the seed belongs
- `path` – list (default: `[]`); the mutation sequence from the initial seed of `parent` to `self`

Warning

Seeds should **not** be created manually: no test is performed to assert that they are built from consistent data nor that they really are seeds of `parent`. If you create seeds with inconsistent data all sort of things can go wrong, even `__eq__()` is no longer guaranteed to give correct answers. Use at your own risk.

F_polynomial(`j`)

Return the `j`-th F-polynomial of `self`.

INPUT:

- `j` – integer in `range(self.parent().rank())`; the index of the F-polynomial to return

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: S = A.initial_seed()
sage: S.F_polynomial(0)
1
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(3)])
>>> S = A.initial_seed()
>>> S.F_polynomial(Integer(0))
1
```

F_polynomials()

Return all the F-polynomials of `self`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: S = A.initial_seed()
sage: S.F_polynomials()
[1, 1, 1]
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(3)])
>>> S = A.initial_seed()
>>> S.F_polynomials()
[1, 1, 1]
```

b_matrix()

Return the exchange matrix of `self`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: S = A.initial_seed()
sage: S.b_matrix()
[ 0  1  0]
[-1  0 -1]
[ 0  1  0]
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(3)])
>>> S = A.initial_seed()
>>> S.b_matrix()
[ 0  1  0]
[-1  0 -1]
[ 0  1  0]
```

c_matrix()

Return the matrix whose columns are the c-vectors of `self`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: S = A.initial_seed()
sage: S.c_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(3)])
>>> S = A.initial_seed()
>>> S.c_matrix()
[1 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 1 0]
[0 0 1]
```

c_vector(j)

Return the j -th c-vector of `self`.

INPUT:

- j – integer in `range(self.parent().rank())`; the index of the c-vector to return

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: S = A.initial_seed()
sage: S.c_vector(0)
(1, 0, 0)
sage: S.mutate(0)
sage: S.c_vector(0)
(-1, 0, 0)
sage: S.c_vector(1)
(1, 1, 0)
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(3)])
>>> S = A.initial_seed()
>>> S.c_vector(Integer(0))
(1, 0, 0)
>>> S.mutate(Integer(0))
>>> S.c_vector(Integer(0))
(-1, 0, 0)
>>> S.c_vector(Integer(1))
(1, 1, 0)
```

c_vectors()

Return all the c-vectors of `self`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: S = A.initial_seed()
sage: S.c_vectors()
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(3)])
>>> S = A.initial_seed()
>>> S.c_vectors()
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
```

cluster_variable(j)

Return the j -th cluster variable of `self`.

INPUT:

- j – integer in `range(self.parent().rank())`; the index of the cluster variable to return

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: S = A.initial_seed()
sage: S.cluster_variable(0)
x0
sage: S.mutate(0)
sage: S.cluster_variable(0)
(x1 + 1)/x0
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(3)])
>>> S = A.initial_seed()
>>> S.cluster_variable(Integer(0))
x0
>>> S.mutate(Integer(0))
>>> S.cluster_variable(Integer(0))
(x1 + 1)/x0
```

cluster_variables()Return all the cluster variables of `self`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: S = A.initial_seed()
sage: S.cluster_variables()
[x0, x1, x2]
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(3)])
>>> S = A.initial_seed()
>>> S.cluster_variables()
[x0, x1, x2]
```

depth()Return the length of a mutation sequence from the initial seed
of `parent()` to `self`.**Warning**

This is the length of the mutation sequence returned by `path_from_initial_seed()`, which need not be the shortest possible.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2])
sage: S1 = A.initial_seed()
sage: S1.mutate([0, 1, 0, 1])
sage: S1.depth()
4
sage: S2 = A.initial_seed()
```

(continues on next page)

(continued from previous page)

```
sage: S2.mutate(1)
sage: S2.depth()
1
sage: S1 == S2
True
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)])
>>> S1 = A.initial_seed()
>>> S1.mutate([Integer(0), Integer(1), Integer(0), Integer(1)])
>>> S1.depth()
4
>>> S2 = A.initial_seed()
>>> S2.mutate(Integer(1))
>>> S2.depth()
1
>>> S1 == S2
True
```

g_matrix()

Return the matrix whose columns are the g-vectors of `self`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: S = A.initial_seed()
sage: S.g_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(3)])
>>> S = A.initial_seed()
>>> S.g_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
```

g_vector(j)

Return the j -th g-vector of `self`.

INPUT:

- j – integer in `range(self.parent().rank())`; the index of the g-vector to return

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: S = A.initial_seed()
sage: S.g_vector(0)
(1, 0, 0)
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(3)])
>>> S = A.initial_seed()
>>> S.g_vector(Integer(0))
(1, 0, 0)
```

g_vectors()

Return all the g-vectors of self.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: S = A.initial_seed()
sage: S.g_vectors()
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(3)])
>>> S = A.initial_seed()
>>> S.g_vectors()
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
```

mutate(direction, **kwargs)

Mutate self.

INPUT:

- **direction** – in which direction(s) to mutate, it can be:
 - an integer in `range(self.rank())` to mutate in one direction only
 - an iterable of such integers to mutate along a sequence
 - a string “sinks” or “sources” to mutate at all sinks or sources simultaneously
- **inplace** – boolean (default: `True`); whether to mutate in place or to return a new object
- **mutating_F** – boolean (default: `True`); whether to compute F-polynomials while mutating

Note

While knowing F-polynomials is essential to computing cluster variables, the process of mutating them is quite slow. If you care only about combinatorial data like g-vectors and c-vectors, setting `mutating_F=False` yields significant benefits in terms of speed.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2])
sage: S = A.initial_seed()
sage: S.mutate(0); S
The seed of a Cluster Algebra with cluster variables x0, x1
and no coefficients over Integer Ring obtained from the initial
by mutating in direction 0
sage: S.mutate(5)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: cannot mutate in direction 5
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)])
>>> S = A.initial_seed()
>>> S.mutate(Integer(0)); S
The seed of a Cluster Algebra with cluster variables x0, x1
and no coefficients over Integer Ring obtained from the initial
by mutating in direction 0
>>> S.mutate(Integer(5))
Traceback (most recent call last):
...
ValueError: cannot mutate in direction 5
```

parent()

Return the parent of self.

EXAMPLES:

```
sage: A = ClusterAlgebra(['B', 3])
sage: A.current_seed().parent() == A
True
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['B', Integer(3)])
>>> A.current_seed().parent() == A
True
```

path_from_initial_seed()

Return a mutation sequence from the initial seed of *parent()* to self.

Warning

This is the path used to compute self and it does not have to be the shortest possible.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2])
sage: S1 = A.initial_seed()
sage: S1.mutate([0, 1, 0, 1])
sage: S1.path_from_initial_seed()
[0, 1, 0, 1]
sage: S2 = A.initial_seed()
sage: S2.mutate(1)
sage: S2.path_from_initial_seed()
[1]
sage: S1 == S2
True
```

```

>>> from sage.all import *
>>> A = ClusterAlgebra(['A', Integer(2)])
>>> S1 = A.initial_seed()
>>> S1.mutate([Integer(0), Integer(1), Integer(0), Integer(1)])
>>> S1.path_from_initial_seed()
[0, 1, 0, 1]
>>> S2 = A.initial_seed()
>>> S2.mutate(Integer(1))
>>> S2.path_from_initial_seed()
[1]
>>> S1 == S2
True

```

class sage.algebras.cluster_algebra.PrincipalClusterAlgebraElement

Bases: *ClusterAlgebraElement*

An element in a cluster algebra with principle coefficients.

F_polynomial()

Return the F-polynomial of self.

EXAMPLES:

```

sage: A = ClusterAlgebra(['B', 2], principal_coefficients=True)
sage: S = A.initial_seed()
sage: S.mutate([0, 1, 0])
sage: S.cluster_variable(0).F_polynomial() == S.F_polynomial(0)
True
sage: sum(A.initial_cluster_variables()).F_polynomial()
Traceback (most recent call last):
...
ValueError: this element does not have a well defined g-vector

```

```

>>> from sage.all import *
>>> A = ClusterAlgebra(['B', Integer(2)], principal_coefficients=True)
>>> S = A.initial_seed()
>>> S.mutate([Integer(0), Integer(1), Integer(0)])
>>> S.cluster_variable(Integer(0)).F_polynomial() == S.F_
    polynomial(Integer(0))
True
>>> sum(A.initial_cluster_variables()).F_polynomial()
Traceback (most recent call last):
...
ValueError: this element does not have a well defined g-vector

```

g_vector()

Return the g-vector of self.

EXAMPLES:

```

sage: A = ClusterAlgebra(['B', 2], principal_coefficients=True)
sage: A.cluster_variable((1, 0)).g_vector() == (1, 0)
True
sage: sum(A.initial_cluster_variables()).g_vector()

```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: this element does not have a well defined g-vector
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['B', Integer(2)], principal_coefficients=True)
>>> A.cluster_variable((Integer(1), Integer(0))).g_vector() == (Integer(1),
↪ Integer(0))
True
>>> sum(A.initial_cluster_variables()).g_vector()
Traceback (most recent call last):
...
ValueError: this element does not have a well defined g-vector
```

homogeneous_components()

Return a dictionary of the homogeneous components of `self`.

OUTPUT:

A dictionary whose keys are homogeneous degrees and whose values are the summands of `self` of the given degree.

EXAMPLES:

```
sage: A = ClusterAlgebra(['B', 2], principal_coefficients=True)
sage: x = A.cluster_variable((1, 0)) + A.cluster_variable((0, 1))
sage: x.homogeneous_components()
{(0, 1): x1, (1, 0): x0}
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['B', Integer(2)], principal_coefficients=True)
>>> x = A.cluster_variable((Integer(1), Integer(0))) + A.cluster_
↪ variable((Integer(0), Integer(1)))
>>> x.homogeneous_components()
{(0, 1): x1, (1, 0): x0}
```

is_homogeneous()

Return True if `self` is a homogeneous element of `self.parent()`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['B', 2], principal_coefficients=True)
sage: A.cluster_variable((1, 0)).is_homogeneous()
True
sage: x = A.cluster_variable((1, 0)) + A.cluster_variable((0, 1))
sage: x.is_homogeneous()
False
```

```
>>> from sage.all import *
>>> A = ClusterAlgebra(['B', Integer(2)], principal_coefficients=True)
>>> A.cluster_variable((Integer(1), Integer(0))).is_homogeneous()
True
>>> x = A.cluster_variable((Integer(1), Integer(0))) + A.cluster_
```

(continues on next page)

(continued from previous page)

```

↪variable((Integer(0), Integer(1)))
>>> x.is_homogeneous()
False

```

theta_basis_decomposition()

Return the decomposition of `self` in the theta basis.

OUTPUT:

A dictionary whose keys are the g-vectors and whose values are the coefficients in the decomposition of `self` in the theta basis.

EXAMPLES:

```

sage: A = ClusterAlgebra(matrix([[0,-2],[3,0]]), principal_coefficients=True)
sage: f = (A.theta_basis_element((1,0)) + A.theta_basis_element((0,1))**2 +
↪A.coefficient(1) * A.theta_basis_element((1,1))
sage: decomposition = f.theta_basis_decomposition()
sage: sum(decomposition[g] * A.theta_basis_element(g) for g in decomposition)
↪== f
True
sage: f = A.theta_basis_element((4,-4))*A.theta_basis_element((1,-1))
sage: decomposition = f.theta_basis_decomposition()
sage: sum(decomposition[g] * A.theta_basis_element(g) for g in decomposition)
↪== f
True

```

```

>>> from sage.all import *
>>> A = ClusterAlgebra(matrix([[Integer(0),-Integer(2)],[Integer(3),
↪Integer(0)]]), principal_coefficients=True)
>>> f = (A.theta_basis_element((Integer(1),Integer(0))) + A.theta_basis_
↪element((Integer(0),Integer(1))))**Integer(2) + A.coefficient(Integer(1))*
↪A.theta_basis_element((Integer(1),Integer(1)))
>>> decomposition = f.theta_basis_decomposition()
>>> sum(decomposition[g] * A.theta_basis_element(g) for g in decomposition)
↪== f
True
>>> f = A.theta_basis_element((Integer(4),-Integer(4)))*A.theta_basis_
↪element((Integer(1),-Integer(1)))
>>> decomposition = f.theta_basis_decomposition()
>>> sum(decomposition[g] * A.theta_basis_element(g) for g in decomposition)
↪== f
True

```

5.6 Descent Algebras

AUTHORS:

- Travis Scrimshaw (2013-07-28): Initial version

class sage.combinat.descent_algebra.**DescentAlgebra**(R, n)

Bases: UniqueRepresentation, Parent

Solomon's descent algebra.

The descent algebra Σ_n over a ring R is a subalgebra of the symmetric group algebra RS_n . (The product in the latter algebra is defined by $(pq)(i) = q(p(i))$ for any two permutations p and q in S_n and every $i \in \{1, 2, \dots, n\}$. The algebra Σ_n inherits this product.)

There are three bases currently implemented for Σ_n :

- the standard basis D_S of (sums of) descent classes, indexed by subsets S of $\{1, 2, \dots, n-1\}$,
- the subset basis B_p , indexed by compositions p of n ,
- the idempotent basis I_p , indexed by compositions p of n , which is used to construct the mutually orthogonal idempotents of the symmetric group algebra.

The idempotent basis is only defined when R is a $\mathbf{Q}$ -algebra.

We follow the notations and conventions in [GR1989], apart from the order of multiplication being different from the one used in that article. Schocker's exposition [Sch2004], in turn, uses the same order of multiplication as we are, but has different notations for the bases.

INPUT:

- R – the base ring
- n – nonnegative integer

REFERENCES:

- [GR1989]
- [At1992]
- [MR1995]
- [Sch2004]

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: D = DA.D(); D
Descent algebra of 4 over Rational Field in the standard basis
sage: B = DA.B(); B
Descent algebra of 4 over Rational Field in the subset basis
sage: I = DA.I(); I
Descent algebra of 4 over Rational Field in the idempotent basis
sage: basis_B = B.basis()
sage: elt = basis_B[Composition([1,2,1])] + 4*basis_B[Composition([1,3])]; elt
B[1, 2, 1] + 4*B[1, 3]
sage: D(elt)
5*D{} + 5*D{1} + D{1, 3} + D{3}
sage: I(elt)
7/6*I[1, 1, 1, 1] + 2*I[1, 1, 2] + 3*I[1, 2, 1] + 4*I[1, 3]
```

```
>>> from sage.all import *
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> D = DA.D(); D
Descent algebra of 4 over Rational Field in the standard basis
>>> B = DA.B(); B
Descent algebra of 4 over Rational Field in the subset basis
>>> I = DA.I(); I
Descent algebra of 4 over Rational Field in the idempotent basis
>>> basis_B = B.basis()
```

(continues on next page)

(continued from previous page)

```

>>> elt = basis_B[Composition([Integer(1), Integer(2), Integer(1)])] +
↳ Integer(4)*basis_B[Composition([Integer(1), Integer(3)])]; elt
B[1, 2, 1] + 4*B[1, 3]
>>> D(elt)
5*D{} + 5*D{1} + D{1, 3} + D{3}
>>> I(elt)
7/6*I[1, 1, 1, 1] + 2*I[1, 1, 2] + 3*I[1, 2, 1] + 4*I[1, 3]

```

As syntactic sugar, one can use the notation $D[i, \dots, 1]$ to construct elements of the basis; note that for the empty set one must use $D[[]]$ due to Python's syntax:

```

sage: D[[]] + D[2] + 2*D[1, 2]
D{} + 2*D{1, 2} + D{2}

```

```

>>> from sage.all import *
>>> D[[]] + D[Integer(2)] + Integer(2)*D[Integer(1), Integer(2)]
D{} + 2*D{1, 2} + D{2}

```

The same syntax works for the other bases:

```

sage: I[1, 2, 1] + 3*I[4] + 2*I[3, 1]
I[1, 2, 1] + 2*I[3, 1] + 3*I[4]

```

```

>>> from sage.all import *
>>> I[Integer(1), Integer(2), Integer(1)] + Integer(3)*I[Integer(4)] +
↳ Integer(2)*I[Integer(3), Integer(1)]
I[1, 2, 1] + 2*I[3, 1] + 3*I[4]

```

```

class B(alg, prefix='B')

```

Bases: `CombinatorialFreeModule`, `BindableClass`

The subset basis of a descent algebra (indexed by compositions).

The subset basis $(B_S)_{S \subseteq \{1, 2, \dots, n-1\}}$ of Σ_n is formed by

$$B_S = \sum_{T \subseteq S} D_T,$$

where $(D_S)_{S \subseteq \{1, 2, \dots, n-1\}}$ is the *standard basis*. However it is more natural to index the subset basis by compositions of n under the bijection $\{i_1, i_2, \dots, i_k\} \mapsto (i_1, i_2 - i_1, i_3 - i_2, \dots, i_k - i_{k-1}, n - i_k)$ (where $i_1 < i_2 < \dots < i_k$), which is what Sage uses to index the basis.

The basis element B_p is denoted Ξ^p in [Sch2004].

By using compositions of n , the product $B_p B_q$ becomes a sum over the nonnegative-integer matrices M with row sum p and column sum q . The summand corresponding to M is B_c , where c is the composition obtained by reading M row-by-row from left-to-right and top-to-bottom and removing all zeroes. This multiplication rule is commonly called “Solomon’s Mackey formula”.

EXAMPLES:

```

sage: DA = DescentAlgebra(QQ, 4)
sage: B = DA.B()
sage: list(B.basis())
[B[1, 1, 1, 1], B[1, 1, 2], B[1, 2, 1], B[1, 3],
 B[2, 1, 1], B[2, 2], B[3, 1], B[4]]

```



```
>>> from sage.all import *
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> B = DA.B()
>>> list(B.basis())
[B[1, 1, 1, 1], B[1, 1, 2], B[1, 2, 1], B[1, 3],
 B[2, 1, 1], B[2, 2], B[3, 1], B[4]]
```

one_basis()

Return the identity element which is the composition $[n]$, as per `AlgebrasWithBasis.ParentMethods.one_basis`.

EXAMPLES:

```
sage: DescentAlgebra(QQ, 4).B().one_basis()
[4]
sage: DescentAlgebra(QQ, 0).B().one_basis()
[]

sage: all( U * DescentAlgebra(QQ, 3).B().one() == U
....:      for U in DescentAlgebra(QQ, 3).B().basis() )
True
```

```
>>> from sage.all import *
>>> DescentAlgebra(QQ, Integer(4)).B().one_basis()
[4]
>>> DescentAlgebra(QQ, Integer(0)).B().one_basis()
[]

>>> all( U * DescentAlgebra(QQ, Integer(3)).B().one() == U
...      for U in DescentAlgebra(QQ, Integer(3)).B().basis() )
True
```

product_on_basis(p, q)

Return $B_p B_q$, where p and q are compositions of n .

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: B = DA.B()
sage: p = Composition([1,2,1])
sage: q = Composition([3,1])
sage: B.product_on_basis(p, q)
B[1, 1, 1, 1] + 2*B[1, 2, 1]
```

```
>>> from sage.all import *
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> B = DA.B()
>>> p = Composition([Integer(1), Integer(2), Integer(1)])
>>> q = Composition([Integer(3), Integer(1)])
>>> B.product_on_basis(p, q)
B[1, 1, 1, 1] + 2*B[1, 2, 1]
```

to_D_basis(p)

Return B_p as a linear combination of D -basis elements.

EXAMPLES:

```

sage: DA = DescentAlgebra(QQ, 4)
sage: B = DA.B()
sage: D = DA.D()
sage: list(map(D, B.basis())) # indirect doctest
[D{} + D{1} + D{1, 2} + D{1, 2, 3}
 + D{1, 3} + D{2} + D{2, 3} + D{3},
 D{} + D{1} + D{1, 2} + D{2},
 D{} + D{1} + D{1, 3} + D{3},
 D{} + D{1},
 D{} + D{2} + D{2, 3} + D{3},
 D{} + D{2},
 D{} + D{3},
 D{}]

```

```

>>> from sage.all import *
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> B = DA.B()
>>> D = DA.D()
>>> list(map(D, B.basis())) # indirect doctest
[D{} + D{1} + D{1, 2} + D{1, 2, 3}
 + D{1, 3} + D{2} + D{2, 3} + D{3},
 D{} + D{1} + D{1, 2} + D{2},
 D{} + D{1} + D{1, 3} + D{3},
 D{} + D{1},
 D{} + D{2} + D{2, 3} + D{3},
 D{} + D{2},
 D{} + D{3},
 D{}]

```

to_I_basis(p)Return B_p as a linear combination of I -basis elements.

This is done using the formula

$$B_p = \sum_{q \leq p} \frac{1}{\mathbf{k}!(q, p)} I_q,$$

where $\leq$ is the refinement order and $\mathbf{k}!(q, p)$ is defined as follows: When $q \leq p$, we can write q as a concatenation $q_{(1)}q_{(2)} \cdots q_{(k)}$ with each $q_{(i)}$ being a composition of the i -th entry of p , and then we set $\mathbf{k}!(q, p)$ to be $l(q_{(1)})!l(q_{(2)})! \cdots l(q_{(k)})!$, where $l(r)$ denotes the number of parts of any composition r .

EXAMPLES:

```

sage: DA = DescentAlgebra(QQ, 4)
sage: B = DA.B()
sage: I = DA.I()
sage: list(map(I, B.basis())) # indirect doctest
[I[1, 1, 1, 1],
 1/2*I[1, 1, 1, 1] + I[1, 1, 2],
 1/2*I[1, 1, 1, 1] + I[1, 2, 1],
 1/6*I[1, 1, 1, 1] + 1/2*I[1, 1, 2] + 1/2*I[1, 2, 1] + I[1, 3],
 1/2*I[1, 1, 1, 1] + I[2, 1, 1],

```

(continues on next page)

(continued from previous page)

```

1/4*I[1, 1, 1, 1] + 1/2*I[1, 1, 2] + 1/2*I[2, 1, 1] + I[2, 2],
1/6*I[1, 1, 1, 1] + 1/2*I[1, 2, 1] + 1/2*I[2, 1, 1] + I[3, 1],
1/24*I[1, 1, 1, 1] + 1/6*I[1, 1, 2] + 1/6*I[1, 2, 1]
+ 1/2*I[1, 3] + 1/6*I[2, 1, 1] + 1/2*I[2, 2] + 1/2*I[3, 1] + I[4]]

```

```

>>> from sage.all import *
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> B = DA.B()
>>> I = DA.I()
>>> list(map(I, B.basis())) # indirect doctest
[I[1, 1, 1, 1],
 1/2*I[1, 1, 1, 1] + I[1, 1, 2],
 1/2*I[1, 1, 1, 1] + I[1, 2, 1],
 1/6*I[1, 1, 1, 1] + 1/2*I[1, 1, 2] + 1/2*I[1, 2, 1] + I[1, 3],
 1/2*I[1, 1, 1, 1] + I[2, 1, 1],
 1/4*I[1, 1, 1, 1] + 1/2*I[1, 1, 2] + 1/2*I[2, 1, 1] + I[2, 2],
 1/6*I[1, 1, 1, 1] + 1/2*I[1, 2, 1] + 1/2*I[2, 1, 1] + I[3, 1],
 1/24*I[1, 1, 1, 1] + 1/6*I[1, 1, 2] + 1/6*I[1, 2, 1]
+ 1/2*I[1, 3] + 1/6*I[2, 1, 1] + 1/2*I[2, 2] + 1/2*I[3, 1] + I[4]]

```

to_nsym(p)

Return B_p as an element in $NSym$, the non-commutative symmetric functions.

This maps B_p to S_p where S denotes the Complete basis of $NSym$.

EXAMPLES:

```

sage: B = DescentAlgebra(QQ, 4).B()
sage: S = NonCommutativeSymmetricFunctions(QQ).Complete()
sage: list(map(S, B.basis())) # indirect doctest
[S[1, 1, 1, 1],
 S[1, 1, 2],
 S[1, 2, 1],
 S[1, 3],
 S[2, 1, 1],
 S[2, 2],
 S[3, 1],
 S[4]]

```

```

>>> from sage.all import *
>>> B = DescentAlgebra(QQ, Integer(4)).B()
>>> S = NonCommutativeSymmetricFunctions(QQ).Complete()
>>> list(map(S, B.basis())) # indirect doctest
[S[1, 1, 1, 1],
 S[1, 1, 2],
 S[1, 2, 1],
 S[1, 3],
 S[2, 1, 1],
 S[2, 2],
 S[3, 1],
 S[4]]

```

```
class D (alg, prefix='D')
```

Bases: `CombinatorialFreeModule`, `BindableClass`

The standard basis of a descent algebra.

This basis is indexed by $S \subseteq \{1, 2, \dots, n-1\}$, and the basis vector indexed by S is the sum of all permutations, taken in the symmetric group algebra RS_n , whose descent set is S . We denote this basis vector by D_S .

Occasionally this basis appears in literature but indexed by compositions of n rather than subsets of $\{1, 2, \dots, n-1\}$. The equivalence between these two indexings is owed to the bijection from the power set of $\{1, 2, \dots, n-1\}$ to the set of all compositions of n which sends every subset $\{i_1, i_2, \dots, i_k\}$ of $\{1, 2, \dots, n-1\}$ (with $i_1 < i_2 < \dots < i_k$) to the composition $(i_1, i_2 - i_1, \dots, i_k - i_{k-1}, n - i_k)$.

The basis element corresponding to a composition p (or to the subset of $\{1, 2, \dots, n-1\}$) is denoted Δ^p in [Sch2004].

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: D = DA.D()
sage: list(D.basis())
[D{}, D{1}, D{2}, D{3}, D{1, 2}, D{1, 3}, D{2, 3}, D{1, 2, 3}]

sage: DA = DescentAlgebra(QQ, 0)
sage: D = DA.D()
sage: list(D.basis())
[D{}]
```

```
>>> from sage.all import *
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> D = DA.D()
>>> list(D.basis())
[D{}, D{1}, D{2}, D{3}, D{1, 2}, D{1, 3}, D{2, 3}, D{1, 2, 3}]

>>> DA = DescentAlgebra(QQ, Integer(0))
>>> D = DA.D()
>>> list(D.basis())
[D{}]
```

one_basis()

Return the identity element, as per `AlgebrasWithBasis.ParentMethods.one_basis`.

EXAMPLES:

```
sage: DescentAlgebra(QQ, 4).D().one_basis()
()
sage: DescentAlgebra(QQ, 0).D().one_basis()
()

sage: all( U * DescentAlgebra(QQ, 3).D().one() == U
....:      for U in DescentAlgebra(QQ, 3).D().basis() )
True
```

```
>>> from sage.all import *
>>> DescentAlgebra(QQ, Integer(4)).D().one_basis()
```

(continues on next page)

(continued from previous page)

```
(
)>>> DescentAlgebra(QQ, Integer(0)).D().one_basis()
(
)>>> all( U * DescentAlgebra(QQ, Integer(3)).D().one() == U
...      for U in DescentAlgebra(QQ, Integer(3)).D().basis() )
True
```

product_on_basis(S, T)Return $D_S D_T$, where S and T are subsets of $[n - 1]$.

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: D = DA.D()
sage: D.product_on_basis((1, 3), (2,))
D{} + D{1} + D{1, 2} + 2*D{1, 2, 3} + D{1, 3} + D{2} + D{2, 3} + D{3}
```

```
>>> from sage.all import *
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> D = DA.D()
>>> D.product_on_basis((Integer(1), Integer(3)), (Integer(2),))
D{} + D{1} + D{1, 2} + 2*D{1, 2, 3} + D{1, 3} + D{2} + D{2, 3} + D{3}
```

to_B_basis(S)Return D_S as a linear combination of B_p -basis elements.

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: D = DA.D()
sage: B = DA.B()
sage: list(map(B, D.basis())) # indirect doctest
[B[4],
 B[1, 3] - B[4],
 B[2, 2] - B[4],
 B[3, 1] - B[4],
 B[1, 1, 2] - B[1, 3] - B[2, 2] + B[4],
 B[1, 2, 1] - B[1, 3] - B[3, 1] + B[4],
 B[2, 1, 1] - B[2, 2] - B[3, 1] + B[4],
 B[1, 1, 1, 1] - B[1, 1, 2] - B[1, 2, 1] + B[1, 3]
 - B[2, 1, 1] + B[2, 2] + B[3, 1] - B[4]]
```

```
>>> from sage.all import *
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> D = DA.D()
>>> B = DA.B()
>>> list(map(B, D.basis())) # indirect doctest
[B[4],
 B[1, 3] - B[4],
 B[2, 2] - B[4],
 B[3, 1] - B[4],
```

(continues on next page)

(continued from previous page)

```

B[1, 1, 2] - B[1, 3] - B[2, 2] + B[4],
B[1, 2, 1] - B[1, 3] - B[3, 1] + B[4],
B[2, 1, 1] - B[2, 2] - B[3, 1] + B[4],
B[1, 1, 1, 1] - B[1, 1, 2] - B[1, 2, 1] + B[1, 3]
- B[2, 1, 1] + B[2, 2] + B[3, 1] - B[4]]

```

to_symmetric_group_algebra_on_basis(*S*)

Return D_S as a linear combination of basis elements in the symmetric group algebra.

EXAMPLES:

```

sage: D = DescentAlgebra(QQ, 4).D()
sage: [D.to_symmetric_group_algebra_on_basis(tuple(b))
....:  for b in Subsets(3)]
[[1, 2, 3, 4],
 [2, 1, 3, 4] + [3, 1, 2, 4] + [4, 1, 2, 3],
 [1, 3, 2, 4] + [1, 4, 2, 3] + [2, 3, 1, 4]
 + [2, 4, 1, 3] + [3, 4, 1, 2],
 [1, 2, 4, 3] + [1, 3, 4, 2] + [2, 3, 4, 1],
 [3, 2, 1, 4] + [4, 2, 1, 3] + [4, 3, 1, 2],
 [2, 1, 4, 3] + [3, 1, 4, 2] + [3, 2, 4, 1]
 + [4, 1, 3, 2] + [4, 2, 3, 1],
 [1, 4, 3, 2] + [2, 4, 3, 1] + [3, 4, 2, 1],
 [4, 3, 2, 1]]

```

```

>>> from sage.all import *
>>> D = DescentAlgebra(QQ, Integer(4)).D()
>>> [D.to_symmetric_group_algebra_on_basis(tuple(b))
...  for b in Subsets(Integer(3))]
[[1, 2, 3, 4],
 [2, 1, 3, 4] + [3, 1, 2, 4] + [4, 1, 2, 3],
 [1, 3, 2, 4] + [1, 4, 2, 3] + [2, 3, 1, 4]
 + [2, 4, 1, 3] + [3, 4, 1, 2],
 [1, 2, 4, 3] + [1, 3, 4, 2] + [2, 3, 4, 1],
 [3, 2, 1, 4] + [4, 2, 1, 3] + [4, 3, 1, 2],
 [2, 1, 4, 3] + [3, 1, 4, 2] + [3, 2, 4, 1]
 + [4, 1, 3, 2] + [4, 2, 3, 1],
 [1, 4, 3, 2] + [2, 4, 3, 1] + [3, 4, 2, 1],
 [4, 3, 2, 1]]

```

class I(*alg*, *prefix*='I')

Bases: `CombinatorialFreeModule`, `BindableClass`

The idempotent basis of a descent algebra.

The idempotent basis $(I_p)_{p \models n}$ is a basis for Σ_n whenever the ground ring is a $\mathbf{Q}$ -algebra. One way to compute it is using the formula (Theorem 3.3 in [GR1989])

$$I_p = \sum_{q \leq p} \frac{(-1)^{l(q)-l(p)}}{\mathbf{k}(q,p)} B_q,$$

where $\leq$ is the refinement order and $l(r)$ denotes the number of parts of any composition r , and where $\mathbf{k}(q,p)$ is defined as follows: When $q \leq p$, we can write q as a concatenation $q_{(1)}q_{(2)} \cdots q_{(k)}$ with each $q_{(i)}$ being a composition of the i -th entry of p , and then we set $\mathbf{k}(q,p)$ to be the product $l(q_{(1)})l(q_{(2)}) \cdots l(q_{(k)})$.

Let $\lambda(p)$ denote the partition obtained from a composition p by sorting. This basis is called the idempotent basis since for any q such that $\lambda(p) = \lambda(q)$, we have:

$$I_p I_q = s(\lambda) I_p$$

where λ denotes $\lambda(p) = \lambda(q)$, and where $s(\lambda)$ is the stabilizer of λ in S_n . (This is part of Theorem 4.2 in [GR1989].)

It is also straightforward to compute the idempotents E_λ for the symmetric group algebra by the formula (Theorem 3.2 in [GR1989]):

$$E_\lambda = \frac{1}{k!} \sum_{\lambda(p)=\lambda} I_p.$$

Note

The basis elements are not orthogonal idempotents.

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: I = DA.I()
sage: list(I.basis())
[I[1, 1, 1, 1], I[1, 1, 2], I[1, 2, 1], I[1, 3], I[2, 1, 1], I[2, 2], I[3, 1],
↪ I[4]]
```

```
>>> from sage.all import *
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> I = DA.I()
>>> list(I.basis())
[I[1, 1, 1, 1], I[1, 1, 2], I[1, 2, 1], I[1, 3], I[2, 1, 1], I[2, 2], I[3, 1],
↪ I[4]]
```

`idempotent(la)`

Return the idempotent corresponding to the partition `la` of n .

EXAMPLES:

```
sage: I = DescentAlgebra(QQ, 4).I()
sage: E = I.idempotent([3,1]); E
1/2*I[1, 3] + 1/2*I[3, 1]
sage: E*E == E
True
sage: E2 = I.idempotent([2,1,1]); E2
1/6*I[1, 1, 2] + 1/6*I[1, 2, 1] + 1/6*I[2, 1, 1]
sage: E2*E2 == E2
True
sage: E*E2 == I.zero()
True
```

```
>>> from sage.all import *
>>> I = DescentAlgebra(QQ, Integer(4)).I()
```

(continues on next page)

(continued from previous page)

```

>>> E = I.idempotent([Integer(3), Integer(1)]); E
1/2*I[1, 3] + 1/2*I[3, 1]
>>> E*E == E
True
>>> E2 = I.idempotent([Integer(2), Integer(1), Integer(1)]); E2
1/6*I[1, 1, 2] + 1/6*I[1, 2, 1] + 1/6*I[2, 1, 1]
>>> E2*E2 == E2
True
>>> E*E2 == I.zero()
True

```

one()

Return the identity element, which is $B_{[n]}$, in the I basis.

EXAMPLES:

```

sage: DescentAlgebra(QQ, 4).I().one()
1/24*I[1, 1, 1, 1] + 1/6*I[1, 1, 2] + 1/6*I[1, 2, 1]
+ 1/2*I[1, 3] + 1/6*I[2, 1, 1] + 1/2*I[2, 2]
+ 1/2*I[3, 1] + I[4]
sage: DescentAlgebra(QQ, 0).I().one()
I[]

```

```

>>> from sage.all import *
>>> DescentAlgebra(QQ, Integer(4)).I().one()
1/24*I[1, 1, 1, 1] + 1/6*I[1, 1, 2] + 1/6*I[1, 2, 1]
+ 1/2*I[1, 3] + 1/6*I[2, 1, 1] + 1/2*I[2, 2]
+ 1/2*I[3, 1] + I[4]
>>> DescentAlgebra(QQ, Integer(0)).I().one()
I[]

```

one_basis()

The element 1 is not (generally) a basis vector in the I basis, thus this raises a `TypeError`.

EXAMPLES:

```

sage: DescentAlgebra(QQ, 4).I().one_basis()
Traceback (most recent call last):
...
TypeError: 1 is not a basis element in the I basis

```

```

>>> from sage.all import *
>>> DescentAlgebra(QQ, Integer(4)).I().one_basis()
Traceback (most recent call last):
...
TypeError: 1 is not a basis element in the I basis

```

product_on_basis(p, q)

Return $I_p I_q$, where p and q are compositions of n .

EXAMPLES:


```

sage: DA = DescentAlgebra(QQ, 4)
sage: I = DA.I()
sage: p = Composition([1,2,1])
sage: q = Composition([3,1])
sage: I.product_on_basis(p, q)
0
sage: I.product_on_basis(p, p)
2*I[1, 2, 1]

```

```

>>> from sage.all import *
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> I = DA.I()
>>> p = Composition([Integer(1), Integer(2), Integer(1)])
>>> q = Composition([Integer(3), Integer(1)])
>>> I.product_on_basis(p, q)
0
>>> I.product_on_basis(p, p)
2*I[1, 2, 1]

```

`to_B_basis(p)`

Return I_p as a linear combination of B -basis elements.

This is computed using the formula (Theorem 3.3 in [GR1989])

$$I_p = \sum_{q \leq p} \frac{(-1)^{l(q)-l(p)}}{\mathbf{k}(q, p)} B_q,$$

where $\leq$ is the refinement order and $l(r)$ denotes the number of parts of any composition r , and where $\mathbf{k}(q, p)$ is defined as follows: When $q \leq p$, we can write q as a concatenation $q_{(1)}q_{(2)} \cdots q_{(k)}$ with each $q_{(i)}$ being a composition of the i -th entry of p , and then we set $\mathbf{k}(q, p)$ to be $l(q_{(1)})l(q_{(2)}) \cdots l(q_{(k)})$.

EXAMPLES:

```

sage: DA = DescentAlgebra(QQ, 4)
sage: B = DA.B()
sage: I = DA.I()
sage: list(map(B, I.basis())) # indirect doctest
[B[1, 1, 1, 1],
 -1/2*B[1, 1, 1, 1] + B[1, 1, 2],
 -1/2*B[1, 1, 1, 1] + B[1, 2, 1],
 1/3*B[1, 1, 1, 1] - 1/2*B[1, 1, 2] - 1/2*B[1, 2, 1] + B[1, 3],
 -1/2*B[1, 1, 1, 1] + B[2, 1, 1],
 1/4*B[1, 1, 1, 1] - 1/2*B[1, 1, 2] - 1/2*B[2, 1, 1] + B[2, 2],
 1/3*B[1, 1, 1, 1] - 1/2*B[1, 2, 1] - 1/2*B[2, 1, 1] + B[3, 1],
 -1/4*B[1, 1, 1, 1] + 1/3*B[1, 1, 2] + 1/3*B[1, 2, 1]
 - 1/2*B[1, 3] + 1/3*B[2, 1, 1] - 1/2*B[2, 2]
 - 1/2*B[3, 1] + B[4]]

```

```

>>> from sage.all import *
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> B = DA.B()
>>> I = DA.I()
>>> list(map(B, I.basis())) # indirect doctest

```

(continues on next page)

(continued from previous page)

```
[B[1, 1, 1, 1],
 -1/2*B[1, 1, 1, 1] + B[1, 1, 2],
 -1/2*B[1, 1, 1, 1] + B[1, 2, 1],
 1/3*B[1, 1, 1, 1] - 1/2*B[1, 1, 2] - 1/2*B[1, 2, 1] + B[1, 3],
 -1/2*B[1, 1, 1, 1] + B[2, 1, 1],
 1/4*B[1, 1, 1, 1] - 1/2*B[1, 1, 2] - 1/2*B[2, 1, 1] + B[2, 2],
 1/3*B[1, 1, 1, 1] - 1/2*B[1, 2, 1] - 1/2*B[2, 1, 1] + B[3, 1],
 -1/4*B[1, 1, 1, 1] + 1/3*B[1, 1, 2] + 1/3*B[1, 2, 1]
 - 1/2*B[1, 3] + 1/3*B[2, 1, 1] - 1/2*B[2, 2]
 - 1/2*B[3, 1] + B[4]]
```

a_realization()Return a particular realization of *self* (the *B*-basis).

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: DA.a_realization()
Descent algebra of 4 over Rational Field in the subset basis
```

```
>>> from sage.all import *
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> DA.a_realization()
Descent algebra of 4 over Rational Field in the subset basis
```

idempotentalias of *I***standard**alias of *D***subset**alias of *B***class** sage.combinat.descent_algebra.DescentAlgebraBases (*base*)Bases: *Category_realization_of_parent*

The category of bases of a descent algebra.

class ElementMethods

Bases: object

to_symmetric_group_algebra()Return *self* in the symmetric group algebra.

EXAMPLES:

```
sage: B = DescentAlgebra(QQ, 4).B()
sage: B[1,3].to_symmetric_group_algebra()
[1, 2, 3, 4] + [2, 1, 3, 4] + [3, 1, 2, 4] + [4, 1, 2, 3]
sage: I = DescentAlgebra(QQ, 4).I()
sage: elt = I(B[1,3])
sage: elt.to_symmetric_group_algebra()
[1, 2, 3, 4] + [2, 1, 3, 4] + [3, 1, 2, 4] + [4, 1, 2, 3]
```

```

>>> from sage.all import *
>>> B = DescentAlgebra(QQ, Integer(4)).B()
>>> B[Integer(1), Integer(3)].to_symmetric_group_algebra()
[1, 2, 3, 4] + [2, 1, 3, 4] + [3, 1, 2, 4] + [4, 1, 2, 3]
>>> I = DescentAlgebra(QQ, Integer(4)).I()
>>> elt = I(B[Integer(1), Integer(3)])
>>> elt.to_symmetric_group_algebra()
[1, 2, 3, 4] + [2, 1, 3, 4] + [3, 1, 2, 4] + [4, 1, 2, 3]

```

class ParentMethods

Bases: object

to_symmetric_group_algebra()

Morphism from `self` to the symmetric group algebra.

EXAMPLES:

```

sage: D = DescentAlgebra(QQ, 4).D()
sage: D.to_symmetric_group_algebra(D[1,3])
[2, 1, 4, 3] + [3, 1, 4, 2] + [3, 2, 4, 1] + [4, 1, 3, 2] + [4, 2, 3, 1]
sage: B = DescentAlgebra(QQ, 4).B()
sage: B.to_symmetric_group_algebra(B[1,2,1])
[1, 2, 3, 4] + [1, 2, 4, 3] + [1, 3, 4, 2] + [2, 1, 3, 4]
+ [2, 1, 4, 3] + [2, 3, 4, 1] + [3, 1, 2, 4] + [3, 1, 4, 2]
+ [3, 2, 4, 1] + [4, 1, 2, 3] + [4, 1, 3, 2] + [4, 2, 3, 1]

```

```

>>> from sage.all import *
>>> D = DescentAlgebra(QQ, Integer(4)).D()
>>> D.to_symmetric_group_algebra(D[Integer(1), Integer(3)])
[2, 1, 4, 3] + [3, 1, 4, 2] + [3, 2, 4, 1] + [4, 1, 3, 2] + [4, 2, 3, 1]
>>> B = DescentAlgebra(QQ, Integer(4)).B()
>>> B.to_symmetric_group_algebra(B[Integer(1), Integer(2), Integer(1)])
[1, 2, 3, 4] + [1, 2, 4, 3] + [1, 3, 4, 2] + [2, 1, 3, 4]
+ [2, 1, 4, 3] + [2, 3, 4, 1] + [3, 1, 2, 4] + [3, 1, 4, 2]
+ [3, 2, 4, 1] + [4, 1, 2, 3] + [4, 1, 3, 2] + [4, 2, 3, 1]

```

to_symmetric_group_algebra_on_basis(S)

Return the basis element index by `s` as a linear combination of basis elements in the symmetric group algebra.

EXAMPLES:

```

sage: B = DescentAlgebra(QQ, 3).B()
sage: [B.to_symmetric_group_algebra_on_basis(c)
....: for c in Compositions(3)]
[[1, 2, 3] + [1, 3, 2] + [2, 1, 3]
+ [2, 3, 1] + [3, 1, 2] + [3, 2, 1],
[1, 2, 3] + [2, 1, 3] + [3, 1, 2],
[1, 2, 3] + [1, 3, 2] + [2, 3, 1],
[1, 2, 3]]
sage: I = DescentAlgebra(QQ, 3).I()
sage: [I.to_symmetric_group_algebra_on_basis(c)
....: for c in Compositions(3)]
[[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1]

```

(continues on next page)

(continued from previous page)

```

+ [3, 1, 2] + [3, 2, 1],
1/2*[1, 2, 3] - 1/2*[1, 3, 2] + 1/2*[2, 1, 3]
- 1/2*[2, 3, 1] + 1/2*[3, 1, 2] - 1/2*[3, 2, 1],
1/2*[1, 2, 3] + 1/2*[1, 3, 2] - 1/2*[2, 1, 3]
+ 1/2*[2, 3, 1] - 1/2*[3, 1, 2] - 1/2*[3, 2, 1],
1/3*[1, 2, 3] - 1/6*[1, 3, 2] - 1/6*[2, 1, 3]
- 1/6*[2, 3, 1] - 1/6*[3, 1, 2] + 1/3*[3, 2, 1]]

```

```

>>> from sage.all import *
>>> B = DescentAlgebra(QQ, Integer(3)).B()
>>> [B.to_symmetric_group_algebra_on_basis(c)
...  for c in Compositions(Integer(3))]
[[1, 2, 3] + [1, 3, 2] + [2, 1, 3]
+ [2, 3, 1] + [3, 1, 2] + [3, 2, 1],
[1, 2, 3] + [2, 1, 3] + [3, 1, 2],
[1, 2, 3] + [1, 3, 2] + [2, 3, 1],
[1, 2, 3]]
>>> I = DescentAlgebra(QQ, Integer(3)).I()
>>> [I.to_symmetric_group_algebra_on_basis(c)
...  for c in Compositions(Integer(3))]
[[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1]
+ [3, 1, 2] + [3, 2, 1],
1/2*[1, 2, 3] - 1/2*[1, 3, 2] + 1/2*[2, 1, 3]
- 1/2*[2, 3, 1] + 1/2*[3, 1, 2] - 1/2*[3, 2, 1],
1/2*[1, 2, 3] + 1/2*[1, 3, 2] - 1/2*[2, 1, 3]
+ 1/2*[2, 3, 1] - 1/2*[3, 1, 2] - 1/2*[3, 2, 1],
1/3*[1, 2, 3] - 1/6*[1, 3, 2] - 1/6*[2, 1, 3]
- 1/6*[2, 3, 1] - 1/6*[3, 1, 2] + 1/3*[3, 2, 1]]

```

super_categories()

The super categories of self.

EXAMPLES:

```

sage: from sage.combinat.descent_algebra import DescentAlgebraBases
sage: DA = DescentAlgebra(QQ, 4)
sage: bases = DescentAlgebraBases(DA)
sage: bases.super_categories()
[Category of finite dimensional algebras with basis over Rational Field,
Category of realizations of Descent algebra of 4 over Rational Field]

```

```

>>> from sage.all import *
>>> from sage.combinat.descent_algebra import DescentAlgebraBases
>>> DA = DescentAlgebra(QQ, Integer(4))
>>> bases = DescentAlgebraBases(DA)
>>> bases.super_categories()
[Category of finite dimensional algebras with basis over Rational Field,
Category of realizations of Descent algebra of 4 over Rational Field]

```

5.7 Down-Up Algebras

AUTHORS:

- Travis Scrimshaw (2023-4): initial version

class sage.algebras.down_up_algebra.DownUpAlgebra(alpha, beta, gamma, base_ring)

Bases: CombinatorialFreeModule

The down-up algebra.

Let R be a commutative ring, and let $\alpha, \beta, \gamma \in R$. The *down-up algebra* is the associative unital algebra $DU(\alpha, \beta, \gamma)$ generated by d, u with relations

$$\begin{aligned}d^2u &= \alpha dud + \beta ud^2 + \gamma d, \\ du^2 &= \alpha udu + \beta u^2d + \gamma u.\end{aligned}$$

The down-up algebra has a PBW-type basis given by

$$\{u^i(du)^j d^k \mid i, j, k \in \mathbf{Z}_{\geq 0}\}.$$

This algebra originates in the study of posets. For a poset P , we define operators acting on $R[P]$ by

$$d(y) = \sum_x x \quad u(y) = \sum_z z,$$

where y covers x and z covers y . For r -differential posets we have $du - ud = r1$, and thus it affords a representation of a *Weyl algebra*. This Weyl algebra is obtained as the quotient of $DU(0, 1, 2r)$ by the ideal generated by $du - ud - r$. For a (q, r) -differential poset, we have the d and u operators satisfying

$$\begin{aligned}d^2u &= q(q+1)dud - q^3ud^2 + rd, \\ du^2 &= q(q+1)udu - q^3u^2d + ru,\end{aligned}$$

or $\alpha = q(q+1)$, $\beta = -q^3$, and $\gamma = r$. Specializing $q = -1$ recovers the r -differential poset relation.

Two other noteworthy quotients are:

- the q -Weyl algebra from $DU(0, q^2, q+1)$ by the ideal generated by $du - qud - 1$, and
- the quantum plane $R_q[d, u]$, where $du = qud$, from $DU(2q, -q^2, 0)$ by the ideal generated by $du - qud$.

EXAMPLES:

We begin by constructing the down-up algebra and perform some basic computations:

```
sage: R.<a,b,g> = QQ[]
sage: DU = algebras.DownUp(a, b, g)
sage: d, u = DU.gens()
sage: d * u
(d*u)
sage: u * d
u*d
sage: d^2 * u
b*u*d^2 + a*(d*u)*d + g*d
sage: d * u^2
b*u^2*d + a*u*(d*u) + g*u
```

```

>>> from sage.all import *
>>> R = QQ['a, b, g']; (a, b, g,) = R._first_ngens(3)
>>> DU = algebras.DownUp(a, b, g)
>>> d, u = DU.gens()
>>> d * u
(d*u)
>>> u * d
u*d
>>> d**Integer(2) * u
b*u*d^2 + a*(d*u)*d + g*d
>>> d * u**Integer(2)
b*u^2*d + a*u*(d*u) + g*u

```

We verify some examples of Proposition 3.5 in [BR1998], which states that the 0-th degree part is commutative:

```

sage: DU0 = [u^i * (d*u)^j * d^i for i,j in
.....:         cartesian_product([range(3), range(3)])]
sage: all(x.degree() == 0 for x in DU0)
True
sage: all(x * y == y * x for x, y in cartesian_product([DU0, DU0]))
True

```

```

>>> from sage.all import *
>>> DU0 = [u**i * (d*u)**j * d**i for i,j in
...         cartesian_product([range(Integer(3)), range(Integer(3))])]
>>> all(x.degree() == Integer(0) for x in DU0)
True
>>> all(x * y == y * x for x, y in cartesian_product([DU0, DU0]))
True

```

We verify that $DU(2, -1, \gamma)$ can be described as the universal enveloping algebra of the 3-dimensional Lie algebra spanned by x, y, z satisfying $z = [x, y]$, $[x, z] = \gamma x$, and $[z, y] = \gamma y$:

```

sage: R.<g> = QQ[]
sage: L = LieAlgebra(R, {('x','y'): {'z': 1}, ('x','z'): {'x': g}, ('z','y'): {'y'
->': g}},
.....:         names='x,y,z')
sage: x, y, z = L.basis()
sage: (L[x, y], L[x, z], L[z, y])
(z, g*x, g*y)
sage: x, y, z = L.pbw_basis().gens()
sage: x^2*y - 2*x*y*x + y*x^2 == g*x
True
sage: x*y^2 - 2*y*x*y + y^2*x == g*y
True
sage: DU = algebras.DownUp(2, -1, g)
sage: d, u = DU.gens()
sage: d^2*u - 2*d*u*d + u*d^2 == g*d
True
sage: d*u^2 - 2*u*d*u + u^2*d == g*u
True

```

```

>>> from sage.all import *
>>> R = QQ['g']; (g,) = R._first_ngens(1)
>>> L = LieAlgebra(R, {'x', 'y': {'z': Integer(1)}, ('x', 'z'): {'x': g}, ('z', 'y'
↪ 'x'): {'y': g}},
...                 names='x,y,z')
>>> x, y, z = L.basis()
>>> (L[x, y], L[x, z], L[z, y])
(z, g*x, g*y)
>>> x, y, z = L.pbw_basis().gens()
>>> x**Integer(2)*y - Integer(2)*x*y*x + y*x**Integer(2) == g*x
True
>>> x*y**Integer(2) - Integer(2)*y*x*y + y**Integer(2)*x == g*y
True
>>> DU = algebras.DownUp(Integer(2), -Integer(1), g)
>>> d, u = DU.gens()
>>> d**Integer(2)*u - Integer(2)*d*u*d + u*d**Integer(2) == g*d
True
>>> d*u**Integer(2) - Integer(2)*u*d*u + u**Integer(2)*d == g*u
True

```

Young's lattice is known to be a differential poset. Thus we can construct a representation of $DU(0, 1, 2)$ on this poset (which gives a proof that Fomin's `growth diagrams` are equivalent to edge local rules or shadow lines construction for `RSK()`):

```

sage: DU = algebras.DownUp(0, 1, 2)
sage: d, u = DU.gens()
sage: d^2*u == 0*d*u*d + 1*u*d*d + 2*d
True
sage: d*u^2 == 0*u*d*u + 1*u*u*d + 2*u
True

sage: YL = CombinatorialFreeModule(DU.base_ring(), Partitions())
sage: def d_action(la):
....:     return YL.sum_of_monomials(la.remove_cell(*c) for c in la.removable_
↪ cells())
sage: def u_action(la):
....:     return YL.sum_of_monomials(la.add_cell(*c) for c in la.addable_cells())
sage: D = YL.module_morphism(on_basis=d_action, codomain=YL)
sage: U = YL.module_morphism(on_basis=u_action, codomain=YL)
sage: for la in PartitionsInBox(5, 5):
....:     b = YL.basis()[la]
....:     assert (D*D*U)(b) == 0*(D*U*D)(b) + 1*(U*D*D)(b) + 2*D(b)
....:     assert (D*U*U)(b) == 0*(U*D*U)(la) + 1*(U*U*D)(b) + 2*U(b)
....:     assert (D*U)(b) == (U*D)(b) + b # the Weyl algebra relation

```

```

>>> from sage.all import *
>>> DU = algebras.DownUp(Integer(0), Integer(1), Integer(2))
>>> d, u = DU.gens()
>>> d**Integer(2)*u == Integer(0)*d*u*d + Integer(1)*u*d*d + Integer(2)*d
True
>>> d*u**Integer(2) == Integer(0)*u*d*u + Integer(1)*u*u*d + Integer(2)*u
True

```

(continues on next page)

(continued from previous page)

```

>>> YL = CombinatorialFreeModule(DU.base_ring(), Partitions())
>>> def d_action(la):
...     return YL.sum_of_monomials(la.remove_cell(*c) for c in la.removable_
↳ cells())
>>> def u_action(la):
...     return YL.sum_of_monomials(la.add_cell(*c) for c in la.addable_cells())
>>> D = YL.module_morphism(on_basis=d_action, codomain=YL)
>>> U = YL.module_morphism(on_basis=u_action, codomain=YL)
>>> for la in PartitionsInBox(Integer(5), Integer(5)):
...     b = YL.basis()[la]
...     assert (D*D*U)(b) == Integer(0)*(D*U*D)(b) + Integer(1)*(U*D*D)(b) +
↳ Integer(2)*D(b)
...     assert (D*U*U)(b) == Integer(0)*(U*D*U)(la) + Integer(1)*(U*U*D)(b) +
↳ Integer(2)*U(b)
...     assert (D*U)(b) == (U*D)(b) + b # the Weyl algebra relation

```

Todo

Implement the homogenized version.

REFERENCES:

- [BR1998]
- [CM2000]

algebra_generators()

Return the algebra generators of self.

EXAMPLES:

```

sage: DU = algebras.DownUp(2, 3, 4)
sage: dict(DU.algebra_generators())
{'d': d, 'u': u}

```

```

>>> from sage.all import *
>>> DU = algebras.DownUp(Integer(2), Integer(3), Integer(4))
>>> dict(DU.algebra_generators())
{'d': d, 'u': u}

```

degree_on_basis(m)

Return the degree of the basis element indexed by m.

EXAMPLES:

```

sage: R.<a,b,g> = QQ[]
sage: DU = algebras.DownUp(a, b, g)
sage: I = DU.indices()
sage: DU.degree_on_basis(I([0, 3, 2]))
-2
sage: DU.degree_on_basis(I([2, 3, 0]))
2
sage: DU.degree_on_basis(I([2, 0, 3]))

```

(continues on next page)

(continued from previous page)

```
-1
sage: DU.degree_on_basis(I([3, 10, 3]))
0
```

```
>>> from sage.all import *
>>> R = QQ['a, b, g']; (a, b, g,) = R._first_ngens(3)
>>> DU = algebras.DownUp(a, b, g)
>>> I = DU.indices()
>>> DU.degree_on_basis(I([Integer(0), Integer(3), Integer(2)]))
-2
>>> DU.degree_on_basis(I([Integer(2), Integer(3), Integer(0)]))
2
>>> DU.degree_on_basis(I([Integer(2), Integer(0), Integer(3)]))
-1
>>> DU.degree_on_basis(I([Integer(3), Integer(10), Integer(3)]))
0
```

gens()

Return the generators of self.

EXAMPLES:

```
sage: DU = algebras.DownUp(2, 3, 4)
sage: DU.gens()
(d, u)
```

```
>>> from sage.all import *
>>> DU = algebras.DownUp(Integer(2), Integer(3), Integer(4))
>>> DU.gens()
(d, u)
```

one_basis()

Return the index of the basis element of 1.

EXAMPLES:

```
sage: DU = algebras.DownUp(2, 3, 4)
sage: DU.one_basis()
(0, 0, 0)
```

```
>>> from sage.all import *
>>> DU = algebras.DownUp(Integer(2), Integer(3), Integer(4))
>>> DU.one_basis()
(0, 0, 0)
```

product_on_basis(m1, m2)

Return the product of the basis elements indexed by m1 and m2.

EXAMPLES:

```
sage: R.<a,b,g> = QQ[]
sage: DU = algebras.DownUp(a, b, g)
```

(continues on next page)

(continued from previous page)

```

sage: I = DU.indices()
sage: DU.product_on_basis(I([2,0,0]), I([4,0,0]))
u^6
sage: DU.product_on_basis(I([2,0,0]), I([0,4,0]))
u^2*(d*u)^4
sage: DU.product_on_basis(I([2,0,0]), I([0,0,4]))
u^2*d^4
sage: DU.product_on_basis(I([0,2,0]), I([0,4,0]))
(d*u)^6
sage: DU.product_on_basis(I([0,2,0]), I([0,0,4]))
(d*u)^2*d^4
sage: DU.product_on_basis(I([0,0,2]), I([0,0,4]))
d^6
sage: DU.product_on_basis(I([5,3,1]), I([1,0,4]))
u^5*(d*u)^4*d^4

sage: DU.product_on_basis(I([0,1,0]), I([1,0,0]))
b*u^2*d + a*u*(d*u) + g*u
sage: DU.product_on_basis(I([0,0,2]), I([1,0,0]))
b*u*d^2 + a*(d*u)*d + g*d
sage: DU.product_on_basis(I([0,0,1]), I([2,0,0]))
b*u^2*d + a*u*(d*u) + g*u
sage: DU.product_on_basis(I([0,0,1]), I([0,1,0]))
b*u*d^2 + a*(d*u)*d + g*d

sage: DU.product_on_basis(I([0,1,0]), I([3,0,0]))
(a^2*b+b^2)*u^4*d + (a^3+2*a*b)*u^3*(d*u) + (a^2*g+a*g+b*g+g)*u^3
sage: DU.product_on_basis(I([1,1,3]), I([0,1,1]))
(a^2*b^2+b^3)*u^3*d^6 + (a^3*b+a*b^2)*u^2*(d*u)*d^5 + (a^2*b*g+b^2*g)*u^2*d^5
+ (a^3+2*a*b)*u*(d*u)^2*d^4 + (a^2*g+a*g+b*g+g)*u*(d*u)*d^4

```

```

>>> from sage.all import *
>>> R = QQ['a, b, g']; (a, b, g) = R._first_ngens(3)
>>> DU = algebras.DownUp(a, b, g)
>>> I = DU.indices()
>>> DU.product_on_basis(I([Integer(2), Integer(0), Integer(0)]), I([Integer(4),
↪ Integer(0), Integer(0)]))
u^6
>>> DU.product_on_basis(I([Integer(2), Integer(0), Integer(0)]), I([Integer(0),
↪ Integer(4), Integer(0)]))
u^2*(d*u)^4
>>> DU.product_on_basis(I([Integer(2), Integer(0), Integer(0)]), I([Integer(0),
↪ Integer(0), Integer(4)]))
u^2*d^4
>>> DU.product_on_basis(I([Integer(0), Integer(2), Integer(0)]), I([Integer(0),
↪ Integer(4), Integer(0)]))
(d*u)^6
>>> DU.product_on_basis(I([Integer(0), Integer(2), Integer(0)]), I([Integer(0),
↪ Integer(0), Integer(4)]))
(d*u)^2*d^4
>>> DU.product_on_basis(I([Integer(0), Integer(0), Integer(2)]), I([Integer(0),
↪ Integer(0), Integer(4)]))

```

(continues on next page)

(continued from previous page)

```

d^6
>>> DU.product_on_basis(I([Integer(5), Integer(3), Integer(1)]), I([Integer(1),
↪ Integer(0), Integer(4)]))
u^5*(d*u)^4*d^4

>>> DU.product_on_basis(I([Integer(0), Integer(1), Integer(0)]), I([Integer(1),
↪ Integer(0), Integer(0)]))
b*u^2*d + a*u*(d*u) + g*u
>>> DU.product_on_basis(I([Integer(0), Integer(0), Integer(2)]), I([Integer(1),
↪ Integer(0), Integer(0)]))
b*u*d^2 + a*(d*u)*d + g*d
>>> DU.product_on_basis(I([Integer(0), Integer(0), Integer(1)]), I([Integer(2),
↪ Integer(0), Integer(0)]))
b*u^2*d + a*u*(d*u) + g*u
>>> DU.product_on_basis(I([Integer(0), Integer(0), Integer(1)]), I([Integer(0),
↪ Integer(1), Integer(0)]))
b*u*d^2 + a*(d*u)*d + g*d

>>> DU.product_on_basis(I([Integer(0), Integer(1), Integer(0)]), I([Integer(3),
↪ Integer(0), Integer(0)]))
(a^2*b+b^2)*u^4*d + (a^3+2*a*b)*u^3*(d*u) + (a^2*g+a*g+b*g+g)*u^3
>>> DU.product_on_basis(I([Integer(1), Integer(1), Integer(3)]), I([Integer(0),
↪ Integer(1), Integer(1)]))
(a^2*b^2+b^3)*u^3*d^6 + (a^3*b+a*b^2)*u^2*(d*u)*d^5 + (a^2*b*g+b^2*g)*u^2*d^5
+ (a^3+2*a*b)*u*(d*u)^2*d^4 + (a^2*g+a*g+b*g+g)*u*(d*u)*d^4

```

verma_module(*la*)Return the *Verma module* $V(\lambda)$ of self.

EXAMPLES:

```

sage: R.<a,b,g> = QQ[]
sage: DU = algebras.DownUp(a, b, g)
sage: DU.verma_module(5)
Verma module of weight 5 of Down-Up algebra with parameters (a, b, g)
over Multivariate Polynomial Ring in a, b, g over Rational Field

```

```

>>> from sage.all import *
>>> R = QQ['a, b, g']; (a, b, g,) = R._first_ngens(3)
>>> DU = algebras.DownUp(a, b, g)
>>> DU.verma_module(Integer(5))
Verma module of weight 5 of Down-Up algebra with parameters (a, b, g)
over Multivariate Polynomial Ring in a, b, g over Rational Field

```

class sage.algebras.down_up_algebra.**VermaModule**(*DU, la*)Bases: *CombinatorialFreeModule*The Verma module $V(\lambda)$ of a down-up algebra.The Verma module $V(\lambda)$ for the down-up algebra generated by d, u is the span of $\{v_n \mid n \in \mathbb{Z}_{\geq 0}\}$ satisfying the relations

$$d \cdot v_n = \lambda_{n-1} v_{n-1}, \quad u \cdot v_n = v_{n+1},$$

where $\lambda_n = \alpha\lambda_{n-1} + \beta\lambda_{n-2} + \gamma$ and we set $\lambda_0 = \lambda$ and $\lambda_{-1} = 0$.

By Proposition 2.4 in [BR1998], $V(\lambda)$ is simple if and only if $\lambda_n \neq 0$ for all $n \geq 0$. Moreover, a maximal submodule is spanned by $\{v_n \mid n > m\}$, where m is the minimal index such that $\lambda_m = 0$. Moreover, this is unique unless $\gamma = \lambda = 0$.

EXAMPLES:

```
sage: R.<a,b> = QQ[]
sage: DU = algebras.DownUp(0, b, 1)
sage: d, u = DU.gens()
sage: V = DU.verma_module(a)
sage: list(V.weights()[:6])
[a, 1, a*b + 1, b + 1, a*b^2 + b + 1, b^2 + b + 1]
sage: v = V.basis()
sage: d^2 * v[2]
a*v[0]
sage: d * (d * v[2])
a*v[0]
```

```
>>> from sage.all import *
>>> R = QQ['a, b']; (a, b,) = R._first_ngens(2)
>>> DU = algebras.DownUp(Integer(0), b, Integer(1))
>>> d, u = DU.gens()
>>> V = DU.verma_module(a)
>>> list(V.weights()[:Integer(6)])
[a, 1, a*b + 1, b + 1, a*b^2 + b + 1, b^2 + b + 1]
>>> v = V.basis()
>>> d**Integer(2) * v[Integer(2)]
a*v[0]
>>> d * (d * v[Integer(2)])
a*v[0]
```

The weight is computed by looking at the scalars associated to the action of du and ud :

```
sage: d*u * v[3]
(b+1)*v[3]
sage: u*d * v[3]
(a*b+1)*v[3]
sage: v[3].weight()
(b + 1, a*b + 1)
```

```
>>> from sage.all import *
>>> d*u * v[Integer(3)]
(b+1)*v[3]
>>> u*d * v[Integer(3)]
(a*b+1)*v[3]
>>> v[Integer(3)].weight()
(b + 1, a*b + 1)
```

An $U(\mathfrak{sl}_2)$ example:

```
sage: DU = algebras.DownUp(2, -1, -2)
sage: d, u = DU.gens()
```

(continues on next page)

(continued from previous page)

```

sage: V = DU.verma_module(5)
sage: list(V.weights()[:10])
[5, 8, 9, 8, 5, 0, -7, -16, -27, -40]
sage: v6 = V.basis()[6]
sage: d * v6
0
sage: [V.basis()[i].weight() for i in range(6)]
[(5, 0), (8, 5), (9, 8), (8, 9), (5, 8), (0, 5)]

```

```

>>> from sage.all import *
>>> DU = algebras.DownUp(Integer(2), -Integer(1), -Integer(2))
>>> d, u = DU.gens()
>>> V = DU.verma_module(Integer(5))
>>> list(V.weights()[:Integer(10)])
[5, 8, 9, 8, 5, 0, -7, -16, -27, -40]
>>> v6 = V.basis()[Integer(6)]
>>> d * v6
0
>>> [V.basis()[i].weight() for i in range(Integer(6))]
[(5, 0), (8, 5), (9, 8), (8, 9), (5, 8), (0, 5)]

```

Note that these are the same $\mathfrak{sl}_2$ weights from the usual construction of the irreducible representation $V(5)$ (but they are different as $\mathfrak{gl}_2$ weights):

```

sage: B = crystals.Tableaux(['A',1], shape=[5]) #_
↪needs sage.graphs
sage: [b.weight() for b in B] #_
↪needs sage.graphs
[(5, 0), (4, 1), (3, 2), (2, 3), (1, 4), (0, 5)]

```

```

>>> from sage.all import *
>>> B = crystals.Tableaux(['A',Integer(1)], shape=[Integer(5)]) #_
↪ # needs sage.graphs
>>> [b.weight() for b in B] #_
↪needs sage.graphs
[(5, 0), (4, 1), (3, 2), (2, 3), (1, 4), (0, 5)]

```

An example with periodic weights (see Theorem 2.13 of [BR1998]):

```

sage: # needs sage.rings.number_field
sage: k.<z6> = CyclotomicField(6)
sage: a1 = z6 + 1
sage: (a1 - 1)^6 == 1
True
sage: DU = algebras.DownUp(a1, 1-a1, 0)
sage: V = DU.verma_module(5)
sage: list(V.weights()[:8])
[5, 5*z6 + 5, 10*z6, 10*z6 - 5, 5*z6 - 5, 0, 5, 5*z6 + 5]

```

```

>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> k = CyclotomicField(Integer(6), names=('z6',)); (z6,) = k._first_ngens(1)

```

(continues on next page)

(continued from previous page)

```

>>> a1 = z6 + Integer(1)
>>> (a1 - Integer(1))*Integer(6) == Integer(1)
True
>>> DU = algebras.DownUp(a1, Integer(1)-a1, Integer(0))
>>> V = DU.verma_module(Integer(5))
>>> list(V.weights()[:Integer(8)])
[5, 5*z6 + 5, 10*z6, 10*z6 - 5, 5*z6 - 5, 0, 5, 5*z6 + 5]

```

class ElementBases: `IndexedFreeModuleElement`

An element of a Verma module of a down-up algebra.

is_weight_vector()Return if `self` is a weight vector.

EXAMPLES:

```

sage: DU = algebras.DownUp(2, -1, -2)
sage: V = DU.verma_module(5)
sage: V.zero().is_weight_vector()
False
sage: B = V.basis()
sage: [B[i].weight() for i in range(6)]
[(5, 0), (8, 5), (9, 8), (8, 9), (5, 8), (0, 5)]
sage: B[5].is_weight_vector()
True
sage: v = B[0] + B[1]
sage: v.is_weight_vector()
False

sage: DU = algebras.DownUp(2, -1, 0)
sage: V = DU.verma_module(0)
sage: B = V.basis()
sage: v = sum(i*B[i] for i in range(1,5))
sage: v.is_weight_vector()
True

```

```

>>> from sage.all import *
>>> DU = algebras.DownUp(Integer(2), -Integer(1), -Integer(2))
>>> V = DU.verma_module(Integer(5))
>>> V.zero().is_weight_vector()
False
>>> B = V.basis()
>>> [B[i].weight() for i in range(Integer(6))]
[(5, 0), (8, 5), (9, 8), (8, 9), (5, 8), (0, 5)]
>>> B[Integer(5)].is_weight_vector()
True
>>> v = B[Integer(0)] + B[Integer(1)]
>>> v.is_weight_vector()
False

>>> DU = algebras.DownUp(Integer(2), -Integer(1), Integer(0))

```

(continues on next page)

(continued from previous page)

```

>>> V = DU.verma_module(Integer(0))
>>> B = V.basis()
>>> v = sum(i*B[i] for i in range(Integer(1), Integer(5)))
>>> v.is_weight_vector()
True

```

weight()

Return the weight of `self`.

For v_n , this is the vector with the pair $(\lambda_n, \lambda_{n-1})$.

EXAMPLES:

```

sage: R.<a,b,g> = QQ[]
sage: DU = algebras.DownUp(a, b, g)
sage: V = DU.verma_module(5)
sage: B = V.basis()
sage: B[0].weight()
(5, 0)
sage: B[1].weight()
(5*a + g, 5)
sage: B[2].weight()
(5*a^2 + a*g + 5*b + g, 5*a + g)

sage: V.zero().weight()
Traceback (most recent call last):
...
ValueError: the zero element does not have well-defined weight
sage: (B[0] + B[1]).weight()
Traceback (most recent call last):
...
ValueError: not a weight vector

```

```

>>> from sage.all import *
>>> R = QQ['a, b, g']; (a, b, g) = R._first_ngens(3)
>>> DU = algebras.DownUp(a, b, g)
>>> V = DU.verma_module(Integer(5))
>>> B = V.basis()
>>> B[Integer(0)].weight()
(5, 0)
>>> B[Integer(1)].weight()
(5*a + g, 5)
>>> B[Integer(2)].weight()
(5*a^2 + a*g + 5*b + g, 5*a + g)

>>> V.zero().weight()
Traceback (most recent call last):
...
ValueError: the zero element does not have well-defined weight
>>> (B[Integer(0)] + B[Integer(1)]).weight()
Traceback (most recent call last):
...
ValueError: not a weight vector

```

highest_weight_vector()

Return the highest weight vector of `self` that generates `self` as a down-up module.

EXAMPLES:

```
sage: DU = algebras.DownUp(1, 2, 3)
sage: V = DU.verma_module(5)
sage: V.highest_weight_vector()
v[0]
```

```
>>> from sage.all import *
>>> DU = algebras.DownUp(Integer(1), Integer(2), Integer(3))
>>> V = DU.verma_module(Integer(5))
>>> V.highest_weight_vector()
v[0]
```

weights()

Return the sequence of weights $(\lambda_n)_{n=0}^\infty$.

EXAMPLES:

```
sage: R.<a,b,g> = QQ[]
sage: DU = algebras.DownUp(a, b, g)
sage: V = DU.verma_module(5)
sage: V.weights()
lazy list [5, 5*a + g, 5*a^2 + a*g + 5*b + g, ...]
```

```
sage: V = DU.verma_module(0)
sage: DU = algebras.DownUp(a, 1-a, 0)
sage: V = DU.verma_module(0)
sage: V.weights()
lazy list [0, 0, 0, ...]
```

```
>>> from sage.all import *
>>> R = QQ['a, b, g']; (a, b, g) = R._first_ngens(3)
>>> DU = algebras.DownUp(a, b, g)
>>> V = DU.verma_module(Integer(5))
>>> V.weights()
lazy list [5, 5*a + g, 5*a^2 + a*g + 5*b + g, ...]

>>> V = DU.verma_module(Integer(0))
>>> DU = algebras.DownUp(a, Integer(1)-a, Integer(0))
>>> V = DU.verma_module(Integer(0))
>>> V.weights()
lazy list [0, 0, 0, ...]
```

We reproduce the Fibonacci numbers example from [BR1998]:

```
sage: R.<la> = QQ[]
sage: DU = algebras.DownUp(1, 1, 0, R)
sage: V = DU.verma_module(la)
sage: list(V.weights()[:11])
[la, la, 2*la, 3*la, 5*la, 8*la, 13*la, 21*la, 34*la, 55*la, 89*la]
```



```

>>> from sage.all import *
>>> R = QQ['la']; (la,) = R._first_ngens(1)
>>> DU = algebras.DownUp(Integer(1), Integer(1), Integer(0), R)
>>> V = DU.verma_module(la)
>>> list(V.weights()[Integer(11)])
[la, la, 2*la, 3*la, 5*la, 8*la, 13*la, 21*la, 34*la, 55*la, 89*la]

```

5.8 Fusion Rings

5.8.1 Fusion Rings

```

class sage.algebras.fusion_rings.fusion_ring.FusionRing(ct, base_ring=Integer Ring, prefix=None,
                                                         style='lattice', k=None, conjugate=False,
                                                         cyclotomic_order=None,
                                                         fusion_labels=None,
                                                         inject_variables=False)

```

Bases: `WeylCharacterRing`

Return the Fusion Ring (Verlinde Algebra) of level k .

INPUT:

- `ct` – the Cartan type of a simple (finite-dimensional) Lie algebra
- `k` – nonnegative integer
- `conjugate` – (default: `False`) set `True` to obtain the complex conjugate ring
- `cyclotomic_order` – (default: computed depending on `ct` and `k`)
- `fusion_labels` – (default: `None`) either a tuple of strings to use as labels of the basis of simple objects, or a string from which the labels will be constructed
- `inject_variables` – (default: `False`) use with `fusion_labels`. If `inject_variables` is `True`, the fusion labels will be variables that can be accessed from the command line

The cyclotomic order is an integer N such that all computations will return elements of the cyclotomic field of N -th roots of unity. Normally you will never need to change this but consider changing it if `root_of_unity()` raises a `ValueError`.

This algebra has a basis (sometimes called *primary fields* but here called *simple objects*) indexed by the weights of level $\leq k$. These arise as the fusion algebras of Wess-Zumino-Witten (WZW) conformal field theories, or as Grothendieck groups of tilting modules for quantum groups at roots of unity. The `FusionRing` class is implemented as a variant of the `WeylCharacterRing`.

REFERENCES:

- [BaKi2001] Chapter 3
- [DFMS1996] Chapter 16
- [EGNO2015] Chapter 8
- [Feingold2004]
- [Fuchs1994]
- [Row2006]
- [Walton1990]

- [Wan2010]

EXAMPLES:

```
sage: A22 = FusionRing("A2", 2)
sage: [f1, f2] = A22.fundamental_weights()
sage: M = [A22(x) for x in [0*f1, 2*f1, 2*f2, f1+f2, f2, f1]]
sage: [M[3] * x for x in M]
[A22(1,1),
 A22(0,1),
 A22(1,0),
 A22(0,0) + A22(1,1),
 A22(0,1) + A22(2,0),
 A22(1,0) + A22(0,2)]
```

```
>>> from sage.all import *
>>> A22 = FusionRing("A2", Integer(2))
>>> [f1, f2] = A22.fundamental_weights()
>>> M = [A22(x) for x in [Integer(0)*f1, Integer(2)*f1, Integer(2)*f2, f1+f2, f2,
↳ f1]]
>>> [M[Integer(3)] * x for x in M]
[A22(1,1),
 A22(0,1),
 A22(1,0),
 A22(0,0) + A22(1,1),
 A22(0,1) + A22(2,0),
 A22(1,0) + A22(0,2)]
```

You may assign your own labels to the basis elements. In the next example, we create the $SO(5)$ fusion ring of level 2, check the weights of the basis elements, then assign new labels to them while injecting them into the global namespace:

```
sage: B22 = FusionRing("B2", 2)
sage: b = [B22(x) for x in B22.get_order()]; b
[B22(0,0), B22(1,0), B22(0,1), B22(2,0), B22(1,1), B22(0,2)]
sage: [x.weight() for x in b]
[(0, 0), (1, 0), (1/2, 1/2), (2, 0), (3/2, 1/2), (1, 1)]
sage: B22.fusion_labels(['I0', 'Y1', 'X', 'Z', 'Xp', 'Y2'], inject_variables=True)
sage: b = [B22(x) for x in B22.get_order()]; b
[I0, Y1, X, Z, Xp, Y2]
sage: [(x, x.weight()) for x in b]
[(I0, (0, 0)),
 (Y1, (1, 0)),
 (X, (1/2, 1/2)),
 (Z, (2, 0)),
 (Xp, (3/2, 1/2)),
 (Y2, (1, 1))]
sage: X * Y1
X + Xp
sage: Z * Z
I0
```

```
>>> from sage.all import *
>>> B22 = FusionRing("B2", Integer(2))
```

(continues on next page)

(continued from previous page)

```

>>> b = [B22(x) for x in B22.get_order()]; b
[B22(0,0), B22(1,0), B22(0,1), B22(2,0), B22(1,1), B22(0,2)]
>>> [x.weight() for x in b]
[(0, 0), (1, 0), (1/2, 1/2), (2, 0), (3/2, 1/2), (1, 1)]
>>> B22.fusion_labels(['I0', 'Y1', 'X', 'Z', 'Xp', 'Y2'], inject_variables=True)
>>> b = [B22(x) for x in B22.get_order()]; b
[I0, Y1, X, Z, Xp, Y2]
>>> [(x, x.weight()) for x in b]
[(I0, (0, 0)),
 (Y1, (1, 0)),
 (X, (1/2, 1/2)),
 (Z, (2, 0)),
 (Xp, (3/2, 1/2)),
 (Y2, (1, 1))]
>>> X * Y1
X + Xp
>>> Z * Z
I0

```

A fixed order of the basis keys is available with `get_order()`. This is the order used by methods such as `s_matrix()`. You may use `CombinatorialFreeModule.set_order()` to reorder the basis:

```

sage: B22.set_order([x.weight() for x in [I0, Y1, Y2, X, Xp, Z]])
sage: [B22(x) for x in B22.get_order()]
[I0, Y1, Y2, X, Xp, Z]

```

```

>>> from sage.all import *
>>> B22.set_order([x.weight() for x in [I0, Y1, Y2, X, Xp, Z]])
>>> [B22(x) for x in B22.get_order()]
[I0, Y1, Y2, X, Xp, Z]

```

To reset the labels, you may run `fusion_labels()` with no parameter:

```

sage: B22.fusion_labels()
sage: [B22(x) for x in B22.get_order()]
[B22(0,0), B22(1,0), B22(0,2), B22(0,1), B22(1,1), B22(2,0)]

```

```

>>> from sage.all import *
>>> B22.fusion_labels()
>>> [B22(x) for x in B22.get_order()]
[B22(0,0), B22(1,0), B22(0,2), B22(0,1), B22(1,1), B22(2,0)]

```

To reset the order to the default, simply set it to the list of basis element keys:

```

sage: B22.set_order(B22.basis().keys().list())
sage: [B22(x) for x in B22.get_order()]
[B22(0,0), B22(1,0), B22(0,1), B22(2,0), B22(1,1), B22(0,2)]

```

```

>>> from sage.all import *
>>> B22.set_order(B22.basis().keys().list())
>>> [B22(x) for x in B22.get_order()]
[B22(0,0), B22(1,0), B22(0,1), B22(2,0), B22(1,1), B22(0,2)]

```

The fusion ring has a number of methods that reflect its role as the Grothendieck ring of a *modular tensor category* (MTC). These include twist methods `Element.twist()` and `Element.ribbon()` for its elements related to the ribbon structure, and the S -matrix `s_ij()`.

There are two natural normalizations of the S -matrix. Both are explained in Chapter 3 of [BaKi2001]. The one that is computed by the method `s_matrix()`, or whose individual entries are computed by `s_ij()` is denoted $\tilde{s}$ in [BaKi2001]. It is not unitary.

The unitary S -matrix is $s = D^{-1/2}\tilde{s}$ where

$$D = \sum_V d_i(V)^2.$$

The sum is over all simple objects V with $d_i(V)$ the *quantum dimension*. We will call quantity D the *global quantum dimension* and $\sqrt{D}$ the *total quantum order*. They are computed by `global_q_dimension()` and `total_q_order()`. The unitary S -matrix s may be obtained using `s_matrix()` with the option `unitary=True`.

Let us check the Verlinde formula, which is [DFMS1996] (16.3). This famous identity states that

$$N_{ij}^k = \sum_l \frac{s(i, l) s(j, l) \overline{s(k, l)}}{s(I, l)},$$

where N_{ij}^k are the fusion coefficients, i.e. the structure constants of the fusion ring, and I is the unit object. The S -matrix has the property that if i^* denotes the dual object of i , implemented in Sage as `i.dual()`, then

$$s(i^*, j) = s(i, j^*) = \overline{s(i, j)}.$$

This is equation (16.5) in [DFMS1996]. Thus with $N_{ijk} = N_{ij}^{k^*}$ the Verlinde formula is equivalent to

$$N_{ijk} = \sum_l \frac{s(i, l) s(j, l) s(k, l)}{s(I, l)},$$

In this formula s is the normalized unitary S -matrix denoted s in [BaKi2001]. We may define a function that corresponds to the right-hand side, except using $\tilde{s}$ instead of s :

```
sage: def V(i, j, k):
.....:     R = i.parent()
.....:     return sum(R.s_ij(i, l) * R.s_ij(j, l) * R.s_ij(k, l) / R.s_ij(R.one(), l),
.....:               ↪l)
.....:         for l in R.basis())
```

```
>>> from sage.all import *
>>> def V(i, j, k):
...     R = i.parent()
...     return sum(R.s_ij(i, l) * R.s_ij(j, l) * R.s_ij(k, l) / R.s_ij(R.one(), l)
...               for l in R.basis())
```

This does not produce `self.N_ijk(i, j, k)` exactly, because of the missing normalization factor. The following code to check the Verlinde formula takes this into account:

```
sage: def test_verlinde(R):
.....:     b0 = R.one()
.....:     c = R.global_q_dimension()
.....:     return all(V(i, j, k) == c * R.N_ijk(i, j, k) for i in R.basis()
.....:               for j in R.basis() for k in R.basis())
```

```
>>> from sage.all import *
>>> def test_verlinde(R):
...     b0 = R.one()
...     c = R.global_q_dimension()
...     return all(V(i, j, k) == c * R.N_ijk(i, j, k) for i in R.basis()
...               for j in R.basis() for k in R.basis())
```

Every fusion ring should pass this test:

```
sage: test_verlinde(FusionRing("A2", 1))
True
sage: test_verlinde(FusionRing("B4", 2)) # long time (.56s)
True
```

```
>>> from sage.all import *
>>> test_verlinde(FusionRing("A2", Integer(1)))
True
>>> test_verlinde(FusionRing("B4", Integer(2))) # long time (.56s)
True
```

As an exercise, the reader may verify the examples in Section 5.3 of [RoStWa2009]. Here we check the example of the Ising modular tensor category, which is related to the Belavin, Polyakov, Zamolodchikov minimal model $M(4, 3)$ or to an E_8 coset model. See [DFMS1996] Sections 7.4.2 and 18.4.1. [RoStWa2009] Example 5.3.4 tells us how to construct it as the conjugate of the E_8 level 2 *FusionRing*:

```
sage: I = FusionRing("E8", 2, conjugate=True)
sage: I.fusion_labels(["i0", "p", "s"], inject_variables=True)
sage: b = I.basis().list(); b
[i0, p, s]
sage: Matrix([[x*y for x in b] for y in b]) # long time (.93s)
[  i0      p      s]
[   p     i0      s]
[   s      s i0 + p]
sage: [x.twist() for x in b]
[0, 1, 1/8]
sage: [x.ribbon() for x in b]
[1, -1, zeta128^8]
sage: [I.r_matrix(i, j, k) for (i, j, k) in [(s, s, i0), (p, p, i0), (p, s, s),
↪ (s, p, s), (s, s, p)]]
[-zeta128^56, -1, -zeta128^32, -zeta128^32, zeta128^24]
sage: I.r_matrix(s, s, i0) == I.root_of_unity(-1/8)
True
sage: I.global_q_dimension()
4
sage: I.total_q_order()
2
sage: [x.q_dimension()^2 for x in b]
[1, 1, 2]
sage: I.s_matrix()
[          1          1 -zeta128^48 + zeta128^16]
[          1          1  zeta128^48 - zeta128^16]
[-zeta128^48 + zeta128^16  zeta128^48 - zeta128^16  0]
sage: I.s_matrix().apply_map(lambda x:x^2)
```

(continues on next page)

(continued from previous page)

```
[1 1 2]
[1 1 2]
[2 2 0]
```

```
>>> from sage.all import *
>>> I = FusionRing("E8", Integer(2), conjugate=True)
>>> I.fusion_labels(["i0", "p", "s"], inject_variables=True)
>>> b = I.basis().list(); b
[i0, p, s]
>>> Matrix([[x*y for x in b] for y in b]) # long time (.93s)
[   i0      p      s]
[   p      i0      s]
[   s      s i0 + p]
>>> [x.twist() for x in b]
[0, 1, 1/8]
>>> [x.ribbon() for x in b]
[1, -1, zeta128^8]
>>> [I.r_matrix(i, j, k) for (i, j, k) in [(s, s, i0), (p, p, i0), (p, s, s), (s,
↪p, s), (s, s, p)]]
[-zeta128^56, -1, -zeta128^32, -zeta128^32, zeta128^24]
>>> I.r_matrix(s, s, i0) == I.root_of_unity(-Integer(1)/Integer(8))
True
>>> I.global_q_dimension()
4
>>> I.total_q_order()
2
>>> [x.q_dimension()*Integer(2) for x in b]
[1, 1, 2]
>>> I.s_matrix()
[               1               1 -zeta128^48 + zeta128^16]
[               1               1  zeta128^48 - zeta128^16]
[-zeta128^48 + zeta128^16  zeta128^48 - zeta128^16          0]
>>> I.s_matrix().apply_map(lambda x:x**Integer(2))
[1 1 2]
[1 1 2]
[2 2 0]
```

The term *modular tensor category* refers to the fact that associated with the category there is a projective representation of the modular group $SL(2, \mathbf{Z})$. We recall that this group is generated by

$$S = \begin{pmatrix} & -1 \\ 1 & \end{pmatrix}, \quad T = \begin{pmatrix} 1 & 1 \\ & 1 \end{pmatrix}$$

subject to the relations $(ST)^3 = S^2$, $S^2T = TS^2$, and $S^4 = I$. Let s be the normalized S -matrix, and t the diagonal matrix whose entries are the twists of the simple objects. Let s the unitary S -matrix and t the matrix of twists, and C the conjugation matrix `conj_matrix()`. Let

$$D_+ = \sum_i d_i^2 \theta_i, \quad D_- = d_i^2 \theta_i^{-1},$$

where d_i and θ_i are the quantum dimensions and twists of the simple objects. Let c be the Virasoro central charge, a rational number that is computed in `virasoro_central_charge()`. It is known that

$$\sqrt{\frac{D_+}{D_-}} = e^{i\pi c/4}.$$

It is proved in [BaKi2001] Equation (3.1.17) that

$$(st)^3 = e^{i\pi c/4} s^2, \quad s^2 = C, \quad C^2 = 1, \quad Ct = tC.$$

Therefore $S \mapsto s, T \mapsto t$ is a projective representation of $SL(2, \mathbf{Z})$. Let us confirm these identities for the Fibonacci MTC `FusionRing("G2", 1)`:

```
sage: R = FusionRing("G2", 1)
sage: S = R.s_matrix(unitary=True)
sage: T = R.twists_matrix()
sage: C = R.conj_matrix()
sage: c = R.virasoro_central_charge(); c
14/5
sage: (S*T)^3 == R.root_of_unity(c/4) * S^2
True
sage: S^2 == C
True
sage: C*T == T*C
True
```

```
>>> from sage.all import *
>>> R = FusionRing("G2", Integer(1))
>>> S = R.s_matrix(unitary=True)
>>> T = R.twists_matrix()
>>> C = R.conj_matrix()
>>> c = R.virasoro_central_charge(); c
14/5
>>> (S*T)**Integer(3) == R.root_of_unity(c/Integer(4)) * S**Integer(2)
True
>>> S**Integer(2) == C
True
>>> C*T == T*C
True
```

D_minus (*base_coercion=True*)

Return $\sum d_i^2 \theta_i^{-1}$ where i runs through the simple objects, d_i is the quantum dimension and θ_i is the twist.

This is denoted p_- in [BaKi2001] Chapter 3.

EXAMPLES:

```
sage: E83 = FusionRing("E8", 3, conjugate=True)
sage: [Dp, Dm] = [E83.D_plus(), E83.D_minus()]
sage: Dp*Dm == E83.global_q_dimension()
True
sage: c = E83.virasoro_central_charge(); c
-248/11
sage: Dp*Dm == E83.global_q_dimension()
True
```

```
>>> from sage.all import *
>>> E83 = FusionRing("E8", Integer(3), conjugate=True)
>>> [Dp, Dm] = [E83.D_plus(), E83.D_minus()]
>>> Dp*Dm == E83.global_q_dimension()
```

(continues on next page)

(continued from previous page)

```

True
>>> c = E83.virasoro_central_charge(); c
-248/11
>>> Dp*Dm == E83.global_q_dimension()
True

```

D_plus (*base_coercion=True*)

Return $\sum d_i^2 \theta_i$ where i runs through the simple objects, d_i is the quantum dimension and θ_i is the twist.

This is denoted p_+ in [BaKi2001] Chapter 3.

EXAMPLES:

```

sage: B31 = FusionRing("B3", 1)
sage: Dp = B31.D_plus(); Dp
2*zeta48^13 - 2*zeta48^5
sage: Dm = B31.D_minus(); Dm
-2*zeta48^3
sage: Dp*Dm == B31.global_q_dimension()
True
sage: c = B31.virasoro_central_charge(); c
7/2
sage: Dp/Dm == B31.root_of_unity(c/2)
True

```

```

>>> from sage.all import *
>>> B31 = FusionRing("B3", Integer(1))
>>> Dp = B31.D_plus(); Dp
2*zeta48^13 - 2*zeta48^5
>>> Dm = B31.D_minus(); Dm
-2*zeta48^3
>>> Dp*Dm == B31.global_q_dimension()
True
>>> c = B31.virasoro_central_charge(); c
7/2
>>> Dp/Dm == B31.root_of_unity(c/Integer(2))
True

```

class ElementBases: `Element`

A class for FusionRing elements.

is_simple_object ()

Determine whether `self` is a simple object of the fusion ring.

EXAMPLES:

```

sage: A22 = FusionRing("A2", 2)
sage: x = A22(1, 0); x
A22(1, 0)
sage: x.is_simple_object()
True
sage: x^2

```

(continues on next page)

(continued from previous page)

```
A22(0,1) + A22(2,0)
sage: (x^2).is_simple_object()
False
```

```
>>> from sage.all import *
>>> A22 = FusionRing("A2", Integer(2))
>>> x = A22(Integer(1), Integer(0)); x
A22(1,0)
>>> x.is_simple_object()
True
>>> x**Integer(2)
A22(0,1) + A22(2,0)
>>> (x**Integer(2)).is_simple_object()
False
```

q_dimension (*base_coercion=True*)

Return the quantum dimension as an element of the cyclotomic field of the 2ℓ -th roots of unity, where $\ell = m(k + h^\vee)$ with $m = 1, 2, 3$ depending on whether type is simply, doubly or triply laced, k is the level and $h^\vee$ is the dual Coxeter number.

EXAMPLES:

```
sage: B22 = FusionRing("B2", 2)
sage: [(b.q_dimension())^2 for b in B22.basis()]
[1, 4, 5, 1, 5, 4]
```

```
>>> from sage.all import *
>>> B22 = FusionRing("B2", Integer(2))
>>> [(b.q_dimension())**Integer(2) for b in B22.basis()]
[1, 4, 5, 1, 5, 4]
```

ribbon (*base_coercion=True*)

Return the twist or ribbon element of *self*.

If h is the rational number modulo 2 produced by *self*.twist(), this method produces $e^{i\pi h}$.

See also

An additive version of this is available as *twist()*.

EXAMPLES:

```
sage: F = FusionRing("A1", 3)
sage: [x.twist() for x in F.basis()]
[0, 3/10, 4/5, 3/2]
sage: [x.ribbon(base_coercion=False) for x in F.basis()]
[1, zeta40^6, zeta40^12 - zeta40^8 + zeta40^4 - 1, -zeta40^10]
sage: [F.root_of_unity(x, base_coercion=False) for x in [0, 3/10, 4/5, 3/2]]
[1, zeta40^6, zeta40^12 - zeta40^8 + zeta40^4 - 1, -zeta40^10]
```

```

>>> from sage.all import *
>>> F = FusionRing("A1", Integer(3))
>>> [x.twist() for x in F.basis()]
[0, 3/10, 4/5, 3/2]
>>> [x.ribbon(base_coercion=False) for x in F.basis()]
[1, zeta40^6, zeta40^12 - zeta40^8 + zeta40^4 - 1, -zeta40^10]
>>> [F.root_of_unity(x, base_coercion=False) for x in [Integer(0),
↳ Integer(3)/Integer(10), Integer(4)/Integer(5), Integer(3)/Integer(2)]]
[1, zeta40^6, zeta40^12 - zeta40^8 + zeta40^4 - 1, -zeta40^10]

```

twist (*reduced=True*)

Return a rational number h such that $\theta = e^{i\pi h}$ is the twist of `self`. The quantity $e^{i\pi h}$ is also available using `ribbon()`.

This method is only available for simple objects. If λ is the weight of the object, then $h = \langle \lambda, \lambda + 2\rho \rangle$, where ρ is half the sum of the positive roots. As in [Row2006], this requires normalizing the invariant bilinear form so that $\langle \alpha, \alpha \rangle = 2$ for short roots.

INPUT:

- `reduced` – boolean (default: True); if True then return the twist reduced modulo 2

EXAMPLES:

```

sage: G21 = FusionRing("G2", 1)
sage: [x.twist() for x in G21.basis()]
[0, 4/5]
sage: [G21.root_of_unity(x.twist()) for x in G21.basis()]
[1, zeta60^14 - zeta60^4]
sage: zeta60 = G21.field().gen()
sage: zeta60^((4/5)*(60/2))
zeta60^14 - zeta60^4

sage: F42 = FusionRing("F4", 2)
sage: [x.twist() for x in F42.basis()]
[0, 18/11, 2/11, 12/11, 4/11]

sage: E62 = FusionRing("E6", 2)
sage: [x.twist() for x in E62.basis()]
[0, 26/21, 12/7, 8/21, 8/21, 26/21, 2/3, 4/7, 2/3]

```

```

>>> from sage.all import *
>>> G21 = FusionRing("G2", Integer(1))
>>> [x.twist() for x in G21.basis()]
[0, 4/5]
>>> [G21.root_of_unity(x.twist()) for x in G21.basis()]
[1, zeta60^14 - zeta60^4]
>>> zeta60 = G21.field().gen()
>>> zeta60**((Integer(4)/Integer(5))*(Integer(60)/Integer(2)))
zeta60^14 - zeta60^4

>>> F42 = FusionRing("F4", Integer(2))
>>> [x.twist() for x in F42.basis()]
[0, 18/11, 2/11, 12/11, 4/11]

```

(continues on next page)

(continued from previous page)

```
>>> E62 = FusionRing("E6", Integer(2))
>>> [x.twist() for x in E62.basis()]
[0, 26/21, 12/7, 8/21, 8/21, 26/21, 2/3, 4/7, 2/3]
```

weight()Return the parametrizing dominant weight in the level k alcove.

This method is only available for basis elements.

EXAMPLES:

```
sage: A21 = FusionRing("A2", 1)
sage: [x.weight() for x in A21.basis().list()]
[(0, 0, 0), (2/3, -1/3, -1/3), (1/3, 1/3, -2/3)]
```

```
>>> from sage.all import *
>>> A21 = FusionRing("A2", Integer(1))
>>> [x.weight() for x in A21.basis().list()]
[(0, 0, 0), (2/3, -1/3, -1/3), (1/3, 1/3, -2/3)]
```

N_ijk(elt_i, elt_j, elt_k)Return the symmetric fusion coefficient N_{ijk} .

INPUT:

- `elt_i, elt_j, elt_k` – elements of the fusion basis

This is the same as N_{ij}^{k*} , where N_{ij}^k are the structure coefficients of the ring (see `Nk_ij()`), and k^* denotes the dual element. The coefficient N_{ijk} is unchanged under permutations of the three basis vectors.

EXAMPLES:

```
sage: G23 = FusionRing("G2", 3)
sage: G23.fusion_labels("g")
sage: b = G23.basis().list(); b
[g0, g1, g2, g3, g4, g5]
sage: [(x, y, z) for x in b for y in b for z in b if G23.N_ijk(x, y, z) > 1]
[(g3, g3, g3), (g3, g3, g4), (g3, g4, g3), (g4, g3, g3)]
sage: all(G23.N_ijk(x, y, z)==G23.N_ijk(y, z, x) for x in b for y in b for z in b)
True
sage: all(G23.N_ijk(x, y, z)==G23.N_ijk(y, x, z) for x in b for y in b for z in b)
True
```

```
>>> from sage.all import *
>>> G23 = FusionRing("G2", Integer(3))
>>> G23.fusion_labels("g")
>>> b = G23.basis().list(); b
[g0, g1, g2, g3, g4, g5]
>>> [(x, y, z) for x in b for y in b for z in b if G23.N_ijk(x, y, z) > Integer(1)]
[(g3, g3, g3), (g3, g3, g4), (g3, g4, g3), (g4, g3, g3)]
>>> all(G23.N_ijk(x, y, z)==G23.N_ijk(y, z, x) for x in b for y in b for z in b)
```

(continues on next page)

(continued from previous page)

```

↪b)
True
>>> all(G23.N_ijk(x, y, z)==G23.N_ijk(y, x, z) for x in b for y in b for z in_
↪b)
True

```

Nk_ij(elt_i, elt_j, elt_k)Return the fusion coefficient N_{ij}^k .

These are the structure coefficients of the fusion ring, so

$$i * j = \sum_k N_{ij}^k k.$$

EXAMPLES:

```

sage: A22 = FusionRing("A2", 2)
sage: b = A22.basis().list()
sage: all(x*y == sum(A22.Nk_ij(x, y, k)*k for k in b) for x in b for y in b)
True

```

```

>>> from sage.all import *
>>> A22 = FusionRing("A2", Integer(2))
>>> b = A22.basis().list()
>>> all(x*y == sum(A22.Nk_ij(x, y, k)*k for k in b) for x in b for y in b)
True

```

conj_matrix()

Return the conjugation matrix, which is the permutation matrix for the conjugation (dual) operation on basis elements.

EXAMPLES:

```

sage: FusionRing("A2", 1).conj_matrix()
[1 0 0]
[0 0 1]
[0 1 0]

```

```

>>> from sage.all import *
>>> FusionRing("A2", Integer(1)).conj_matrix()
[1 0 0]
[0 0 1]
[0 1 0]

```

field()Return a cyclotomic field large enough to contain the 2ℓ -th roots of unity, as well as all the S -matrix entries.

EXAMPLES:

```

sage: FusionRing("A2", 2).field()
Cyclotomic Field of order 60 and degree 16
sage: FusionRing("B2", 2).field()
Cyclotomic Field of order 40 and degree 16

```

```
>>> from sage.all import *
>>> FusionRing("A2", Integer(2)).field()
Cyclotomic Field of order 60 and degree 16
>>> FusionRing("B2", Integer(2)).field()
Cyclotomic Field of order 40 and degree 16
```

fusion_l()

Return the product $\ell = m_g(k + h^\vee)$, where m_g denotes the square of the ratio of the lengths of long to short roots of the underlying Lie algebra, k denotes the level of the FusionRing, and $h^\vee$ denotes the dual Coxeter number of the underlying Lie algebra.

This value is used to define the associated root 2ℓ -th of unity $q = e^{i\pi/\ell}$.

EXAMPLES:

```
sage: B22 = FusionRing('B2', 2)
sage: B22.fusion_l()
10
sage: D52 = FusionRing('D5', 2)
sage: D52.fusion_l()
10
```

```
>>> from sage.all import *
>>> B22 = FusionRing('B2', Integer(2))
>>> B22.fusion_l()
10
>>> D52 = FusionRing('D5', Integer(2))
>>> D52.fusion_l()
10
```

fusion_labels (*labels=None, inject_variables=False*)

Set the labels of the basis.

INPUT:

- *labels* – (default: None) a list of strings or string
- *inject_variables* – boolean (default: False); if True, then inject the variable names into the global namespace; note that this could override objects already defined

If *labels* is a list, the length of the list must equal the number of basis elements. These become the names of the basis elements.

If *labels* is a string, this is treated as a prefix and a list of names is generated.

If *labels* is None, then this resets the labels to the default.

EXAMPLES:

```
sage: A13 = FusionRing("A1", 3)
sage: A13.fusion_labels("x")
sage: fb = list(A13.basis()); fb
[x0, x1, x2, x3]
sage: Matrix([[x*y for y in A13.basis()] for x in A13.basis()])
[
  x0      x1      x2      x3
  x1 x0 + x2 x1 + x3      x2]
```

(continues on next page)

(continued from previous page)

```
[      x2 x1 + x3 x0 + x2      x1]
[      x3      x2      x1      x0]
```

```
>>> from sage.all import *
>>> A13 = FusionRing("A1", Integer(3))
>>> A13.fusion_labels("x")
>>> fb = list(A13.basis()); fb
[x0, x1, x2, x3]
>>> Matrix([[x*y for y in A13.basis()] for x in A13.basis()])
[      x0      x1      x2      x3]
[      x1 x0 + x2 x1 + x3      x2]
[      x2 x1 + x3 x0 + x2      x1]
[      x3      x2      x1      x0]
```

We give an example where the variables are injected into the global namespace:

```
sage: A13.fusion_labels("y", inject_variables=True)
sage: y0
y0
sage: y0.parent() is A13
True
```

```
>>> from sage.all import *
>>> A13.fusion_labels("y", inject_variables=True)
>>> y0
y0
>>> y0.parent() is A13
True
```

We reset the labels to the default:

```
sage: A13.fusion_labels()
sage: fb
[A13(0), A13(1), A13(2), A13(3)]
sage: y0
A13(0)
```

```
>>> from sage.all import *
>>> A13.fusion_labels()
>>> fb
[A13(0), A13(1), A13(2), A13(3)]
>>> y0
A13(0)
```

fusion_level()

Return the level k of `self`.

EXAMPLES:

```
sage: B22 = FusionRing('B2', 2)
sage: B22.fusion_level()
2
```

```
>>> from sage.all import *
>>> B22 = FusionRing('B2', Integer(2))
>>> B22.fusion_level()
2
```

fvars_field()

Return a field containing the CyclotomicField computed by *field()* as well as all the F-symbols of the associated FMatrix factory object.

This method is only available if *self* is multiplicity-free.

OUTPUT:

Depending on the CartanType associated to *self* and whether a call to an F-matrix solver has been made, this method will return the same field as *field()*, a *NumberField()*, or the *QQbar*. See *FMatrix.attempt_number_field_computation()* for more details.

Before running an F-matrix solver, the output of this method matches that of *field()*. However, the output may change upon successfully computing F-symbols. Requesting braid generators triggers a call to *FMatrix.find_orthogonal_solution()*, so the output of this method may change after such a computation.

By default, the output of methods like *r_matrix()*, *s_matrix()*, *twists_matrix()*, etc. will lie in the *fvars_field*, unless the *base_coercion* option is set to *False*.

This method does not trigger a solver run.

EXAMPLES:

```
sage: A13 = FusionRing("A1", 3, fusion_labels='a', inject_variables=True)
sage: A13.fvars_field()
Cyclotomic Field of order 40 and degree 16
sage: A13.field()
Cyclotomic Field of order 40 and degree 16
sage: a2**4
2*a0 + 3*a2
sage: comp_basis, sig = A13.get_braid_generators(a2, a2, 3, verbose=False)
↪# long time (<3s)
sage: A13.fvars_field()                                     #_
↪long time
Number Field in a with defining polynomial y^32 - ... - 500*y^2 + 25
sage: a2.q_dimension().parent()                             #_
↪long time
Number Field in a with defining polynomial y^32 - ... - 500*y^2 + 25
sage: A13.field()
Cyclotomic Field of order 40 and degree 16
```

```
>>> from sage.all import *
>>> A13 = FusionRing("A1", Integer(3), fusion_labels='a', inject_
↪variables=True)
>>> A13.fvars_field()
Cyclotomic Field of order 40 and degree 16
>>> A13.field()
Cyclotomic Field of order 40 and degree 16
>>> a2**Integer(4)
2*a0 + 3*a2
>>> comp_basis, sig = A13.get_braid_generators(a2, a2, Integer(3), ↪
```

(continues on next page)

(continued from previous page)

```

↪ verbose=False)      # long time (<3s)
>>> A13.fvars_field()                                     #↪
↪ long time
Number Field in a with defining polynomial y^32 - ... - 500*y^2 + 25
>>> a2.q_dimension().parent()                             #↪
↪ long time
Number Field in a with defining polynomial y^32 - ... - 500*y^2 + 25
>>> A13.field()
Cyclotomic Field of order 40 and degree 16

```

In some cases, the `NumberField.optimized_representation()` may be used to obtain a better defining polynomial for the computed `NumberField()`.

gens_satisfy_braid_gp_rels(sig)

Return True if the matrices in the list `sig` satisfy the braid relations.

This if n is the cardinality of `sig`, this confirms that these matrices define a representation of the Artin braid group on $n + 1$ strands. Tests correctness of `get_braid_generators()`.

EXAMPLES:

```

sage: F41 = FusionRing("F4", 1, fusion_labels='f', inject_variables=True)
sage: f1*f1
f0 + f1
sage: comp, sig = F41.get_braid_generators(f1, f0, 4, verbose=False)
sage: F41.gens_satisfy_braid_gp_rels(sig)
True

```

```

>>> from sage.all import *
>>> F41 = FusionRing("F4", Integer(1), fusion_labels='f', inject_
↪ variables=True)
>>> f1*f1
f0 + f1
>>> comp, sig = F41.get_braid_generators(f1, f0, Integer(4), verbose=False)
>>> F41.gens_satisfy_braid_gp_rels(sig)
True

```

get_braid_generators(fusing_anyon, total_charge_anyon, n_strands, checkpoint=False, save_results="", warm_start="", use_mp=True, verbose=True)

Compute generators of the Artin braid group on `n_strands` strands.

If $a = \text{``fusing}_a\text{nyon``}$ and $b = \text{``total\_charge}_a\text{nyon``}$ the generators are endomorphisms of $\text{Hom}(b, a^n)$.

INPUT:

- `fusing_anyon` – a basis element of `self`
- `total_charge_anyon` – a basis element of `self`
- `n_strands` – positive integer greater than 2
- `checkpoint` – boolean (default: `False`); whether the F-matrix solver should pickle checkpoints
- `save_results` – (optional) a string indicating the name of a file in which to pickle computed F-symbols for later use

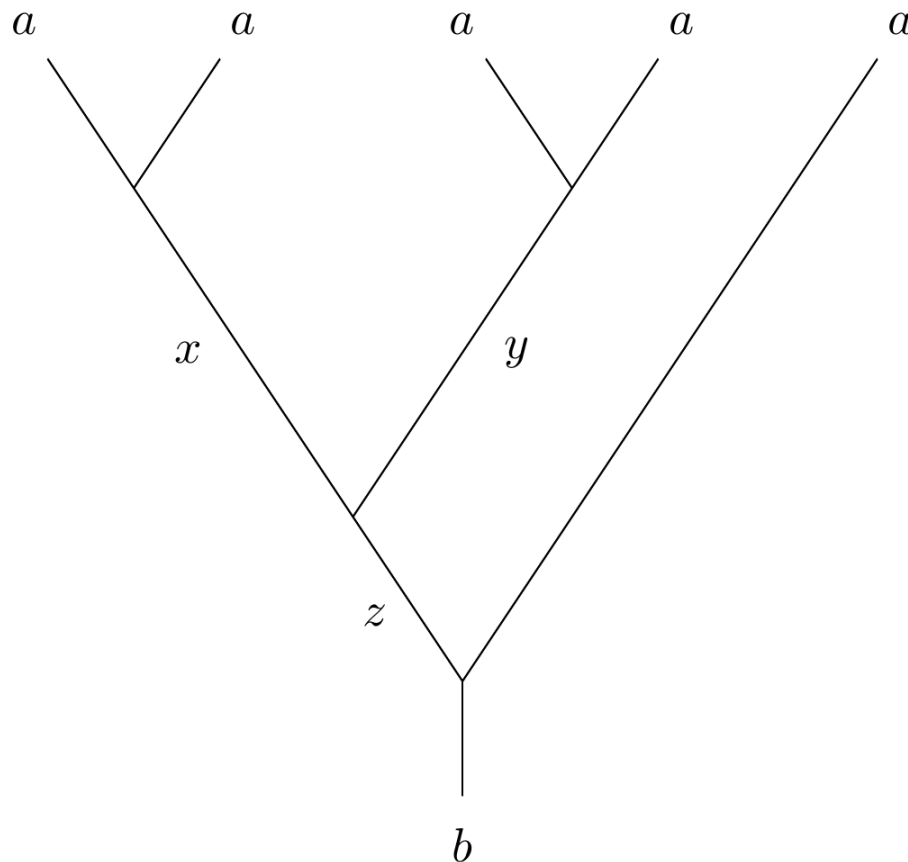
- `warm_start` – (optional) a string indicating the name of a pickled checkpoint file to “warm” start the F-matrix solver. The pickle may be a checkpoint generated by the solver, or a file containing solver results. If all F-symbols are known, we don’t run the solver again.
- `use_mp` – boolean (default: `True`); whether to use multiprocessing to speed up the computation; this is highly recommended.
- `verbose` – boolean (default: `True`); whether to be verbose with the computation

For more information on the optional parameters, see `FMatrix.find_orthogonal_solution()`.

Given a simple object in the fusion category, here called `fusing_anyon` allowing the universal R-matrix to act on adjacent pairs in the fusion of `n_strands` copies of `fusing_anyon` produces an action of the braid group. This representation can be decomposed over another anyon, here called `total_charge_anyon`. See [CHW2015].

OUTPUT:

The method outputs a pair of data `(comp_basis, sig)` where `comp_basis` is a list of basis elements of the braid group module, parametrized by a list of fusion ring elements describing a fusion tree. For example with 5 strands the fusion tree is as follows. See `get_computational_basis()` for more information.



`sig` is a list of braid group generators as matrices. In some cases these will be represented as sparse matrices.

In the following example we compute a 5-dimensional braid group representation on 5 strands associated to the spin representation in the modular tensor category $SU(2)_4 \cong SO(3)_2$.

EXAMPLES:

```

sage: A14 = FusionRing("A1", 4)
sage: A14.get_order()
[(0, 0), (1/2, -1/2), (1, -1), (3/2, -3/2), (2, -2)]
sage: A14.fusion_labels(["one", "two", "three", "four", "five"], inject_
↳variables=True)
sage: [A14(x) for x in A14.get_order()]
[one, two, three, four, five]
sage: two ** 5
5*two + 4*four
sage: comp_basis, sig = A14.get_braid_generators(two, two, 5, verbose=False)
↳# long time
sage: A14.gens_satisfy_braid_gp_rels(sig) #_
↳long time
True
sage: len(comp_basis) == 5 #_
↳long time
True

```

```

>>> from sage.all import *
>>> A14 = FusionRing("A1", Integer(4))
>>> A14.get_order()
[(0, 0), (1/2, -1/2), (1, -1), (3/2, -3/2), (2, -2)]
>>> A14.fusion_labels(["one", "two", "three", "four", "five"], inject_
↳variables=True)
>>> [A14(x) for x in A14.get_order()]
[one, two, three, four, five]
>>> two ** Integer(5)
5*two + 4*four
>>> comp_basis, sig = A14.get_braid_generators(two, two, Integer(5),_
↳verbose=False) # long time
>>> A14.gens_satisfy_braid_gp_rels(sig) #_
↳long time
True
>>> len(comp_basis) == Integer(5) _
↳ # long time
True

```

get_computational_basis($a, b, n\_strands$)

Return the so-called computational basis for $\text{Hom}(b, a^n)$.

INPUT:

- a – a basis element
- b – another basis element
- $n\_strands$ – the number of strands for a braid group

Let $n = n\_strands$ and let k be the greatest integer $\leq n/2$. The braid group acts on $\text{Hom}(b, a^n)$. This action is computed in `get_braid_generators()`. This method returns the computational basis in the form of a list of fusion trees. Each tree is represented by an $(n - 2)$ -tuple

$$(m_1, \dots, m_k, l_1, \dots, l_{k-2})$$

such that each m_j is an irreducible constituent in $a \otimes a$ and

$$\begin{aligned} b &\in l_{k-2} \otimes m_k, \\ l_{k-2} &\in l_{k-3} \otimes m_{k-1}, \\ &\dots, \\ l_2 &\in l_1 \otimes m_3, \\ l_1 &\in m_1 \otimes m_2, \end{aligned}$$

where $z \in x \otimes y$ means $N_{xy}^z \neq 0$.

As a computational device when `n_strands` is odd, we pad the vector $(m_1, \dots, m_k)$ with an additional m_{k+1} equal to a . However, this m_{k+1} does *not* appear in the output of this method.

The following example appears in Section 3.1 of [CW2015].

EXAMPLES:

```
sage: A14 = FusionRing("A1", 4)
sage: A14.get_order()
[(0, 0), (1/2, -1/2), (1, -1), (3/2, -3/2), (2, -2)]
sage: A14.fusion_labels(["zero", "one", "two", "three", "four"], inject_
↪variables=True)
sage: [A14(x) for x in A14.get_order()]
[zero, one, two, three, four]
sage: A14.get_computational_basis(one, two, 4)
[(two, two), (two, zero), (zero, two)]
```

```
>>> from sage.all import *
>>> A14 = FusionRing("A1", Integer(4))
>>> A14.get_order()
[(0, 0), (1/2, -1/2), (1, -1), (3/2, -3/2), (2, -2)]
>>> A14.fusion_labels(["zero", "one", "two", "three", "four"], inject_
↪variables=True)
>>> [A14(x) for x in A14.get_order()]
[zero, one, two, three, four]
>>> A14.get_computational_basis(one, two, Integer(4))
[(two, two), (two, zero), (zero, two)]
```

get_fmatrix(*args, **kwargs)

Construct an `FMatrix` factory to solve the pentagon relations and organize the resulting F-symbols.

EXAMPLES:

```
sage: A15 = FusionRing("A1", 5)
sage: A15.get_fmatrix()
F-Matrix factory for The Fusion Ring of Type A1 and level 5 with Integer Ring_
↪coefficients
```

```
>>> from sage.all import *
>>> A15 = FusionRing("A1", Integer(5))
>>> A15.get_fmatrix()
F-Matrix factory for The Fusion Ring of Type A1 and level 5 with Integer Ring_
↪coefficients
```

get_order()

Return the weights of the basis vectors in a fixed order.

You may change the order of the basis using `CombinatorialFreeModule.set_order()`

EXAMPLES:

```
sage: A15 = FusionRing("A1", 5)
sage: w = A15.get_order(); w
[(0, 0), (1/2, -1/2), (1, -1), (3/2, -3/2), (2, -2), (5/2, -5/2)]
sage: A15.set_order([w[k] for k in [0, 4, 1, 3, 5, 2]])
sage: [A15(x) for x in A15.get_order()]
[A15(0), A15(4), A15(1), A15(3), A15(5), A15(2)]
```

```
>>> from sage.all import *
>>> A15 = FusionRing("A1", Integer(5))
>>> w = A15.get_order(); w
[(0, 0), (1/2, -1/2), (1, -1), (3/2, -3/2), (2, -2), (5/2, -5/2)]
>>> A15.set_order([w[k] for k in [Integer(0), Integer(4), Integer(1),
↳ Integer(3), Integer(5), Integer(2)]])
>>> [A15(x) for x in A15.get_order()]
[A15(0), A15(4), A15(1), A15(3), A15(5), A15(2)]
```

Warning

This duplicates `get_order()` from `CombinatorialFreeModule` except the result is *not* cached. Caching of `CombinatorialFreeModule.get_order()` causes inconsistent results after calling `CombinatorialFreeModule.set_order()`.

`global_q_dimension(base_coercion=True)`

Return $\sum d_i^2$, where the sum is over all simple objects and d_i is the quantum dimension.

The global q -dimension is a positive real number.

EXAMPLES:

```
sage: FusionRing("E6", 1).global_q_dimension()
3
```

```
>>> from sage.all import *
>>> FusionRing("E6", Integer(1)).global_q_dimension()
3
```

`is_multiplicity_free()`

Return True if the fusion multiplicities `Nk_ij()` are bounded by 1.

The `FMatrix` is available only for multiplicity free instances of `FusionRing`.

EXAMPLES:

```
sage: [FusionRing(ct, k).is_multiplicity_free() for ct in ("A1", "A2", "B2",
↳ "C3") for k in (1, 2, 3)]
[True, True, True, True, True, False, True, True, False, True, False, False]
```

```
>>> from sage.all import *
>>> [FusionRing(ct, k).is_multiplicity_free() for ct in ("A1", "A2", "B2", "C3
```

(continues on next page)

(continued from previous page)

```
↪") for k in (Integer(1), Integer(2), Integer(3))
[True, True, True, True, True, False, True, True, False, True, False, False]
```

`r_matrix(i, j, k, base_coercion=True)`

Return the R-matrix entry corresponding to the subobject k in the tensor product of i with j .

Warning

This method only gives complete information when $N_{ij}^k = 1$ (an important special case). Tables of MTC including R-matrices may be found in Section 5.3 of [RoStWa2009] and in [Bond2007].

The R-matrix is a homomorphism $i \otimes j \rightarrow j \otimes i$. This may be hard to describe since the object $i \otimes j$ may be reducible. However if k is a simple subobject of $i \otimes j$ it is also a subobject of $j \otimes i$. If we fix embeddings $k \rightarrow i \otimes j, k \rightarrow j \otimes i$ we may ask for the scalar automorphism of k induced by the R-matrix. This method computes that scalar. It is possible to adjust the set of embeddings $k \rightarrow i \otimes j$ (called a *gauge*) so that this scalar equals

$$\pm \sqrt{\frac{\theta_k}{\theta_i \theta_j}}.$$

If $i \neq j$, the gauge may be used to control the sign of the square root. But if $i = j$ then we must be careful about the sign. These cases are computed by a formula of [BDGRTW2019], Proposition 2.3.

EXAMPLES:

```
sage: I = FusionRing("E8", 2, conjugate=True) # Ising MTC
sage: I.fusion_labels(["i0", "p", "s"], inject_variables=True)
sage: I.r_matrix(s, s, i0) == I.root_of_unity(-1/8)
True
sage: I.r_matrix(p, p, i0)
-1
sage: I.r_matrix(p, s, s) == I.root_of_unity(-1/2)
True
sage: I.r_matrix(s, p, s) == I.root_of_unity(-1/2)
True
sage: I.r_matrix(s, s, p) == I.root_of_unity(3/8)
True
```

```
>>> from sage.all import *
>>> I = FusionRing("E8", Integer(2), conjugate=True) # Ising MTC
>>> I.fusion_labels(["i0", "p", "s"], inject_variables=True)
>>> I.r_matrix(s, s, i0) == I.root_of_unity(-Integer(1)/Integer(8))
True
>>> I.r_matrix(p, p, i0)
-1
>>> I.r_matrix(p, s, s) == I.root_of_unity(-Integer(1)/Integer(2))
True
>>> I.r_matrix(s, p, s) == I.root_of_unity(-Integer(1)/Integer(2))
True
>>> I.r_matrix(s, s, p) == I.root_of_unity(Integer(3)/Integer(8))
True
```

root_of_unity(*r*, *base_coercion*=True)

Return $e^{i\pi r}$ as an element of `self.field()` if possible.

INPUT:

- *r* – a rational number

EXAMPLES:

```
sage: A11 = FusionRing("A1", 1)
sage: A11.field()
Cyclotomic Field of order 24 and degree 8
sage: for n in [1..7]:
....:     try:
....:         print(n, A11.root_of_unity(2/n))
....:     except ValueError as err:
....:         print(n, err)
1 1
2 -1
3 zeta24^4 - 1
4 zeta24^6
5 not a root of unity in the field
6 zeta24^4
7 not a root of unity in the field
```

```
>>> from sage.all import *
>>> A11 = FusionRing("A1", Integer(1))
>>> A11.field()
Cyclotomic Field of order 24 and degree 8
>>> for n in (ellipsis_range(Integer(1), Ellipsis, Integer(7))):
...     try:
...         print(n, A11.root_of_unity(Integer(2)/n))
...     except ValueError as err:
...         print(n, err)
1 1
2 -1
3 zeta24^4 - 1
4 zeta24^6
5 not a root of unity in the field
6 zeta24^4
7 not a root of unity in the field
```

s_ij(*elt_i*, *elt_j*, *base_coercion*=True)

Return the element of the S -matrix of this fusion ring corresponding to the given elements.

This is the unnormalized S -matrix, denoted $\tilde{s}_{ij}$ in [BaKi2001]. To obtain the normalized S -matrix, divide by `global_q_dimension()` or use `S_matrix()` with the option `unitary=True`.

This is computed using the formula

$$s_{i,j} = \frac{1}{\theta_i \theta_j} \sum_k N_{ik}^j d_k \theta_k,$$

where θ_k is the twist and d_k is the quantum dimension. See [Row2006] Equation (2.2) or [EGNO2015] Proposition 8.13.8.

INPUT:

- `elt_i, elt_j` – elements of the fusion basis

EXAMPLES:

```
sage: G21 = FusionRing("G2", 1)
sage: b = G21.basis()
sage: [G21.s_ij(x, y) for x in b for y in b]
[1, -zeta60^14 + zeta60^6 + zeta60^4, -zeta60^14 + zeta60^6 + zeta60^4, -1]
```

```
>>> from sage.all import *
>>> G21 = FusionRing("G2", Integer(1))
>>> b = G21.basis()
>>> [G21.s_ij(x, y) for x in b for y in b]
[1, -zeta60^14 + zeta60^6 + zeta60^4, -zeta60^14 + zeta60^6 + zeta60^4, -1]
```

s_ijconj(*elt_i, elt_j, base_coercion=True*)

Return the conjugate of the element of the S -matrix given by `self.s_ij(elt_i, elt_j, base_coercion=base_coercion)`.

See `s_ij()`.

EXAMPLES:

```
sage: G21 = FusionRing("G2", 1)
sage: b = G21.basis()
sage: [G21.s_ijconj(x, y) for x in b for y in b]
[1, -zeta60^14 + zeta60^6 + zeta60^4, -zeta60^14 + zeta60^6 + zeta60^4, -1]
```

```
>>> from sage.all import *
>>> G21 = FusionRing("G2", Integer(1))
>>> b = G21.basis()
>>> [G21.s_ijconj(x, y) for x in b for y in b]
[1, -zeta60^14 + zeta60^6 + zeta60^4, -zeta60^14 + zeta60^6 + zeta60^4, -1]
```

This method works with all possible types of fields returned by `self.fmts.field()`.

s_matrix(*unitary=False, base_coercion=True*)

Return the S -matrix of this fusion ring.

OPTIONAL:

- `unitary` – boolean (default: `False`); set to `True` to obtain the unitary S -matrix

Without the `unitary` parameter, this is the matrix denoted $\tilde{s}$ in [BaKi2001].

EXAMPLES:

```
sage: D91 = FusionRing("D9", 1)
sage: D91.s_matrix()
[
  1      1      1      1
  1      1      -1     -1
  1     -1 -zeta136^34  zeta136^34
  1     -1  zeta136^34 -zeta136^34
sage: S = D91.s_matrix(unitary=True); S
[
  1/2      1/2      1/2      1/2
  1/2      1/2     -1/2     -1/2
  1/2     -1/2 -1/2*zeta136^34  1/2*zeta136^34]
```

(continues on next page)

(continued from previous page)

```
[
      1/2      -1/2  1/2*zeta136^34 -1/2*zeta136^34]
sage: S*S.conjugate()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

```
>>> from sage.all import *
>>> D91 = FusionRing("D9", Integer(1))
>>> D91.s_matrix()
[
      1      1      1      1]
[
      1      1      -1     -1]
[
      1     -1 -zeta136^34  zeta136^34]
[
      1     -1  zeta136^34 -zeta136^34]
>>> S = D91.s_matrix(unitary=True); S
[
      1/2      1/2      1/2      1/2]
[
      1/2      1/2     -1/2     -1/2]
[
      1/2     -1/2 -1/2*zeta136^34  1/2*zeta136^34]
[
      1/2     -1/2  1/2*zeta136^34 -1/2*zeta136^34]
>>> S*S.conjugate()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

some_elements()

Return some elements of self.

EXAMPLES:

```
sage: D41 = FusionRing('D4', 1)
sage: D41.some_elements()
[D41(1,0,0,0), D41(0,0,1,0), D41(0,0,0,1)]
```

```
>>> from sage.all import *
>>> D41 = FusionRing('D4', Integer(1))
>>> D41.some_elements()
[D41(1,0,0,0), D41(0,0,1,0), D41(0,0,0,1)]
```

test_braid_representation(max_strands=6, anyon=None)

Check that we can compute valid braid group representations.

INPUT:

- max_strands – (default: 6) maximum number of braid group strands
- anyon – (optional) run this test on this particular simple object

Create a braid group representation using `get_braid_generators()` and confirms the braid relations. This test indirectly partially verifies the correctness of the orthogonal F-matrix solver. If the code were incorrect the method would not be deterministic because the fusing anyon is chosen randomly. (A different choice is made for each number of strands tested.) However the doctest is deterministic since it will always return True. If the anyon parameter is omitted, a random anyon is tested for each number of strands up to max_strands.

EXAMPLES:

```
sage: A21 = FusionRing("A2", 1)
sage: A21.test_braid_representation(max_strands=4)
True
sage: F41 = FusionRing("F4", 1)           # long time
sage: F41.test_braid_representation()     # long time
True
```

```
>>> from sage.all import *
>>> A21 = FusionRing("A2", Integer(1))
>>> A21.test_braid_representation(max_strands=Integer(4))
True
>>> F41 = FusionRing("F4", Integer(1))     # long time
>>> F41.test_braid_representation()       # long time
True
```

total_q_order (*base_coercion=True*)

Return the positive square root of `self.global_q_dimension()` as an element of `self.field()`.

This is implemented as $D_+e^{-i\pi c/4}$, where D_+ is `D_plus()` and c is `virasoro_central_charge()`.

EXAMPLES:

```
sage: F = FusionRing("G2", 1)
sage: tqo=F.total_q_order(); tqo
zeta60^15 - zeta60^11 - zeta60^9 + 2*zeta60^3 + zeta60
sage: tqo.is_real_positive()
True
sage: tqo^2 == F.global_q_dimension()
True
```

```
>>> from sage.all import *
>>> F = FusionRing("G2", Integer(1))
>>> tqo=F.total_q_order(); tqo
zeta60^15 - zeta60^11 - zeta60^9 + 2*zeta60^3 + zeta60
>>> tqo.is_real_positive()
True
>>> tqo**Integer(2) == F.global_q_dimension()
True
```

twists_matrix()

Return a diagonal matrix describing the twist corresponding to each simple object in the `FusionRing`.

EXAMPLES:

```
sage: B21=FusionRing("B2", 1)
sage: [x.twist() for x in B21.basis().list()]
[0, 1, 5/8]
sage: [B21.root_of_unity(x.twist()) for x in B21.basis().list()]
[1, -1, zeta32^10]
sage: B21.twists_matrix()
[      1      0      0]
[      0     -1      0]
[      0      0 zeta32^10]
```

```

>>> from sage.all import *
>>> B21=FusionRing("B2", Integer(1))
>>> [x.twist() for x in B21.basis().list()]
[0, 1, 5/8]
>>> [B21.root_of_unity(x.twist()) for x in B21.basis().list()]
[1, -1, zeta32^10]
>>> B21.twists_matrix()
[      1      0      0]
[      0     -1      0]
[      0      0 zeta32^10]

```

`virasoro_central_charge()`

Return the Virasoro central charge of the WZW conformal field theory associated with the Fusion Ring.

If $\mathfrak{g}$ is the corresponding semisimple Lie algebra, this is

$$\frac{k \dim \mathfrak{g}}{k + h^\vee},$$

where k is the level and $h^\vee$ is the dual Coxeter number. See [DFMS1996] Equation (15.61).

Let d_i and θ_i be the quantum dimensions and twists of the simple objects. By Proposition 2.3 in [RoStWa2009], there exists a rational number c such that $D_+/\sqrt{D} = e^{i\pi c/4}$, where $D_+ = \sum d_i^2 \theta_i$ is computed in `D_plus()` and $D = \sum d_i^2 > 0$ is computed by `global_q_dimension()`. Squaring this identity and remembering that $D_+ D_- = D$ gives

$$D_+/D_- = e^{i\pi c/2}.$$

EXAMPLES:

```

sage: R = FusionRing("A1", 2)
sage: c = R.virasoro_central_charge(); c
3/2
sage: Dp = R.D_plus(); Dp
2*zeta32^6
sage: Dm = R.D_minus(); Dm
-2*zeta32^10
sage: Dp / Dm == R.root_of_unity(c/2)
True

```

```

>>> from sage.all import *
>>> R = FusionRing("A1", Integer(2))
>>> c = R.virasoro_central_charge(); c
3/2
>>> Dp = R.D_plus(); Dp
2*zeta32^6
>>> Dm = R.D_minus(); Dm
-2*zeta32^10
>>> Dp / Dm == R.root_of_unity(c/Integer(2))
True

```

5.8.2 The F-Matrix of a Fusion Ring

```
class sage.algebras.fusion_rings.f_matrix.FMatrix(fusion_ring, fusion_label='f', var_prefix='fx',
                                                inject_variables=False)
```

Bases: SageObject

An F-matrix for a *FusionRing*.

INPUT:

- *FR* – a *FusionRing*
- *fusion_label* – (optional) a string used to label basis elements of the *FusionRing* associated to *self* (see *FusionRing.fusion_labels()*)
- *var_prefix* – (optional) a string indicating the desired prefix for variables denoting F-symbols to be solved
- *inject_variables* – boolean (default: *False*); whether to inject variables (*FusionRing* basis element labels and F-symbols) into the global namespace

The *FusionRing* or Verlinde algebra is the Grothendieck ring of a modular tensor category [BaKi2001]. Such categories arise in conformal field theory or in the representation theories of affine Lie algebras, or quantum groups at roots of unity. They have applications to low dimensional topology and knot theory, to conformal field theory and to topological quantum computing. The *FusionRing* captures much information about a fusion category, but to complete the picture, the F-matrices or 6j-symbols are needed. For example these are required in order to construct braid group representations. This can be done using the *FusionRing* method *FusionRing.get_braid_generators()*, which uses the F-matrix.

We only undertake to compute the F-matrix if the *FusionRing* is *multiplicity free* meaning that the Fusion coefficients N_k^{ij} are bounded by 1. For Cartan Types X_r and level k , the multiplicity-free cases are given by the following table.

Cartan Type	k
A_1	any
$A_r, r \geq 2$	≤ 2
$B_r, r \geq 2$	≤ 2
C_2	≤ 2
$C_r, r \geq 3$	≤ 1
$D_r, r \geq 4$	≤ 2
G_2, F_4, E_6, E_7	≤ 2
E_8	≤ 3

Beyond this limitation, computation of the F-matrix can involve very large systems of equations. A rule of thumb is that this code can compute the F-matrix for systems with ≤ 14 simple objects (primary fields) on a machine with 16 GB of memory. (Larger examples can be quite time consuming.)

The *FusionRing* and its methods capture much of the structure of the underlying tensor category. But an important aspect that is not encoded in the fusion ring is the associator, which is a homomorphism $(A \otimes B) \otimes C \rightarrow A \otimes (B \otimes C)$ that requires an additional tool, the F-matrix or 6j-symbol. To specify this, we fix a simple object D and represent the transformation

$$\mathrm{Hom}(D, (A \otimes B) \otimes C) \rightarrow \mathrm{Hom}(D, A \otimes (B \otimes C))$$

by a matrix F_D^{ABC} . This depends on a pair of additional simple objects X and Y . Indeed, we can get a basis for $\mathrm{Hom}(D, (A \otimes B) \otimes C)$ indexed by simple objects X in which the corresponding homomorphism factors through $X \otimes C$, and similarly $\mathrm{Hom}(D, A \otimes (B \otimes C))$ has a basis indexed by Y , in which the basis vector factors through $A \otimes Y$.

See [TTWL2009] for an introduction to this topic, [EGNO2015] Section 4.9 for a precise mathematical definition, and [Bond2007] Section 2.5 and [Ab2022] for discussions of how to compute the F-matrix. In addition to [Bond2007], worked out F-matrices may be found in [RoStWa2009] and [CHW2015].

The F-matrix is only determined up to a *gauge*. This is a family of embeddings $C \rightarrow A \otimes B$ for simple objects A, B, C such that $\text{Hom}(C, A \otimes B)$ is nonzero. Changing the gauge changes the F-matrix though not in a very essential way. By varying the gauge it is possible to make the F-matrices unitary, or it is possible to make them cyclotomic.

Due to the large number of equations we may fail to find a Groebner basis if there are too many variables.

EXAMPLES:

```
sage: I = FusionRing("E8", 2, conjugate=True)
sage: I.fusion_labels(["i0", "p", "s"], inject_variables=True)
sage: f = I.get_fmatrix(inject_variables=True); f
creating variables fx1..fx14
Defining fx0, fx1, fx2, fx3, fx4, fx5, fx6, fx7, fx8, fx9, fx10, fx11, fx12, fx13
F-Matrix factory for The Fusion Ring of Type E8 and level 2 with Integer Ring
↪coefficients
```

```
>>> from sage.all import *
>>> I = FusionRing("E8", Integer(2), conjugate=True)
>>> I.fusion_labels(["i0", "p", "s"], inject_variables=True)
>>> f = I.get_fmatrix(inject_variables=True); f
creating variables fx1..fx14
Defining fx0, fx1, fx2, fx3, fx4, fx5, fx6, fx7, fx8, fx9, fx10, fx11, fx12, fx13
F-Matrix factory for The Fusion Ring of Type E8 and level 2 with Integer Ring
↪coefficients
```

We have injected two sets of variables to the global namespace. We created three variables i_0 , p , s to represent the primary fields (simple elements) of the *FusionRing*. Creating the *FMatrix* factory also created variables fx_1 , fx_2 , ..., fx_{14} in order to solve the hexagon and pentagon equations describing the F-matrix. Since we called *FMatrix* with the parameter `inject_variables=True`, these have been injected into the global namespace. This is not necessary for the code to work but if you want to run the code experimentally you may want access to these variables.

EXAMPLES:

```
sage: f.fmatrix(s, s, s, s)
[fx10 fx11]
[fx12 fx13]
```

```
>>> from sage.all import *
>>> f.fmatrix(s, s, s, s)
[fx10 fx11]
[fx12 fx13]
```

The F-matrix has not been computed at this stage, so the F-matrix F_s^{sss} is filled with variables fx_{10} , fx_{11} , fx_{12} , fx_{13} . The task is to solve for these.

As explained above The F-matrix $(F_D^{ABC})_{X,Y}$ two other variables X and Y . We have methods to tell us (depending on A, B, C, D) what the possibilities for these are. In this example with $A = B = C = D = s$ both X and Y are allowed to be i_0 or s .

```
sage: f.f_from(s, s, s, s), f.f_to(s, s, s, s)
([i0, p], [i0, p])
```

```
>>> from sage.all import *
>>> f.f_from(s, s, s, s), f.f_to(s, s, s, s)
([i0, p], [i0, p])
```

The last two statements show that the possible values of X and Y when $A = B = C = D = s$ are i_0 and p .

The F-matrix is computed by solving the so-called pentagon and hexagon equations. The *pentagon equations* reflect the Mac Lane pentagon axiom in the definition of a monoidal category. The hexagon relations reflect the axioms of a *braided monoidal category*, which are constraints on both the F-matrix and on the R-matrix. Optionally, orthogonality constraints may be imposed to obtain an orthogonal F-matrix.

```
sage: sorted(f.get_defining_equations("pentagons"))[1:3]
[fx9*fx12 - fx2*fx13, fx4*fx11 - fx2*fx13]
sage: sorted(f.get_defining_equations("hexagons"))[1:3]
[fx6 - 1, fx2 + 1]
sage: sorted(f.get_orthogonality_constraints())[1:3]
[fx10*fx11 + fx12*fx13, fx10*fx11 + fx12*fx13]
```

```
>>> from sage.all import *
>>> sorted(f.get_defining_equations("pentagons"))[Integer(1):Integer(3)]
[fx9*fx12 - fx2*fx13, fx4*fx11 - fx2*fx13]
>>> sorted(f.get_defining_equations("hexagons"))[Integer(1):Integer(3)]
[fx6 - 1, fx2 + 1]
>>> sorted(f.get_orthogonality_constraints())[Integer(1):Integer(3)]
[fx10*fx11 + fx12*fx13, fx10*fx11 + fx12*fx13]
```

There are two methods available to compute an F-matrix. The first, `find_cyclotomic_solution()` uses only the pentagon and hexagon relations. The second, `find_orthogonal_solution()` uses additionally the orthogonality relations. There are some differences that should be kept in mind.

`find_cyclotomic_solution()` currently works only with smaller examples. For example the *FusionRing* for G_2 at level 2 is too large. When it is available, this method produces an F-matrix whose entries are in the same cyclotomic field as the underlying *FusionRing*.

```
sage: f.find_cyclotomic_solution()
Setting up hexagons and pentagons...
Finding a Groebner basis...
Solving...
Fixing the gauge...
adding equation... fx1 - 1
adding equation... fx11 - 1
Done!
```

```
>>> from sage.all import *
>>> f.find_cyclotomic_solution()
Setting up hexagons and pentagons...
Finding a Groebner basis...
Solving...
Fixing the gauge...
adding equation... fx1 - 1
```

(continues on next page)

(continued from previous page)

```
adding equation... fx11 - 1
Done!
```

We now have access to the values of the F-matrix using the methods `fmatrix()` and `fmat()`:

```
sage: f.fmatrix(s, s, s, s)
[(-1/2*zeta128^48 + 1/2*zeta128^16) 1]
[ 1/2 (1/2*zeta128^48 - 1/2*zeta128^16)]
sage: f.fmat(s, s, s, s, p, p)
(1/2*zeta128^48 - 1/2*zeta128^16)
```

```
>>> from sage.all import *
>>> f.fmatrix(s, s, s, s)
[(-1/2*zeta128^48 + 1/2*zeta128^16) 1]
[ 1/2 (1/2*zeta128^48 - 1/2*zeta128^16)]
>>> f.fmat(s, s, s, s, p, p)
(1/2*zeta128^48 - 1/2*zeta128^16)
```

`find_orthogonal_solution()` is much more powerful and is capable of handling large cases, sometimes quickly but sometimes (in larger cases) after hours of computation. Its F-matrices are not always in the cyclotomic field that is the base ring of the underlying *FusionRing*, but sometimes in an extension field adjoining some square roots. When this happens, the *FusionRing* is modified, adding an attribute `_basecoer` that is a coercion from the cyclotomic field to the field containing the F-matrix. The field containing the F-matrix is available through `field()`.

```
sage: f = FusionRing("B3", 2).get_fmatrix()
sage: f.find_orthogonal_solution(verbose=False, checkpoint=True) # not tested
↳ (~100 s)
sage: all(v in CyclotomicField(56) for v in f.get_fvars().values()) # not tested
True

sage: f = FusionRing("G2", 2).get_fmatrix()
sage: f.find_orthogonal_solution(verbose=False) # long time (~11 s)
sage: f.field() # long time
Algebraic Field
```

```
>>> from sage.all import *
>>> f = FusionRing("B3", Integer(2)).get_fmatrix()
>>> f.find_orthogonal_solution(verbose=False, checkpoint=True) # not tested (~
↳ 100 s)
>>> all(v in CyclotomicField(Integer(56)) for v in f.get_fvars().values()) # not
↳ tested
True

>>> f = FusionRing("G2", Integer(2)).get_fmatrix()
>>> f.find_orthogonal_solution(verbose=False) # long time (~11 s)
>>> f.field() # long time
Algebraic Field
```

FR()

Return the *FusionRing* associated to self.

EXAMPLES:

```
sage: f = FusionRing("D3", 1).get_fmatrix()
sage: f.FR()
The Fusion Ring of Type D3 and level 1 with Integer Ring coefficients
```

```
>>> from sage.all import *
>>> f = FusionRing("D3", Integer(1)).get_fmatrix()
>>> f.FR()
The Fusion Ring of Type D3 and level 1 with Integer Ring coefficients
```

`attempt_number_field_computation()`

Based on the `CartanType` of `self` and data known on March 17, 2021, determine whether to attempt to find a `NumberField()` containing all the F-symbols.

This method is used by `find_orthogonal_solution()` to determine a field containing all F-symbols. See `field()` and `get_non_cyclotomic_roots()`.

For certain *fusion rings*, the number field computation does not terminate in reasonable time. In these cases, we report F-symbols as elements of the `QQbar`.

EXAMPLES:

```
sage: f = FusionRing("F4", 2).get_fmatrix()
sage: f.attempt_number_field_computation()
False
sage: f = FusionRing("G2", 1).get_fmatrix()
sage: f.attempt_number_field_computation()
True
```

```
>>> from sage.all import *
>>> f = FusionRing("F4", Integer(2)).get_fmatrix()
>>> f.attempt_number_field_computation()
False
>>> f = FusionRing("G2", Integer(1)).get_fmatrix()
>>> f.attempt_number_field_computation()
True
```

Note

In certain cases, F-symbols are found in the associated *FusionRing*'s cyclotomic field and a `NumberField()` computation is not needed. In these cases this method returns `True` but the `find_orthogonal_solution()` solver does *not* undertake a `NumberField()` computation.

`certify_pentagons` (*use_mp=True, verbose=False*)

Obtain a certificate of satisfaction for the pentagon equations, up to floating-point error.

This method converts the computed F-symbols (available through `get_fvars()`) to native Python floats and then checks whether the pentagon equations are satisfied using floating point arithmetic.

When `self.FR().basis()` has many elements, verifying satisfaction of the pentagon relations exactly using `get_defining_equations()` with `option="pentagons"` may take a long time. This method is faster, but it cannot provide mathematical guarantees.

EXAMPLES:

```

sage: f = FusionRing("C3", 1).get_fmatrix()
sage: f.find_orthogonal_solution()          # long time
Computing F-symbols for The Fusion Ring of Type C3 and level 1 with Integer_
↳Ring coefficients with 71 variables...
Set up 134 hex and orthogonality constraints...
Partitioned 134 equations into 17 components of size:
[12, 12, 6, 6, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 1, 1, 1]
Elimination epoch completed... 10 eqns remain in ideal basis
Elimination epoch completed... 0 eqns remain in ideal basis
Hex elim step solved for 51 / 71 variables
Set up 121 reduced pentagons...
Elimination epoch completed... 18 eqns remain in ideal basis
Elimination epoch completed... 5 eqns remain in ideal basis
Pent elim step solved for 64 / 71 variables
Partitioned 5 equations into 1 components of size:
[4]
Elimination epoch completed... 0 eqns remain in ideal basis
Partitioned 6 equations into 6 components of size:
[1, 1, 1, 1, 1, 1]
Computing appropriate NumberField...
sage: f.certify_pentagons() is None          # not tested (cypari issue in_
↳doctest framework), long time (~1.5s)
True

```

```

>>> from sage.all import *
>>> f = FusionRing("C3", Integer(1)).get_fmatrix()
>>> f.find_orthogonal_solution()          # long time
Computing F-symbols for The Fusion Ring of Type C3 and level 1 with Integer_
↳Ring coefficients with 71 variables...
Set up 134 hex and orthogonality constraints...
Partitioned 134 equations into 17 components of size:
[12, 12, 6, 6, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 1, 1, 1]
Elimination epoch completed... 10 eqns remain in ideal basis
Elimination epoch completed... 0 eqns remain in ideal basis
Hex elim step solved for 51 / 71 variables
Set up 121 reduced pentagons...
Elimination epoch completed... 18 eqns remain in ideal basis
Elimination epoch completed... 5 eqns remain in ideal basis
Pent elim step solved for 64 / 71 variables
Partitioned 5 equations into 1 components of size:
[4]
Elimination epoch completed... 0 eqns remain in ideal basis
Partitioned 6 equations into 6 components of size:
[1, 1, 1, 1, 1, 1]
Computing appropriate NumberField...
>>> f.certify_pentagons() is None          # not tested (cypari issue in_
↳doctest framework), long time (~1.5s)
True

```

clear_equations()

Clear the list of equations to be solved.

EXAMPLES:


```

sage: f = FusionRing("E6", 1).get_fmatrix()
sage: f.get_defining_equations('hexagons', output=False)
sage: len(f.ideal_basis)
6
sage: f.clear_equations()
sage: len(f.ideal_basis) == 0
True

```

```

>>> from sage.all import *
>>> f = FusionRing("E6", Integer(1)).get_fmatrix()
>>> f.get_defining_equations('hexagons', output=False)
>>> len(f.ideal_basis)
6
>>> f.clear_equations()
>>> len(f.ideal_basis) == Integer(0)
True

```

clear_vars()

Reset the F-symbols.

EXAMPLES:

```

sage: f = FusionRing("C4", 1).get_fmatrix()
sage: fvars = f.get_fvars()
sage: some_key = sorted(fvars)[0]
sage: fvars[some_key]
fx0
sage: fvars[some_key] = 1
sage: f.get_fvars()[some_key]
1
sage: f.clear_vars()
sage: f.get_fvars()[some_key]
fx0

```

```

>>> from sage.all import *
>>> f = FusionRing("C4", Integer(1)).get_fmatrix()
>>> fvars = f.get_fvars()
>>> some_key = sorted(fvars)[Integer(0)]
>>> fvars[some_key]
fx0
>>> fvars[some_key] = Integer(1)
>>> f.get_fvars()[some_key]
1
>>> f.clear_vars()
>>> f.get_fvars()[some_key]
fx0

```

equations_graph(eqns=None)

Construct a graph corresponding to the given equations.

Every node corresponds to a variable and nodes are connected when the corresponding variables appear together in an equation.

INPUT:

- eqns – list of polynomials

Each polynomial is either an object in the ring returned by `get_poly_ring()` or it is a tuple of pairs representing a polynomial using the internal representation.

If no list of equations is passed, the graph is built from the polynomials in `self.ideal_basis`. In this case the method assumes the internal representation of a polynomial as a tuple of pairs is used.

This method is crucial to `find_orthogonal_solution()`. The hexagon equations, obtained using `get_defining_equations()`, define a disconnected graph that breaks up into many small components. The `find_orthogonal_solution()` solver exploits this when undertaking a Groebner basis computation.

OUTPUT:

A `Graph` object. If a list of polynomial objects was given, the set of nodes in the output graph is the subset polynomial ring generators appearing in the equations.

If the internal representation was used, the set of nodes is the subset of indices corresponding to polynomial ring generators. This option is meant for internal use by the F-matrix solver.

EXAMPLES:

```
sage: f = FusionRing("A3", 1).get_fmatrix()
sage: f.get_poly_ring().ngens()
27
sage: he = f.get_defining_equations('hexagons')
sage: graph = f.equations_graph(he)
sage: graph.connected_components_sizes()
[6, 3, 3, 3, 3, 3, 1, 1, 1]
```

```
>>> from sage.all import *
>>> f = FusionRing("A3", Integer(1)).get_fmatrix()
>>> f.get_poly_ring().ngens()
27
>>> he = f.get_defining_equations('hexagons')
>>> graph = f.equations_graph(he)
>>> graph.connected_components_sizes()
[6, 3, 3, 3, 3, 3, 1, 1, 1]
```

f_from(*a*, *b*, *c*, *d*)

Return the possible x such that there are morphisms $d \rightarrow x \otimes c \rightarrow (a \otimes b) \otimes c$.

INPUT:

- *a*, *b*, *c*, *d* – basis elements of the associated `FusionRing`

EXAMPLES:

```
sage: fr = FusionRing("A1", 3, fusion_labels='a', inject_variables=True)
sage: f = fr.get_fmatrix()
sage: f.fmatrix(a1, a1, a2, a2)
[fx6 fx7]
[fx8 fx9]
sage: f.f_from(a1, a1, a2, a2)
[a0, a2]
sage: f.f_to(a1, a1, a2, a2)
[a1, a3]
```

```

>>> from sage.all import *
>>> fr = FusionRing("A1", Integer(3), fusion_labels='a', inject_
↳ variables=True)
>>> f = fr.get_fmatrix()
>>> f.fmatrix(a1, a1, a2, a2)
[fx6 fx7]
[fx8 fx9]
>>> f.f_from(a1, a1, a2, a2)
[a0, a2]
>>> f.f_to(a1, a1, a2, a2)
[a1, a3]

```

f_to(*a*, *b*, *c*, *d*)

Return the possible *y* such that there are morphisms $d \rightarrow a \otimes y \rightarrow a \otimes (b \otimes c)$.

INPUT:

- *a*, *b*, *c*, *d* – basis elements of the associated *FusionRing*

EXAMPLES:

```

sage: b22 = FusionRing("B2", 2)
sage: b22.fusion_labels("b", inject_variables=True)
sage: B = b22.get_fmatrix()
sage: B.fmatrix(b2, b4, b2, b4)
[fx266 fx267 fx268]
[fx269 fx270 fx271]
[fx272 fx273 fx274]
sage: B.f_from(b2, b4, b2, b4)
[b1, b3, b5]
sage: B.f_to(b2, b4, b2, b4)
[b1, b3, b5]

```

```

>>> from sage.all import *
>>> b22 = FusionRing("B2", Integer(2))
>>> b22.fusion_labels("b", inject_variables=True)
>>> B = b22.get_fmatrix()
>>> B.fmatrix(b2, b4, b2, b4)
[fx266 fx267 fx268]
[fx269 fx270 fx271]
[fx272 fx273 fx274]
>>> B.f_from(b2, b4, b2, b4)
[b1, b3, b5]
>>> B.f_to(b2, b4, b2, b4)
[b1, b3, b5]

```

field()

Return the base field containing the F-symbols.

When *self* is initialized, the field is set to be the cyclotomic field of the *FusionRing* associated to *self*.

The field may change after running *find_orthogonal_solution*(). At that point, this method could return the associated *FusionRing*'s cyclotomic field, an appropriate *NumberField*() that was computed on the fly by the F-matrix solver, or the *QQbar*.

Depending on the `CartanType` of `self`, the solver may need to compute an extension field containing certain square roots that do not belong to the associated `FusionRing`'s cyclotomic field.

In certain cases we revert to `QQbar` because the extension field computation does not seem to terminate. See `attempt_number_field_computation()` for more details.

The method `get_non_cyclotomic_roots()` returns a list of roots defining the extension of the `FusionRing`'s cyclotomic field needed to contain all F-symbols.

EXAMPLES:

```
sage: f = FusionRing("G2", 1).get_fmatrix()
sage: f.field()
Cyclotomic Field of order 60 and degree 16
sage: f.find_orthogonal_solution(verbose=False)
sage: f.field()
Number Field in a with defining polynomial y^32 - ... - 22*y^2 + 1
sage: phi = f.get_qqbar_embedding()
sage: [phi(r).n() for r in f.get_non_cyclotomic_roots()]
[-0.786151377757423 - 8.92806368517581e-31*I]
```

```
>>> from sage.all import *
>>> f = FusionRing("G2", Integer(1)).get_fmatrix()
>>> f.field()
Cyclotomic Field of order 60 and degree 16
>>> f.find_orthogonal_solution(verbose=False)
>>> f.field()
Number Field in a with defining polynomial y^32 - ... - 22*y^2 + 1
>>> phi = f.get_qqbar_embedding()
>>> [phi(r).n() for r in f.get_non_cyclotomic_roots()]
[-0.786151377757423 - 8.92806368517581e-31*I]
```

Note

Consider using `self.field().optimized_representation()` to obtain an equivalent `NumberField()` with a defining polynomial with smaller coefficients, for a more efficient element representation.

find_cyclotomic_solution(*equations=None, algorithm="", verbose=True, output=False*)

Solve the hexagon and pentagon relations to evaluate the F-matrix.

This method (omitting the orthogonality constraints) produces output in the cyclotomic field, but it is very limited in the size of examples it can handle: for example, G_2 at level 2 is too large for this method. You may use `find_orthogonal_solution()` to solve much larger examples.

INPUT:

- `equations` – (optional) a set of equations to be solved; defaults to the hexagon and pentagon equations
- `algorithm` – (optional) algorithm to compute Groebner Basis
- `output` – boolean (default: `False`); output a dictionary of F-matrix values; this may be useful to see but may be omitted since this information will be available afterwards via the `fmatrix()` and `fmatrix()` methods.

EXAMPLES:

```

sage: fr = FusionRing("A2", 1, fusion_labels='a', inject_variables=True)
sage: f = fr.get_fmatrix(inject_variables=True)
creating variables fx1..fx8
Defining fx0, fx1, fx2, fx3, fx4, fx5, fx6, fx7
sage: f.find_cyclotomic_solution(output=True)
Setting up hexagons and pentagons...
Finding a Groebner basis...
Solving...
Fixing the gauge...
adding equation... fx4 - 1
Done!
{(a2, a2, a2, a0, a1, a1): 1,
 (a2, a2, a1, a2, a1, a0): 1,
 (a2, a1, a2, a2, a0, a0): 1,
 (a2, a1, a1, a1, a0, a2): 1,
 (a1, a2, a2, a2, a0, a1): 1,
 (a1, a2, a1, a1, a0, a0): 1,
 (a1, a1, a2, a1, a2, a0): 1,
 (a1, a1, a1, a0, a2, a2): 1}

```

```

>>> from sage.all import *
>>> fr = FusionRing("A2", Integer(1), fusion_labels='a', inject_
↳ variables=True)
>>> f = fr.get_fmatrix(inject_variables=True)
creating variables fx1..fx8
Defining fx0, fx1, fx2, fx3, fx4, fx5, fx6, fx7
>>> f.find_cyclotomic_solution(output=True)
Setting up hexagons and pentagons...
Finding a Groebner basis...
Solving...
Fixing the gauge...
adding equation... fx4 - 1
Done!
{(a2, a2, a2, a0, a1, a1): 1,
 (a2, a2, a1, a2, a1, a0): 1,
 (a2, a1, a2, a2, a0, a0): 1,
 (a2, a1, a1, a1, a0, a2): 1,
 (a1, a2, a2, a2, a0, a1): 1,
 (a1, a2, a1, a1, a0, a0): 1,
 (a1, a1, a2, a1, a2, a0): 1,
 (a1, a1, a1, a0, a2, a2): 1}

```

After you successfully run `find_cyclotomic_solution()` you may check the correctness of the F-matrix by running `get_defining_equations()` with `option='hexagons'` and `option='pentagons'`. These should return empty lists of equations.

EXAMPLES:

```

sage: f.get_defining_equations("hexagons")
[]
sage: f.get_defining_equations("pentagons")
[]

```

```
>>> from sage.all import *
>>> f.get_defining_equations("hexagons")
[]
>>> f.get_defining_equations("pentagons")
[]
```

find_orthogonal_solution(*checkpoint=False, save_results="", warm_start="", use_mp=True, verbose=True*)

Solve the hexagon and pentagon relations, along with orthogonality constraints, to evaluate an orthogonal F-matrix.

INPUT:

- *checkpoint* – boolean (default: `False`); whether the computation should be checkpointed. Depending on the associated `CartanType`, the computation may take hours to complete. For large examples, checkpoints are recommended. This method supports “warm” starting, so the calculation may be resumed from a checkpoint, using the *warm_start* option.

Checkpoints store necessary state in the pickle file `"fmatrix_solver_checkpoint_" + key + ".pickle"`, where *key* is the result of `get_fr_str()`.

Checkpoint pickles are automatically deleted when the solver exits a successful run.

- *save_results* – (optional) a string indicating the name of a pickle file in which to store calculated F-symbols for later use.

If *save_results* is not provided (default), F-matrix results are not stored to file.

The F-symbols may be saved to file after running the solver using `save_fvars()`.

- *warm_start* – (optional) a string indicating the name of a pickle file containing checkpointed solver state. This file must have been produced by a previous call to the solver using the *checkpoint* option.

If no file name is provided, the calculation begins from scratch.

- *use_mp* – boolean (default: `True`); whether to use multiprocessing to speed up calculation. The default value `True` is highly recommended, since parallel processing yields results much more quickly.
- *verbose* – boolean (default: `True`); whether the solver should print out intermediate progress reports

OUTPUT:

This method returns `None`. If the solver runs successfully, the results may be accessed through various methods, such as `get_fvars()`, `fmatrix()`, `fmat()`, etc.

EXAMPLES:

```
sage: f = FusionRing("B5", 1).get_fmatrix(fusion_label='b', inject_
↳variables=True)
creating variables fx1..fx14
Defining fx0, fx1, fx2, fx3, fx4, fx5, fx6, fx7, fx8, fx9, fx10, fx11, fx12,
↳fx13
sage: f.find_orthogonal_solution()
Computing F-symbols for The Fusion Ring of Type B5 and level 1 with Integer
↳Ring coefficients with 14 variables...
Set up 25 hex and orthogonality constraints...
Partitioned 25 equations into 5 components of size:
[4, 3, 3, 3, 1]
Elimination epoch completed... 0 eqns remain in ideal basis
Hex elim step solved for 10 / 14 variables
Set up 7 reduced pentagons...
```

(continues on next page)

(continued from previous page)

```

Elimination epoch completed... 0 eqns remain in ideal basis
Pent elim step solved for 12 / 14 variables
Partitioned 0 equations into 0 components of size:
[]
Partitioned 2 equations into 2 components of size:
[1, 1]
sage: f.fmatrix(b2, b2, b2, b2)
[ 1/2*zeta80^30 - 1/2*zeta80^10 -1/2*zeta80^30 + 1/2*zeta80^10]
[ 1/2*zeta80^30 - 1/2*zeta80^10  1/2*zeta80^30 - 1/2*zeta80^10]
sage: f.fmat(b2, b2, b2, b2, b0, b1)
-1/2*zeta80^30 + 1/2*zeta80^10

```

```

>>> from sage.all import *
>>> f = FusionRing("B5", Integer(1)).get_fmatrix(fusion_label='b', inject_
↳variables=True)
creating variables fx1..fx14
Defining fx0, fx1, fx2, fx3, fx4, fx5, fx6, fx7, fx8, fx9, fx10, fx11, fx12,
↳fx13
>>> f.find_orthogonal_solution()
Computing F-symbols for The Fusion Ring of Type B5 and level 1 with Integer
↳Ring coefficients with 14 variables...
Set up 25 hex and orthogonality constraints...
Partitioned 25 equations into 5 components of size:
[4, 3, 3, 3, 1]
Elimination epoch completed... 0 eqns remain in ideal basis
Hex elim step solved for 10 / 14 variables
Set up 7 reduced pentagons...
Elimination epoch completed... 0 eqns remain in ideal basis
Pent elim step solved for 12 / 14 variables
Partitioned 0 equations into 0 components of size:
[]
Partitioned 2 equations into 2 components of size:
[1, 1]
>>> f.fmatrix(b2, b2, b2, b2)
[ 1/2*zeta80^30 - 1/2*zeta80^10 -1/2*zeta80^30 + 1/2*zeta80^10]
[ 1/2*zeta80^30 - 1/2*zeta80^10  1/2*zeta80^30 - 1/2*zeta80^10]
>>> f.fmat(b2, b2, b2, b2, b0, b1)
-1/2*zeta80^30 + 1/2*zeta80^10

```

Every F-matrix $F_d^{a,b,c}$ is orthogonal and in many cases real. We may use `fmats_are_orthogonal()` and `fvars_are_real()` to obtain correctness certificates.

EXAMPLES:

```

sage: f.fstats_are_orthogonal()
True

```

```

>>> from sage.all import *
>>> f.fstats_are_orthogonal()
True

```

In any case, the F-symbols are obtained as elements of the associated `FusionRing`'s `Cyclotomic field`, a computed `NumberField()`, or `QQbar`. Currently, the field containing the F-symbols is determined based

on the `CartanType` associated to `self`.

See also

`attempt_number_field_computation()`

findcases (*output=False*)

Return unknown F-matrix entries.

If run with `output=True`, this returns two dictionaries; otherwise it just returns the number of unknown values.

EXAMPLES:

```
sage: f = FusionRing("G2", 1, fusion_labels=("i0", "t")).get_fmatrix()
sage: f.findcases()
5
sage: f.findcases(output=True)
({0: (t, t, t, i0, t, t),
 1: (t, t, t, t, i0, i0),
 2: (t, t, t, t, i0, t),
 3: (t, t, t, t, t, i0),
 4: (t, t, t, t, t, t)},
{(t, t, t, i0, t, t): fx0,
 (t, t, t, t, i0, i0): fx1,
 (t, t, t, t, i0, t): fx2,
 (t, t, t, t, t, i0): fx3,
 (t, t, t, t, t, t): fx4})
```

```
>>> from sage.all import *
>>> f = FusionRing("G2", Integer(1), fusion_labels=("i0", "t")).get_fmatrix()
>>> f.findcases()
5
>>> f.findcases(output=True)
({0: (t, t, t, i0, t, t),
 1: (t, t, t, t, i0, i0),
 2: (t, t, t, t, i0, t),
 3: (t, t, t, t, t, i0),
 4: (t, t, t, t, t, t)},
{(t, t, t, i0, t, t): fx0,
 (t, t, t, t, i0, i0): fx1,
 (t, t, t, t, i0, t): fx2,
 (t, t, t, t, t, i0): fx3,
 (t, t, t, t, t, t): fx4})
```

fmat (*a, b, c, d, x, y, data=True*)

Return the F-Matrix coefficient $(F_d^{a,b,c})_{x,y}$.

EXAMPLES:

```
sage: fr = FusionRing("G2", 1, fusion_labels=("i0", "t"), inject_
↪variables=True)
sage: f = fr.get_fmatrix()
```

(continues on next page)

(continued from previous page)

```

sage: [f.fmat(t, t, t, t, x, y) for x in fr.basis() for y in fr.basis()]
[fx1, fx2, fx3, fx4]
sage: f.find_cyclotomic_solution(output=True)
Setting up hexagons and pentagons...
Finding a Groebner basis...
Solving...
Fixing the gauge...
adding equation... fx2 - 1
Done!
{(t, t, t, i0, t, t): 1,
 (t, t, t, t, i0, i0): (-zeta60^14 + zeta60^6 + zeta60^4 - 1),
 (t, t, t, t, i0, t): 1,
 (t, t, t, t, t, i0): (-zeta60^14 + zeta60^6 + zeta60^4 - 1),
 (t, t, t, t, t, t): (zeta60^14 - zeta60^6 - zeta60^4 + 1)}
sage: [f.fmat(t, t, t, t, x, y) for x in f._FR.basis() for y in f._FR.basis()]
[(-zeta60^14 + zeta60^6 + zeta60^4 - 1),
 1,
 (-zeta60^14 + zeta60^6 + zeta60^4 - 1),
 (zeta60^14 - zeta60^6 - zeta60^4 + 1)]

```

```

>>> from sage.all import *
>>> fr = FusionRing("G2", Integer(1), fusion_labels=("i0", "t"), inject_
↳variables=True)
>>> f = fr.get_fmatrix()
>>> [f.fmat(t, t, t, t, x, y) for x in fr.basis() for y in fr.basis()]
[fx1, fx2, fx3, fx4]
>>> f.find_cyclotomic_solution(output=True)
Setting up hexagons and pentagons...
Finding a Groebner basis...
Solving...
Fixing the gauge...
adding equation... fx2 - 1
Done!
{(t, t, t, i0, t, t): 1,
 (t, t, t, t, i0, i0): (-zeta60^14 + zeta60^6 + zeta60^4 - 1),
 (t, t, t, t, i0, t): 1,
 (t, t, t, t, t, i0): (-zeta60^14 + zeta60^6 + zeta60^4 - 1),
 (t, t, t, t, t, t): (zeta60^14 - zeta60^6 - zeta60^4 + 1)}
>>> [f.fmat(t, t, t, t, x, y) for x in f._FR.basis() for y in f._FR.basis()]
[(-zeta60^14 + zeta60^6 + zeta60^4 - 1),
 1,
 (-zeta60^14 + zeta60^6 + zeta60^4 - 1),
 (zeta60^14 - zeta60^6 - zeta60^4 + 1)]

```

fmatrix(a, b, c, d)

Return the F-Matrix $F_d^{a,b,c}$.

INPUT:

- a, b, c, d – basis elements of the associated *FusionRing*

EXAMPLES:

```

sage: fr = FusionRing("A1", 2, fusion_labels='c', inject_variables=True)
sage: f = fr.get_fmatrix(new=True)
sage: f.fmatrix(c1, c1, c1, c1)
[fx0 fx1]
[fx2 fx3]
sage: f.find_cyclotomic_solution(verbose=False);
adding equation... fx4 - 1
adding equation... fx10 - 1
sage: f.f_from(c1, c1, c1, c1)
[c0, c2]
sage: f.f_to(c1, c1, c1, c1)
[c0, c2]
sage: f.fmatrix(c1, c1, c1, c1)
[ (1/2*zeta32^12 - 1/2*zeta32^4) (-1/2*zeta32^12 + 1/2*zeta32^4)]
[ (1/2*zeta32^12 - 1/2*zeta32^4) (1/2*zeta32^12 - 1/2*zeta32^4)]

```

```

>>> from sage.all import *
>>> fr = FusionRing("A1", Integer(2), fusion_labels='c', inject_
↳ variables=True)
>>> f = fr.get_fmatrix(new=True)
>>> f.fmatrix(c1, c1, c1, c1)
[fx0 fx1]
[fx2 fx3]
>>> f.find_cyclotomic_solution(verbose=False);
adding equation... fx4 - 1
adding equation... fx10 - 1
>>> f.f_from(c1, c1, c1, c1)
[c0, c2]
>>> f.f_to(c1, c1, c1, c1)
[c0, c2]
>>> f.fmatrix(c1, c1, c1, c1)
[ (1/2*zeta32^12 - 1/2*zeta32^4) (-1/2*zeta32^12 + 1/2*zeta32^4)]
[ (1/2*zeta32^12 - 1/2*zeta32^4) (1/2*zeta32^12 - 1/2*zeta32^4)]

```

fmats_are_orthogonal()

Verify that all F-matrices are orthogonal.

This method should always return True when called after running *find_orthogonal_solution()*.

EXAMPLES:

```

sage: f = FusionRing("D4", 1).get_fmatrix()
sage: f.find_orthogonal_solution(verbose=False)
sage: f.fmats_are_orthogonal()
True

```

```

>>> from sage.all import *
>>> f = FusionRing("D4", Integer(1)).get_fmatrix()
>>> f.find_orthogonal_solution(verbose=False)
>>> f.fmats_are_orthogonal()
True

```

fvars_are_real()

Test whether all F-symbols are real.

EXAMPLES:

```
sage: f = FusionRing("A1", 3).get_fmatrix()
sage: f.find_orthogonal_solution(verbose=False) # long time
sage: f.fvars_are_real() # not tested (cypari issue in_
↳doctesting framework)
True
```

```
>>> from sage.all import *
>>> f = FusionRing("A1", Integer(3)).get_fmatrix()
>>> f.find_orthogonal_solution(verbose=False) # long time
>>> f.fvars_are_real() # not tested (cypari issue in_
↳doctesting framework)
True
```

get_coerce_map_from_fr_cyclotomic_field()

Return a coercion map from the associated *FusionRing*'s cyclotomic field into the base field containing all F-symbols (this could be the *FusionRing*'s Cyclotomic field, a *NumberField()*, or *QQbar*).

EXAMPLES:

```
sage: f = FusionRing("G2", 1).get_fmatrix()
sage: f.find_orthogonal_solution(verbose=False)
sage: f.FR().field()
Cyclotomic Field of order 60 and degree 16
sage: f.field()
Number Field in a with defining polynomial y^32 - ... - 22*y^2 + 1
sage: phi = f.get_coerce_map_from_fr_cyclotomic_field()
sage: phi.domain() == f.FR().field()
True
sage: phi.codomain() == f.field()
True
```

```
>>> from sage.all import *
>>> f = FusionRing("G2", Integer(1)).get_fmatrix()
>>> f.find_orthogonal_solution(verbose=False)
>>> f.FR().field()
Cyclotomic Field of order 60 and degree 16
>>> f.field()
Number Field in a with defining polynomial y^32 - ... - 22*y^2 + 1
>>> phi = f.get_coerce_map_from_fr_cyclotomic_field()
>>> phi.domain() == f.FR().field()
True
>>> phi.codomain() == f.field()
True
```

When F-symbols are computed as elements of the associated *FusionRing*'s base Cyclotomic field, we have `self.field() == self.FR().field()` and this returns the identity map on `self.field()`.

```
sage: f = FusionRing("A2", 1).get_fmatrix()
sage: f.find_orthogonal_solution(verbose=False)
sage: phi = f.get_coerce_map_from_fr_cyclotomic_field()
sage: f.field()
Cyclotomic Field of order 48 and degree 16
```

(continues on next page)

(continued from previous page)

```
sage: f.field() == f.FR().field()
True
sage: phi.domain() == f.field()
True
sage: phi.is_identity()
True
```

```
>>> from sage.all import *
>>> f = FusionRing("A2", Integer(1)).get_fmatrix()
>>> f.find_orthogonal_solution(verbose=False)
>>> phi = f.get_coerce_map_from_fr_cyclotomic_field()
>>> f.field()
Cyclotomic Field of order 48 and degree 16
>>> f.field() == f.FR().field()
True
>>> phi.domain() == f.field()
True
>>> phi.is_identity()
True
```

get_defining_equations (*option*, *output=True*)

Get the equations defining the ideal generated by the hexagon or pentagon relations.

INPUT:

- *option* – string determining equations to be set up:
 - 'hexagons' – get equations imposed on the F-matrix by the hexagon relations in the definition of a braided category
 - 'pentagons' – get equations imposed on the F-matrix by the pentagon relations in the definition of a monoidal category
- *output* – boolean (default: `True`); whether results should be returned, where the equations will be polynomials. Otherwise, the constraints are appended to `self.ideal_basis`. Constraints are stored in the internal tuple representation. The `output=False` option is meant only for internal use by the F-matrix solver. When computing the hexagon equations with the `output=False` option, the initial state of the F-symbols is used.

Note

To set up the defining equations using parallel processing, use `start_worker_pool()` to initialize multiple processes *before* calling this method.

EXAMPLES:

```
sage: f = FusionRing("B2", 1).get_fmatrix()
sage: sorted(f.get_defining_equations('hexagons'))
[fx7 + 1,
 fx6 - 1,
 fx2 + 1,
 fx0 - 1,
 fx11*fx12 + (-zeta32^8)*fx13^2 + (zeta32^12)*fx13,
```

(continues on next page)

(continued from previous page)

```

fx10*fx12 + (-zeta32^8)*fx12*fx13 + (zeta32^4)*fx12,
fx10*fx11 + (-zeta32^8)*fx11*fx13 + (zeta32^4)*fx11,
fx10^2 + (-zeta32^8)*fx11*fx12 + (-zeta32^12)*fx10,
fx4*fx9 + fx7,
fx3*fx8 - fx6,
fx1*fx5 + fx2]
sage: pe = f.get_defining_equations('pentagons')
sage: len(pe)
33

```

```

>>> from sage.all import *
>>> f = FusionRing("B2", Integer(1)).get_fmatrix()
>>> sorted(f.get_defining_equations('hexagons'))
[fx7 + 1,
 fx6 - 1,
 fx2 + 1,
 fx0 - 1,
 fx11*fx12 + (-zeta32^8)*fx13^2 + (zeta32^12)*fx13,
 fx10*fx12 + (-zeta32^8)*fx12*fx13 + (zeta32^4)*fx12,
 fx10*fx11 + (-zeta32^8)*fx11*fx13 + (zeta32^4)*fx11,
 fx10^2 + (-zeta32^8)*fx11*fx12 + (-zeta32^12)*fx10,
 fx4*fx9 + fx7,
 fx3*fx8 - fx6,
 fx1*fx5 + fx2]
>>> pe = f.get_defining_equations('pentagons')
>>> len(pe)
33

```

get_fr_str()

Auto-generate an identifying key for saving results.

EXAMPLES:

```

sage: f = FusionRing("B3", 1).get_fmatrix()
sage: f.get_fr_str()
'B31'

```

```

>>> from sage.all import *
>>> f = FusionRing("B3", Integer(1)).get_fmatrix()
>>> f.get_fr_str()
'B31'

```

get_fvars()

Return a dictionary of F-symbols.

The keys are sextuples (a, b, c, d, x, y) of basis elements of $\text{self.FR}()$ and the values are the corresponding F-symbols $(F_d^{a,b,c})_{xy}$.

These values reflect the current state of a solver's computation.

EXAMPLES:

```

sage: f = FusionRing("A2", 1).get_fmatrix(inject_variables=True)
creating variables fx1..fx8
Defining fx0, fx1, fx2, fx3, fx4, fx5, fx6, fx7
sage: f.get_fvars()[(f1, f1, f1, f0, f2, f2)]
fx0
sage: f.find_orthogonal_solution(verbose=False)
sage: f.get_fvars()[(f1, f1, f1, f0, f2, f2)]
1

```

```

>>> from sage.all import *
>>> f = FusionRing("A2", Integer(1)).get_fmatrix(inject_variables=True)
creating variables fx1..fx8
Defining fx0, fx1, fx2, fx3, fx4, fx5, fx6, fx7
>>> f.get_fvars()[(f1, f1, f1, f0, f2, f2)]
fx0
>>> f.find_orthogonal_solution(verbose=False)
>>> f.get_fvars()[(f1, f1, f1, f0, f2, f2)]
1

```

get_fvars_by_size(*n*, *indices*=False)

Return the set of F-symbols that are entries of an $n \times n$ matrix $F_d^{a,b,c}$.

INPUT:

- *n* – positive integer
- *indices* – boolean (default: False)

If *indices* is False (default), this method returns a set of sextuples (a, b, c, d, x, y) identifying the corresponding F-symbol. Each sextuple is a key in the dictionary returned by `get_fvars()`.

Otherwise the method returns a list of integer indices that internally identify the F-symbols. The *indices=True* option is meant for internal use.

EXAMPLES:

```

sage: f = FusionRing("A2", 2).get_fmatrix(inject_variables=True)
creating variables fx1..fx287
Defining fx0, ..., fx286
sage: f.largest_fmat_size()
2
sage: f.get_fvars_by_size(2)
{(f2, f2, f2, f4, f1, f1),
 (f2, f2, f2, f4, f1, f5),
 ...
 (f4, f4, f4, f4, f4, f0),
 (f4, f4, f4, f4, f4, f4)}

```

```

>>> from sage.all import *
>>> f = FusionRing("A2", Integer(2)).get_fmatrix(inject_variables=True)
creating variables fx1..fx287
Defining fx0, ..., fx286
>>> f.largest_fmat_size()
2
>>> f.get_fvars_by_size(Integer(2))

```

(continues on next page)

(continued from previous page)

```
{(f2, f2, f2, f4, f1, f1),
 (f2, f2, f2, f4, f1, f5),
 ...
 (f4, f4, f4, f4, f4, f0),
 (f4, f4, f4, f4, f4, f4)}
```

get_fvars_in_alg_field()

Return F-symbols as elements of the `QQbar`.

This method uses the embedding defined by `get_qqbar_embedding()` to coerce F-symbols into `QQbar`.

EXAMPLES:

```
sage: fr = FusionRing("G2", 1)
sage: f = fr.get_fmatrix(fusion_label='g', inject_variables=True, new=True)
creating variables fx1..fx5
Defining fx0, fx1, fx2, fx3, fx4
sage: f.find_orthogonal_solution(verbose=False)
sage: f.field()
Number Field in a with defining polynomial y^32 - ... - 22*y^2 + 1
sage: f.get_fvars_in_alg_field()
{(g1, g1, g1, g0, g1, g1): 1,
 (g1, g1, g1, g1, g0, g0): 0.61803399? + 0.?e-8*I,
 (g1, g1, g1, g1, g0, g1): -0.7861514? + 0.?e-8*I,
 (g1, g1, g1, g1, g1, g0): -0.7861514? + 0.?e-8*I,
 (g1, g1, g1, g1, g1, g1): -0.61803399? + 0.?e-8*I}
```

```
>>> from sage.all import *
>>> fr = FusionRing("G2", Integer(1))
>>> f = fr.get_fmatrix(fusion_label='g', inject_variables=True, new=True)
creating variables fx1..fx5
Defining fx0, fx1, fx2, fx3, fx4
>>> f.find_orthogonal_solution(verbose=False)
>>> f.field()
Number Field in a with defining polynomial y^32 - ... - 22*y^2 + 1
>>> f.get_fvars_in_alg_field()
{(g1, g1, g1, g0, g1, g1): 1,
 (g1, g1, g1, g1, g0, g0): 0.61803399? + 0.?e-8*I,
 (g1, g1, g1, g1, g0, g1): -0.7861514? + 0.?e-8*I,
 (g1, g1, g1, g1, g1, g0): -0.7861514? + 0.?e-8*I,
 (g1, g1, g1, g1, g1, g1): -0.61803399? + 0.?e-8*I}
```

get_non_cyclotomic_roots()

Return a list of roots that define the extension of the associated *FusionRing*'s base Cyclotomic field, containing all the F-symbols.

OUTPUT:

The list of non-cyclotomic roots is given as a list of elements of the field returned by `field()`.

If `self.field() == self.FR().field()` then this method returns an empty list.

EXAMPLES:

```

sage: f = FusionRing("E6", 1).get_fmatrix()
sage: f.find_orthogonal_solution(verbose=False)
sage: f.field() == f.FR().field()
True
sage: f.get_non_cyclotomic_roots()
[]
sage: f = FusionRing("G2", 1).get_fmatrix()
sage: f.find_orthogonal_solution(verbose=False)
sage: f.field() == f.FR().field()
False
sage: phi = f.get_qqbar_embedding()
sage: [phi(r).n() for r in f.get_non_cyclotomic_roots()]
[-0.786151377757423 - 8.92806368517581e-31*I]

```

```

>>> from sage.all import *
>>> f = FusionRing("E6", Integer(1)).get_fmatrix()
>>> f.find_orthogonal_solution(verbose=False)
>>> f.field() == f.FR().field()
True
>>> f.get_non_cyclotomic_roots()
[]
>>> f = FusionRing("G2", Integer(1)).get_fmatrix()
>>> f.find_orthogonal_solution(verbose=False)
>>> f.field() == f.FR().field()
False
>>> phi = f.get_qqbar_embedding()
>>> [phi(r).n() for r in f.get_non_cyclotomic_roots()]
[-0.786151377757423 - 8.92806368517581e-31*I]

```

When `self.field()` is a `NumberField`, one may use `get_qqbar_embedding()` to embed the resulting values into `QQbar`.

get_orthogonality_constraints (*output=True*)

Get equations imposed on the F-matrix by orthogonality.

INPUT:

- `output` – boolean

OUTPUT:

If `output=True`, orthogonality constraints are returned as polynomial objects.

Otherwise, the constraints are appended to `self.ideal_basis`. They are stored in the internal tuple representation. The `output=False` option is meant mostly for internal use by the F-matrix solver.

EXAMPLES:

```

sage: f = FusionRing("B4", 1).get_fmatrix()
sage: f.get_orthogonality_constraints()
[fx0^2 - 1,
 fx1^2 - 1,
 fx2^2 - 1,
 fx3^2 - 1,
 fx4^2 - 1,
 fx5^2 - 1,

```

(continues on next page)

(continued from previous page)

```

fx6^2 - 1,
fx7^2 - 1,
fx8^2 - 1,
fx9^2 - 1,
fx10^2 + fx12^2 - 1,
fx10*fx11 + fx12*fx13,
fx10*fx11 + fx12*fx13,
fx11^2 + fx13^2 - 1]

```

```

>>> from sage.all import *
>>> f = FusionRing("B4", Integer(1)).get_fmatrix()
>>> f.get_orthogonality_constraints()
[fx0^2 - 1,
 fx1^2 - 1,
 fx2^2 - 1,
 fx3^2 - 1,
 fx4^2 - 1,
 fx5^2 - 1,
 fx6^2 - 1,
 fx7^2 - 1,
 fx8^2 - 1,
 fx9^2 - 1,
 fx10^2 + fx12^2 - 1,
 fx10*fx11 + fx12*fx13,
 fx10*fx11 + fx12*fx13,
 fx11^2 + fx13^2 - 1]

```

get_poly_ring()

Return the polynomial ring whose generators denote the desired F-symbols.

EXAMPLES:

```

sage: f = FusionRing("B6", 1).get_fmatrix()
sage: f.get_poly_ring()
Multivariate Polynomial Ring in fx0, ..., fx13 over
Cyclotomic Field of order 96 and degree 32

```

```

>>> from sage.all import *
>>> f = FusionRing("B6", Integer(1)).get_fmatrix()
>>> f.get_poly_ring()
Multivariate Polynomial Ring in fx0, ..., fx13 over
Cyclotomic Field of order 96 and degree 32

```

get_qqbar_embedding()

Return an embedding from the base field containing F-symbols (the associated *FusionRing*'s *Cyclotomic field*, a *NumberField()*, or *QQbar*) into *QQbar*.

This embedding is useful for getting a better sense for the F-symbols, particularly when they are computed as elements of a *NumberField()*. See also *get_non_cyclotomic_roots()*.

EXAMPLES:

```

sage: fr = FusionRing("G2", 1)
sage: f = fr.get_fmatrix(fusion_label='g', inject_variables=True, new=True)
creating variables fx1..fx5
Defining fx0, fx1, fx2, fx3, fx4
sage: f.find_orthogonal_solution()
Computing F-symbols for The Fusion Ring of Type G2 and level 1 with Integer_
↳ Ring coefficients with 5 variables...
Set up 10 hex and orthogonality constraints...
Partitioned 10 equations into 2 components of size:
[4, 1]
Elimination epoch completed... 0 eqns remain in ideal basis
Hex elim step solved for 4 / 5 variables
Set up 0 reduced pentagons...
Pent elim step solved for 4 / 5 variables
Partitioned 0 equations into 0 components of size:
[]
Partitioned 1 equations into 1 components of size:
[1]
Computing appropriate NumberField...
sage: phi = f.get_qqbar_embedding()
sage: phi(f.fmat(g1, g1, g1, g1, g1, g1)).n()
-0.618033988749895 + 1.46674215951686e-29*I

```

```

>>> from sage.all import *
>>> fr = FusionRing("G2", Integer(1))
>>> f = fr.get_fmatrix(fusion_label='g', inject_variables=True, new=True)
creating variables fx1..fx5
Defining fx0, fx1, fx2, fx3, fx4
>>> f.find_orthogonal_solution()
Computing F-symbols for The Fusion Ring of Type G2 and level 1 with Integer_
↳ Ring coefficients with 5 variables...
Set up 10 hex and orthogonality constraints...
Partitioned 10 equations into 2 components of size:
[4, 1]
Elimination epoch completed... 0 eqns remain in ideal basis
Hex elim step solved for 4 / 5 variables
Set up 0 reduced pentagons...
Pent elim step solved for 4 / 5 variables
Partitioned 0 equations into 0 components of size:
[]
Partitioned 1 equations into 1 components of size:
[1]
Computing appropriate NumberField...
>>> phi = f.get_qqbar_embedding()
>>> phi(f.fmat(g1, g1, g1, g1, g1, g1)).n()
-0.618033988749895 + 1.46674215951686e-29*I

```

get_radical_expression()

Return a radical expression of F-symbols.

EXAMPLES:

```

sage: f = FusionRing("G2", 1).get_fmatrix()
sage: f.FR().fusion_labels("g", inject_variables=True)
sage: f.find_orthogonal_solution(verbose=False)
sage: radical_fvars = f.get_radical_expression()           # long time (~1.5s)
sage: radical_fvars[g1, g1, g1, g1, g1, g0]               # long time
-sqrt(1/2*sqrt(5) - 1/2)

```

```

>>> from sage.all import *
>>> f = FusionRing("G2", Integer(1)).get_fmatrix()
>>> f.FR().fusion_labels("g", inject_variables=True)
>>> f.find_orthogonal_solution(verbose=False)
>>> radical_fvars = f.get_radical_expression()           # long time (~1.5s)
>>> radical_fvars[g1, g1, g1, g1, g1, g0]               # long time
-sqrt(1/2*sqrt(5) - 1/2)

```

largest_fmat_size()

Get the size of the largest F-matrix F_d^{abc} .

EXAMPLES:

```

sage: f = FusionRing("B3", 2).get_fmatrix()
sage: f.largest_fmat_size()
4

```

```

>>> from sage.all import *
>>> f = FusionRing("B3", Integer(2)).get_fmatrix()
>>> f.largest_fmat_size()
4

```

load_fvars(filename)

Load previously computed F-symbols from a pickle file.

See `save_fvars()` for more information.

EXAMPLES:

```

sage: f = FusionRing("A2", 1).get_fmatrix(new=True)
sage: f.find_orthogonal_solution(verbose=False)
sage: fvars = f.get_fvars()
sage: K = f.field()
sage: filename = f.get_fr_str() + "_solver_results.pickle"
sage: f.save_fvars(filename)
sage: del f
sage: f2 = FusionRing("A2", 1).get_fmatrix(new=True)
sage: f2.load_fvars(filename)
sage: fvars == f2.get_fvars()
True
sage: K == f2.field()
True
sage: os.remove(filename)

```

```

>>> from sage.all import *
>>> f = FusionRing("A2", Integer(1)).get_fmatrix(new=True)

```

(continues on next page)

(continued from previous page)

```

>>> f.find_orthogonal_solution(verbose=False)
>>> fvars = f.get_fvars()
>>> K = f.field()
>>> filename = f.get_fr_str() + "_solver_results.pickle"
>>> f.save_fvars(filename)
>>> del f
>>> f2 = FusionRing("A2", Integer(1)).get_fmatrix(new=True)
>>> f2.load_fvars(filename)
>>> fvars == f2.get_fvars()
True
>>> K == f2.field()
True
>>> os.remove(filename)

```

Note

`save_fvars()`. This method does not work with intermediate checkpoint pickles; it only works with pickles containing *all* F-symbols, i.e. those created by `save_fvars()` and by specifying an optional `save_results` parameter for `find_orthogonal_solution()`.

save_fvars(filename)

Save computed F-symbols for later use.

INPUT:

- filename – string specifying the name of the pickle file to be used

The current directory is used unless an absolute path to a file in a different directory is provided.

Note

This method should only be used *after* successfully running one of the solvers, e.g. `find_cyclo-tomic_solution()` or `find_orthogonal_solution()`.

When used in conjunction with `load_fvars()`, this method may be used to restore state of an *FMatrix* object at the end of a successful F-matrix solver run.

EXAMPLES:

```

sage: f = FusionRing("A2", 1).get_fmatrix(new=True)
sage: f.find_orthogonal_solution(verbose=False)
sage: fvars = f.get_fvars()
sage: K = f.field()
sage: filename = f.get_fr_str() + "_solver_results.pickle"
sage: f.save_fvars(filename)
sage: del f
sage: f2 = FusionRing("A2", 1).get_fmatrix(new=True)
sage: f2.load_fvars(filename)
sage: fvars == f2.get_fvars()
True
sage: K == f2.field()

```

(continues on next page)

(continued from previous page)

```
True
sage: os.remove(filename)
```

```
>>> from sage.all import *
>>> f = FusionRing("A2", Integer(1)).get_fmatrix(new=True)
>>> f.find_orthogonal_solution(verbose=False)
>>> fvars = f.get_fvars()
>>> K = f.field()
>>> filename = f.get_fr_str() + "_solver_results.pickle"
>>> f.save_fvars(filename)
>>> del f
>>> f2 = FusionRing("A2", Integer(1)).get_fmatrix(new=True)
>>> f2.load_fvars(filename)
>>> fvars == f2.get_fvars()
True
>>> K == f2.field()
True
>>> os.remove(filename)
```

shutdown_worker_pool()

Shutdown the given worker pool and dispose of shared memory resources created when the pool was set up using `start_worker_pool()`.

Warning

Failure to call this method after using `start_worker_pool()` to create a process pool may result in a memory leak, since shared memory resources outlive the process that created them.

EXAMPLES:

```
sage: f = FusionRing("A1", 3).get_fmatrix(new=True)
sage: f.start_worker_pool()
sage: he = f.get_defining_equations('hexagons')
sage: f.shutdown_worker_pool()
```

```
>>> from sage.all import *
>>> f = FusionRing("A1", Integer(3)).get_fmatrix(new=True)
>>> f.start_worker_pool()
>>> he = f.get_defining_equations('hexagons')
>>> f.shutdown_worker_pool()
```

start_worker_pool(processes=None)

Initialize a multiprocessing worker pool for parallel processing, which may be used e.g. to set up defining equations using `get_defining_equations()`.

This method sets `self`'s `pool` attribute. The worker pool may be used time and again. Upon initialization, each process in the pool attaches to the necessary shared memory resources.

When you are done using the worker pool, use `shutdown_worker_pool()` to close the pool and properly dispose of shared memory resources.

INPUT:

- `processes` – integer indicating the number of workers in the pool; if left unspecified, the number of workers equals the number of processors available

OUTPUT: boolean; whether a worker pool was successfully initialized

EXAMPLES:

```
sage: f = FusionRing("G2", 1).get_fmatrix(new=True)
sage: f.start_worker_pool()
sage: he = f.get_defining_equations('hexagons')
sage: sorted(he)
[fx0 - 1,
 fx2*fx3 + (zeta60^14 + zeta60^12 - zeta60^6 - zeta60^4 + 1)*fx4^2 + (zeta60^
 ↪ 6)*fx4,
 fx1*fx3 + (zeta60^14 + zeta60^12 - zeta60^6 - zeta60^4 + 1)*fx3*fx4 + ↪
 ↪ (zeta60^14 - zeta60^4)*fx3,
 fx1*fx2 + (zeta60^14 + zeta60^12 - zeta60^6 - zeta60^4 + 1)*fx2*fx4 + ↪
 ↪ (zeta60^14 - zeta60^4)*fx2,
 fx1^2 + (zeta60^14 + zeta60^12 - zeta60^6 - zeta60^4 + 1)*fx2*fx3 + (-zeta60^
 ↪ 12)*fx1]
sage: pe = f.get_defining_equations('pentagons')
sage: f.shutdown_worker_pool()
```

```
>>> from sage.all import *
>>> f = FusionRing("G2", Integer(1)).get_fmatrix(new=True)
>>> f.start_worker_pool()
>>> he = f.get_defining_equations('hexagons')
>>> sorted(he)
[fx0 - 1,
 fx2*fx3 + (zeta60^14 + zeta60^12 - zeta60^6 - zeta60^4 + 1)*fx4^2 + (zeta60^
 ↪ 6)*fx4,
 fx1*fx3 + (zeta60^14 + zeta60^12 - zeta60^6 - zeta60^4 + 1)*fx3*fx4 + ↪
 ↪ (zeta60^14 - zeta60^4)*fx3,
 fx1*fx2 + (zeta60^14 + zeta60^12 - zeta60^6 - zeta60^4 + 1)*fx2*fx4 + ↪
 ↪ (zeta60^14 - zeta60^4)*fx2,
 fx1^2 + (zeta60^14 + zeta60^12 - zeta60^6 - zeta60^4 + 1)*fx2*fx3 + (-zeta60^
 ↪ 12)*fx1]
>>> pe = f.get_defining_equations('pentagons')
>>> f.shutdown_worker_pool()
```

Warning

This method is needed to initialize the worker pool using the necessary shared memory resources. Simply using the `multiprocessing.Pool` constructor will not work with our class methods.

Warning

Failure to call `shutdown_worker_pool()` may result in a memory leak, since shared memory resources outlive the process that created them.

5.8.3 The Fusion Ring of the Drinfeld Double of a Finite Group

class sage.algebras.fusion_rings.fusion_double.**FusionDouble** (*G*, *prefix*='s')

Bases: [CombinatorialFreeModule](#)

The fusion ring corresponding to the Drinfeld double of a finite group.

This is the fusion ring of the modular tensor category of modules over the Drinfeld double of a finite group. Usage is similar to [FusionRing](#); we refer the reader to that class for more information.

INPUT:

- *G* – a finite group
- *prefix* – (default: 's') a prefix for the names of simple objects
- *inject_variables* – (optional) set to `True` to create variables for the simple objects

REFERENCES:

- [BaKi2001] Chapter 3
- [Mas1995]
- [CHW2015]
- [Goff1999]

EXAMPLES:

```
sage: G = DihedralGroup(5)
sage: H = FusionDouble(G, inject_variables=True)
sage: H.basis()
Finite family {0: s0, 1: s1, 2: s2, 3: s3, 4: s4, 5: s5, 6: s6, 7: s7, 8: s8,
              9: s9, 10: s10, 11: s11, 12: s12, 13: s13, 14: s14, 15: s15}
sage: for x in H.basis():
....:     print ("%s : %s"%(x,x^2))
....:
s0 : s0
s1 : s0
s2 : s0 + s1 + s3
s3 : s0 + s1 + s2
s4 : s0 + s2 + s3 + s6 + s7 + s8 + s9 + s10 + s11 + s12 + s13 + s14 + s15
s5 : s0 + s2 + s3 + s6 + s7 + s8 + s9 + s10 + s11 + s12 + s13 + s14 + s15
s6 : s0 + s1 + s11
s7 : s0 + s1 + s13
s8 : s0 + s1 + s15
s9 : s0 + s1 + s12
s10 : s0 + s1 + s14
s11 : s0 + s1 + s6
s12 : s0 + s1 + s9
s13 : s0 + s1 + s7
s14 : s0 + s1 + s10
s15 : s0 + s1 + s8
sage: s4*s5
s1 + s2 + s3 + s6 + s7 + s8 + s9 + s10 + s11 + s12 + s13 + s14 + s15
sage: s4.ribbon()
1
sage: s5.ribbon()
```

(continues on next page)

(continued from previous page)

```

-1
sage: s8.ribbon()
zeta5^3

>>> from sage.all import *
>>> G = DihedralGroup(Integer(5))
>>> H = FusionDouble(G, inject_variables=True)
>>> H.basis()
Finite family {0: s0, 1: s1, 2: s2, 3: s3, 4: s4, 5: s5, 6: s6, 7: s7, 8: s8,
               9: s9, 10: s10, 11: s11, 12: s12, 13: s13, 14: s14, 15: s15}
>>> for x in H.basis():
...     print ("%s : %s"%(x,x**Integer(2)))
....:
s0 : s0
s1 : s0
s2 : s0 + s1 + s3
s3 : s0 + s1 + s2
s4 : s0 + s2 + s3 + s6 + s7 + s8 + s9 + s10 + s11 + s12 + s13 + s14 + s15
s5 : s0 + s2 + s3 + s6 + s7 + s8 + s9 + s10 + s11 + s12 + s13 + s14 + s15
s6 : s0 + s1 + s11
s7 : s0 + s1 + s13
s8 : s0 + s1 + s15
s9 : s0 + s1 + s12
s10 : s0 + s1 + s14
s11 : s0 + s1 + s6
s12 : s0 + s1 + s9
s13 : s0 + s1 + s7
s14 : s0 + s1 + s10
s15 : s0 + s1 + s8
>>> s4*s5
s1 + s2 + s3 + s6 + s7 + s8 + s9 + s10 + s11 + s12 + s13 + s14 + s15
>>> s4.ribbon()
1
>>> s5.ribbon()
-1
>>> s8.ribbon()
zeta5^3

```

If the fusion double is multiplicity-free, meaning that the fusion coefficients N_k^{ij} are bounded by 1, then the F-matrix may be computed, by solving the pentagon and hexagon relations as described in [Bond2007] and [Ab2022], just as for *FusionRing*. There is a caveat here, since even if the fusion rules are multiplicity-free, if there are too many F-matrix values to compute, even if many of them are zero, in the current implementation singular cannot create enough variables. At least, this code can compute the F-matrix for the Fusion Double of the symmetric group S_3 , duplicating the result of [CHW2015].

```

sage: G1 = SymmetricGroup(3)
sage: H1 = FusionDouble(G1, prefix='u', inject_variables=True)
sage: F = H1.get_fmatrix()

```

```

>>> from sage.all import *
>>> G1 = SymmetricGroup(Integer(3))
>>> H1 = FusionDouble(G1, prefix='u', inject_variables=True)

```

(continues on next page)

(continued from previous page)

```
>>> F = H1.get_fmatrix()
```

The above commands create the F-matrix. You can compute all of the F-matrices with the command:

```
sage: H1.find_orthogonal_solution() # not tested (10-15 minutes)
```

```
>>> from sage.all import *
>>> H1.find_orthogonal_solution() # not tested (10-15 minutes)
```

Individual F-matrices may be computed thus:

```
sage: F.fmatrix(u3, u3, u3, u4) # not tested
```

```
>>> from sage.all import *
>>> F.fmatrix(u3, u3, u3, u4) # not tested
```

See `FMatrix` for more information.

Unfortunately beyond S_3 the number of simple objects is seemingly impractical. Although the `FusionDouble` class and its methods work well for groups of moderate size, the `FMatrix` may not be computable. For the dihedral group of order 8, there are already 22 simple objects, and the F-matrix seems out of reach. The actual limitation is that singular will not create a polynomial ring in more than $2^{15} - 1 = 32767$ symbols, and there are more than this many F-matrix values to be computed for the dihedral group of order 8, so in the current implementation, this `FusionRing` is out of reach.

It is an open problem to classify the finite groups whose fusion doubles are multiplicity-free. Abelian groups, dihedral groups, dicyclic groups, and all groups of order 16 are multiplicity-free. On the other hand, for groups of order 32, some are multiplicity-free and others are not. These can all be constructed using `SmallPermutationGroup`.

EXAMPLES:

```
sage: G = SmallPermutationGroup(16,9)
sage: F = FusionDouble(G, prefix='b', inject_variables=True)
sage: b13^2 # long time (4s)
b0 + b3 + b4
```

```
>>> from sage.all import *
>>> G = SmallPermutationGroup(Integer(16), Integer(9))
>>> F = FusionDouble(G, prefix='b', inject_variables=True)
>>> b13**Integer(2) # long time (4s)
b0 + b3 + b4
```

D_minus (*base_coercion=True*)

Return the positive square root of `self.global_q_dimension()` as an element of `self.field()`.

For the Drinfeld double of a finite group G , this equals the cardinality of G . This is also equal to $\sum d_i^2 \theta_i^{\pm 1}$, where i runs through the simple objects, d_i is the quantum dimension, and θ_i is the twist. This sum with θ_i is denoted p_- in [BaKi2001] Chapter 3.

EXAMPLES:

```
sage: FusionDouble(DihedralGroup(7)).total_q_order()
14
```

```
>>> from sage.all import *
>>> FusionDouble(DihedralGroup(Integer(7))).total_q_order()
14
```

D_plus (*base_coercion=True*)

Return the positive square root of `self.global_q_dimension()` as an element of `self.field()`.

For the Drinfeld double of a finite group G , this equals the cardinality of G . This is also equal to $\sum d_i^2 \theta_i^{\pm 1}$, where i runs through the simple objects, d_i is the quantum dimension, and θ_i is the twist. This sum with θ_i is denoted p_- in [BaKi2001] Chapter 3.

EXAMPLES:

```
sage: FusionDouble(DihedralGroup(7)).total_q_order()
14
```

```
>>> from sage.all import *
>>> FusionDouble(DihedralGroup(Integer(7))).total_q_order()
14
```

class Element

Bases: `IndexedFreeModuleElement`

char()

Return the character χ corresponding to `self`.

The data determining a simple object consists of a conjugacy class representative g and an irreducible character χ of the centralizer of g .

See also

`g()`

EXAMPLES:

```
sage: G = DihedralGroup(5)
sage: H = FusionDouble(G, prefix='f', inject_variables=True)
sage: f10.g()
(1, 2, 3, 4, 5)
sage: f10.char()
Character of Subgroup generated by [(1, 2, 3, 4, 5)] of
(Dihedral group of order 10 as a permutation group)
```

```
>>> from sage.all import *
>>> G = DihedralGroup(Integer(5))
>>> H = FusionDouble(G, prefix='f', inject_variables=True)
>>> f10.g()
(1, 2, 3, 4, 5)
>>> f10.char()
Character of Subgroup generated by [(1, 2, 3, 4, 5)] of
(Dihedral group of order 10 as a permutation group)
```

dual()

Return the dual of `self`.

This method is only available for simple objects.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: H = FusionDouble(G, prefix='j')
sage: [x for x in H.basis() if x == x.dual()]
[j0, j1, j8, j9]
```

```
>>> from sage.all import *
>>> G = CyclicPermutationGroup(Integer(4))
>>> H = FusionDouble(G, prefix='j')
>>> [x for x in H.basis() if x == x.dual()]
[j0, j1, j8, j9]
```

g()

The data determining a simple object consists of a conjugacy class representative g and an irreducible character χ of the centralizer of g .

This returns the conjugacy class representative of the underlying group corresponding to a simple object. See also `char()`.

EXAMPLES:

```
sage: G = QuaternionGroup()
sage: H = FusionDouble(G, prefix='e', inject_variables=True)
sage: e10.g()
(1, 3) (2, 4) (5, 7) (6, 8)
sage: e10.char()
Character of Subgroup generated by [(1, 2, 3, 4) (5, 6, 7, 8), (1, 5, 3, 7) (2, 8, 4,
↪ 6)]
of (Quaternion group of order 8 as a permutation group)
```

```
>>> from sage.all import *
>>> G = QuaternionGroup()
>>> H = FusionDouble(G, prefix='e', inject_variables=True)
>>> e10.g()
(1, 3) (2, 4) (5, 7) (6, 8)
>>> e10.char()
Character of Subgroup generated by [(1, 2, 3, 4) (5, 6, 7, 8), (1, 5, 3, 7) (2, 8, 4,
↪ 6)]
of (Quaternion group of order 8 as a permutation group)
```

is_simple_object()

Determine whether `self` is a simple object (basis element) of the fusion ring.

EXAMPLES:

```
sage: H = FusionDouble(CyclicPermutationGroup(2), prefix='g', inject_
↪ variables=True)
sage: [x.is_simple_object() for x in [g0, g1, g0+g1]]
[True, True, False]
```

```
>>> from sage.all import *
>>> H = FusionDouble(CyclicPermutationGroup(Integer(2)), prefix='g',
↳ inject_variables=True)
>>> [x.is_simple_object() for x in [g0, g1, g0+g1]]
[True, True, False]
```

q_dimension (*base_coercion=True*)

Return the q-dimension of self.

This method is only available for simple objects.

EXAMPLES:

```
sage: G = AlternatingGroup(4)
sage: H = FusionDouble(G)
sage: [x.q_dimension() for x in H.basis()]
[1, 1, 1, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4]
sage: sum(x.q_dimension()^2 for x in H.basis()) == G.order()^2
True
```

```
>>> from sage.all import *
>>> G = AlternatingGroup(Integer(4))
>>> H = FusionDouble(G)
>>> [x.q_dimension() for x in H.basis()]
[1, 1, 1, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4]
>>> sum(x.q_dimension()**Integer(2) for x in H.basis()) == G.
↳ order()**Integer(2)
True
```

ribbon (*base_coercion=True*)

Return the twist or ribbon of the simple object.

EXAMPLES:

```
sage: H = FusionDouble(CyclicPermutationGroup(3))
sage: [i.ribbon() for i in H.basis()]
[1, 1, 1, 1, zeta3, -zeta3 - 1, 1, -zeta3 - 1, zeta3]
```

```
>>> from sage.all import *
>>> H = FusionDouble(CyclicPermutationGroup(Integer(3)))
>>> [i.ribbon() for i in H.basis()]
[1, 1, 1, 1, zeta3, -zeta3 - 1, 1, -zeta3 - 1, zeta3]
```

twist (*reduced=True*)

Return a rational number h such that $\theta = e^{i\pi h}$ is the twist of self.

The quantity $e^{i\pi h}$ is also available using `ribbon()`.

This method is only available for simple objects.

EXAMPLES:

```
sage: Q=FusionDouble(CyclicPermutationGroup(3))
sage: [x.twist() for x in Q.basis()]
[0, 0, 0, 0, 2/3, 4/3, 0, 4/3, 2/3]
```

(continues on next page)

(continued from previous page)

```
sage: [x.ribbon() for x in Q.basis()]
[1, 1, 1, 1, zeta3, -zeta3 - 1, 1, -zeta3 - 1, zeta3]
```

```
>>> from sage.all import *
>>> Q=FusionDouble(CyclicPermutationGroup(Integer(3)))
>>> [x.twist() for x in Q.basis()]
[0, 0, 0, 0, 2/3, 4/3, 0, 4/3, 2/3]
>>> [x.ribbon() for x in Q.basis()]
[1, 1, 1, 1, zeta3, -zeta3 - 1, 1, -zeta3 - 1, zeta3]
```

N_ijk(*i, j, k*)

The symmetric invariant of three simple objects.

This is the dimension of

$$\text{Hom}(i \otimes j \otimes k, s_0),$$

where s_0 is the unit element (assuming `prefix='s'`). Method of computation is through the Verlinde formula, deducing the values from the known values of the S -matrix.

EXAMPLES:

```
sage: A = FusionDouble(AlternatingGroup(4), prefix='a', inject_variables=True)
sage: [A.N_ijk(a10, a11, x) for x in A.basis()]
[0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0]
```

```
>>> from sage.all import *
>>> A = FusionDouble(AlternatingGroup(Integer(4)), prefix='a', inject_
↪ variables=True)
>>> [A.N_ijk(a10, a11, x) for x in A.basis()]
[0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0]
```

Nk_ij(*i, j, k, use_characters=False*)Return the fusion coefficient N_{ij}^k .

INPUT:

- *i, j, k* – basis elements
- `use_characters` – boolean (default: `False`); see the algorithm description below

ALGORITHM:

If `use_characters=False`, then this is computed using the Verlinde formula:

$$N_{ij}^k = \sum_l \frac{s(i, \ell) s(j, \ell) \overline{s(k, \ell)}}{s(I, \ell)}.$$

Otherwise we use a character theoretic method to compute the fusion coefficient N_{ij}^k as follows. Each simple object, for example i corresponds to a conjugacy class $\mathcal{C}_i$ of the underlying group G , and an irreducible character χ_i of the centralizer $C(g_i)$ of a fixed representative g_i of $\mathcal{C}_i$. In addition to the fixed representative g_k of the class $\mathcal{C}_i$ and $\mathcal{C}_j$, the formula will make use of variable elements h_i and h_j that are subject to the condition $h_i h_j = g_k$. See [GoMa2010] equation (7).

$$\frac{|\mathcal{C}_k|}{|G|} \sum_{\substack{h_i \in \mathcal{C}_i \\ h_j \in \mathcal{C}_j \\ h_i h_j = g_k}} |C(h_i) \cap C(h_j)| \langle \chi_i^{(h_i)} \chi_j^{(h_j)}, \chi_k \rangle_{C(h_i) \cap C(h_j)},$$

where $\chi_i^{(h_i)}$ is the character χ_i of $C(g_i)$ conjugated to a character of $C(h_i)$, when h_i is a conjugate of the fixed representative g_i . More exactly, there exists r_i such that $r_i g_i r_i^{-1} = h_i$, and then $\chi_i^{(h_i)}(x) = \chi_i(r_i^{-1} x r_i)$, and this definition does not depend on the choice of r_i .

Note

This should be functionally equivalent, and testing shows that it is, but it is slower.

EXAMPLES:

```
sage: A = FusionDouble(AlternatingGroup(4), prefix='aa', inject_variables=True)
sage: [A.Nk_ij(aa8, aa10, x) for x in A.basis()]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 1]

sage: B = FusionDouble(CyclicPermutationGroup(2))
sage: all(B.Nk_ij(x, y, z, use_characters=True) == B.Nk_ij(x, y, z)
....:      for x in B.basis() for y in B.basis() for z in B.basis())
True
```

```
>>> from sage.all import *
>>> A = FusionDouble(AlternatingGroup(Integer(4)), prefix='aa', inject_
↳ variables=True)
>>> [A.Nk_ij(aa8, aa10, x) for x in A.basis()]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 1]

>>> B = FusionDouble(CyclicPermutationGroup(Integer(2)))
>>> all(B.Nk_ij(x, y, z, use_characters=True) == B.Nk_ij(x, y, z)
...      for x in B.basis() for y in B.basis() for z in B.basis())
True
```

dual(i)

Return the dual object i^{ast} to i .

The dual is also available as an element method of i .

EXAMPLES:

```
sage: K = FusionDouble(CyclicPermutationGroup(3), prefix='k')
sage: [(x, K.dual(x)) for x in K.basis()]
[(k0, k0),
 (k1, k2),
 (k2, k1),
 (k3, k6),
 (k4, k8),
 (k5, k7),
 (k6, k3),
 (k7, k5),
 (k8, k4)]
sage: all(K.dual(x) == x.dual() for x in K.basis())
True
```

```
>>> from sage.all import *
>>> K = FusionDouble(CyclicPermutationGroup(Integer(3)), prefix='k')
```

(continues on next page)

(continued from previous page)

```
>>> [(x, K.dual(x)) for x in K.basis()]
[(k0, k0),
 (k1, k2),
 (k2, k1),
 (k3, k6),
 (k4, k8),
 (k5, k7),
 (k6, k3),
 (k7, k5),
 (k8, k4)]
>>> all(K.dual(x)==x.dual() for x in K.basis())
True
```

field()

Return a cyclotomic field large enough to contain the values of R-matrices and twists that can arise for this fusion ring.

EXAMPLES:

```
sage: FusionDouble(SymmetricGroup(3)).field()
Cyclotomic Field of order 24 and degree 8
```

```
>>> from sage.all import *
>>> FusionDouble(SymmetricGroup(Integer(3))).field()
Cyclotomic Field of order 24 and degree 8
```

fvars_field()

Return a field containing the CyclotomicField computed by *field()* as well as all the F-symbols of the associated FMatrix factory object.

This method is only available if *self* is multiplicity-free.

EXAMPLES:

```
sage: FusionDouble(SymmetricGroup(3)).fvars_field()
Cyclotomic Field of order 24 and degree 8
```

```
>>> from sage.all import *
>>> FusionDouble(SymmetricGroup(Integer(3))).fvars_field()
Cyclotomic Field of order 24 and degree 8
```

get_fmatrix(*args, **kwargs)

Construct an FMatrix factory to solve the pentagon and hexagon relations and organize the resulting F-symbols.

EXAMPLES:

```
sage: f = FusionDouble(SymmetricGroup(3)).get_fmatrix(); f
F-Matrix factory for The Fusion Ring of the Drinfeld Double of
Symmetric group of order 3! as a permutation group
```

```
>>> from sage.all import *
>>> f = FusionDouble(SymmetricGroup(Integer(3))).get_fmatrix(); f
```

(continues on next page)

(continued from previous page)

```
F-Matrix factory for The Fusion Ring of the Drinfeld Double of
Symmetric group of order 3! as a permutation group
```

global_q_dimension(*base_coercion=True*)

Return the global quantum dimension, which is the sum of the squares of the quantum dimensions of the simple objects.

For the Drinfeld double, it is the square of the order of the underlying quantum group.

EXAMPLES:

```
sage: G = SymmetricGroup(4)
sage: H = FusionDouble(G)
sage: H.global_q_dimension()
576
sage: sum(x.q_dimension()^2 for x in H.basis())
576
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(4))
>>> H = FusionDouble(G)
>>> H.global_q_dimension()
576
>>> sum(x.q_dimension()**Integer(2) for x in H.basis())
576
```

group()

Return the underlying group.

EXAMPLES:

```
sage: FusionDouble(DiCyclicGroup(4)).group()
Dicyclic group of order 16 as a permutation group
```

```
>>> from sage.all import *
>>> FusionDouble(DiCyclicGroup(Integer(4))).group()
Dicyclic group of order 16 as a permutation group
```

inject_variables()

Create variables for the simple objects in the global name space.

EXAMPLES:

```
sage: F = FusionDouble(DiCyclicGroup(3), prefix='d')
sage: F.inject_variables()
sage: d0 + d1 + d5
d0 + d1 + d5
```

```
>>> from sage.all import *
>>> F = FusionDouble(DiCyclicGroup(Integer(3)), prefix='d')
>>> F.inject_variables()
>>> d0 + d1 + d5
d0 + d1 + d5
```


is_multiplicity_free (*verbose=False*)

Return True if all fusion coefficients are at most 1.

EXAMPLES:

```
sage: FusionDouble(SymmetricGroup(3)).is_multiplicity_free()
True
sage: FusionDouble(SymmetricGroup(4)).is_multiplicity_free()
False

sage: FusionDouble(SymmetricGroup(3)).is_multiplicity_free(True)
Checking multiplicity freeness
True
sage: FusionDouble(SymmetricGroup(4)).is_multiplicity_free(True)
Checking multiplicity freeness
N(s2,s13,s13) = 2
False
```

```
>>> from sage.all import *
>>> FusionDouble(SymmetricGroup(Integer(3))).is_multiplicity_free()
True
>>> FusionDouble(SymmetricGroup(Integer(4))).is_multiplicity_free()
False

>>> FusionDouble(SymmetricGroup(Integer(3))).is_multiplicity_free(True)
Checking multiplicity freeness
True
>>> FusionDouble(SymmetricGroup(Integer(4))).is_multiplicity_free(True)
Checking multiplicity freeness
N(s2,s13,s13) = 2
False
```

one_basis ()

The unit element of the ring, which is the first basis element.

EXAMPLES:

```
sage: FusionDouble(CyclicPermutationGroup(2), prefix='h').one()
h1
```

```
>>> from sage.all import *
>>> FusionDouble(CyclicPermutationGroup(Integer(2)), prefix='h').one()
h1
```

product_on_basis (*a, b*)

Return the product of two basis elements corresponding to keys *a* and *b*.

INPUT:

- *a, b* – keys for the dictionary `self._names` representing simple objects

EXAMPLES:

```
sage: Q=FusionDouble(SymmetricGroup(3),prefix='q',inject_variables=True)
sage: q3*q4
```

(continues on next page)

(continued from previous page)

```

q1 + q2 + q5 + q6 + q7
sage: Q._names
{0: 'q0', 1: 'q1', 2: 'q2', 3: 'q3', 4: 'q4', 5: 'q5', 6: 'q6', 7: 'q7'}
sage: Q.product_on_basis(3,4)
q1 + q2 + q5 + q6 + q7

```

```

>>> from sage.all import *
>>> Q=FusionDouble(SymmetricGroup(Integer(3)),prefix='q',inject_
↳variables=True)
>>> q3*q4
q1 + q2 + q5 + q6 + q7
>>> Q._names
{0: 'q0', 1: 'q1', 2: 'q2', 3: 'q3', 4: 'q4', 5: 'q5', 6: 'q6', 7: 'q7'}
>>> Q.product_on_basis(Integer(3),Integer(4))
q1 + q2 + q5 + q6 + q7

```

r_matrix(*i, j, k, base_coercion=True*)

Return the R-matrix entry corresponding to the subobject *k* in the tensor product of *i* with *j*. This method is only correct if the fusion coefficient $N_{ij}^k \leq 1$. See the [FusionRing](#) method for more information, including the reason for this caveat, and the algorithm.

EXAMPLES:

```

sage: C = FusionDouble(SymmetricGroup(3),prefix='c',inject_variables=True)
sage: c4*c5
c3 + c4
sage: [C.r_matrix(c4,c5,k) for k in [c3,c4]]
[-zeta24^6, 1]
sage: c6^2
c0 + c1 + c6
sage: [C.r_matrix(c6,c6,k) for k in [c0,c1,c6]]
[zeta3, -zeta3, -zeta3 - 1]

```

```

>>> from sage.all import *
>>> C = FusionDouble(SymmetricGroup(Integer(3)),prefix='c',inject_
↳variables=True)
>>> c4*c5
c3 + c4
>>> [C.r_matrix(c4,c5,k) for k in [c3,c4]]
[-zeta24^6, 1]
>>> c6**Integer(2)
c0 + c1 + c6
>>> [C.r_matrix(c6,c6,k) for k in [c0,c1,c6]]
[zeta3, -zeta3, -zeta3 - 1]

```

root_of_unity(*r, base_coercion=True*)

Return $e^{i\pi r}$ as an element of `self.field()` if possible.

INPUT:

- *r* – a rational number

EXAMPLES:

```

sage: H = FusionDouble(DihedralGroup(6))
sage: H.field()
Cyclotomic Field of order 24 and degree 8
sage: for n in [1..7]:
.....:     try:
.....:         print (n,H.root_of_unity(2/n))
.....:     except ValueError as err:
.....:         print (n,err)
.....:
1 1
2 -1
3 zeta24^4 - 1
4 zeta24^6
5 not a root of unity in the field
6 zeta24^4
7 not a root of unity in the field

```

```

>>> from sage.all import *
>>> H = FusionDouble(DihedralGroup(Integer(6)))
>>> H.field()
Cyclotomic Field of order 24 and degree 8
>>> for n in (ellipsis_range(Integer(1),Ellipsis,Integer(7))):
...     try:
...         print (n,H.root_of_unity(Integer(2)/n))
...     except ValueError as err:
...         print (n,err)
.....:
1 1
2 -1
3 zeta24^4 - 1
4 zeta24^6
5 not a root of unity in the field
6 zeta24^4
7 not a root of unity in the field

```

s_ij(*i*, *j*, *unitary=False*, *base_coercion=True*)

Return the element of the S -matrix of this fusion ring corresponding to the given elements.

Without the unitary option set true, this is the unnormalized S -matrix entry, denoted $\tilde{s}_{ij}$, in [BaKi2001] Chapter 3. The normalized S -matrix entries are denoted s_{ij} .

INPUT:

- *i*, *j*, – a pair of basis elements
- *unitary* – boolean (default: False); set to True to obtain the unitary S -matrix

EXAMPLES:

```

sage: D = FusionDouble(SymmetricGroup(3), prefix='t', inject_variables=True)
sage: [D.s_ij(t2, x) for x in D.basis()]
[2, 2, 4, 0, 0, -2, -2, -2]
sage: [D.s_ij(t2, x, unitary=True) for x in D.basis()]
[1/3, 1/3, 2/3, 0, 0, -1/3, -1/3, -1/3]

```

```
>>> from sage.all import *
>>> D = FusionDouble(SymmetricGroup(Integer(3)), prefix='t', inject_
↳variables=True)
>>> [D.s_ij(t2, x) for x in D.basis()]
[2, 2, 4, 0, 0, -2, -2, -2]
>>> [D.s_ij(t2, x, unitary=True) for x in D.basis()]
[1/3, 1/3, 2/3, 0, 0, -1/3, -1/3, -1/3]
```

s_ijconj(*i*, *j*, *unitary=False*, *base_coercion=True*)

Return the conjugate of the element of the S -matrix given by `self.s_ij(elt_i, elt_j, base_coercion=base_coercion)`.

See also

`s_ij()`

EXAMPLES:

```
sage: P=FusionDouble(CyclicPermutationGroup(3),prefix='p',inject_
↳variables=True)
sage: P.s_ij(p1,p3)
zeta3
sage: P.s_ijconj(p1,p3)
-zeta3 - 1
```

```
>>> from sage.all import *
>>> P=FusionDouble(CyclicPermutationGroup(Integer(3)),prefix='p',inject_
↳variables=True)
>>> P.s_ij(p1,p3)
zeta3
>>> P.s_ijconj(p1,p3)
-zeta3 - 1
```

s_matrix(*unitary=False*, *base_coercion=True*)

Return the S -matrix of this fusion ring.

OPTIONAL:

- `unitary` – boolean (default: `False`); set to `True` to obtain the unitary S -matrix

Without the `unitary` parameter, this is the matrix denoted $\tilde{s}$ in [BaKi2001].

EXAMPLES:

```
sage: FusionDouble(SymmetricGroup(3)).s_matrix()
[ 1  1  2  3  3  2  2  2]
[ 1  1  2 -3 -3  2  2  2]
[ 2  2  4  0  0 -2 -2 -2]
[ 3 -3  0  3 -3  0  0  0]
[ 3 -3  0 -3  3  0  0  0]
[ 2  2 -2  0  0  4 -2 -2]
[ 2  2 -2  0  0 -2 -2  4]
[ 2  2 -2  0  0 -2  4 -2]
```

(continues on next page)

(continued from previous page)

```
sage: FusionDouble(SymmetricGroup(3)).s_matrix(unitary=True)
[ 1/6  1/6  1/3  1/2  1/2  1/3  1/3  1/3]
[ 1/6  1/6  1/3 -1/2 -1/2  1/3  1/3  1/3]
[ 1/3  1/3  2/3   0   0 -1/3 -1/3 -1/3]
[ 1/2 -1/2   0  1/2 -1/2   0   0   0]
[ 1/2 -1/2   0 -1/2  1/2   0   0   0]
[ 1/3  1/3 -1/3   0   0  2/3 -1/3 -1/3]
[ 1/3  1/3 -1/3   0   0 -1/3 -1/3  2/3]
[ 1/3  1/3 -1/3   0   0 -1/3  2/3 -1/3]
```

```
>>> from sage.all import *
>>> FusionDouble(SymmetricGroup(Integer(3))).s_matrix()
[ 1  1  2  3  3  2  2  2]
[ 1  1  2 -3 -3  2  2  2]
[ 2  2  4  0  0 -2 -2 -2]
[ 3 -3  0  3 -3  0  0  0]
[ 3 -3  0 -3  3  0  0  0]
[ 2  2 -2  0  0  4 -2 -2]
[ 2  2 -2  0  0 -2 -2  4]
[ 2  2 -2  0  0 -2  4 -2]
>>> FusionDouble(SymmetricGroup(Integer(3))).s_matrix(unitary=True)
[ 1/6  1/6  1/3  1/2  1/2  1/3  1/3  1/3]
[ 1/6  1/6  1/3 -1/2 -1/2  1/3  1/3  1/3]
[ 1/3  1/3  2/3   0   0 -1/3 -1/3 -1/3]
[ 1/2 -1/2   0  1/2 -1/2   0   0   0]
[ 1/2 -1/2   0 -1/2  1/2   0   0   0]
[ 1/3  1/3 -1/3   0   0  2/3 -1/3 -1/3]
[ 1/3  1/3 -1/3   0   0 -1/3 -1/3  2/3]
[ 1/3  1/3 -1/3   0   0 -1/3  2/3 -1/3]
```

total_q_order (*base_coercion=True*)

Return the positive square root of `self.global_q_dimension()` as an element of `self.field()`.

For the Drinfeld double of a finite group G , this equals the cardinality of G . This is also equal to $\sum d_i^2 \theta_i^{\pm 1}$, where i runs through the simple objects, d_i is the quantum dimension, and θ_i is the twist. This sum with θ_i is denoted p_- in [BaKi2001] Chapter 3.

EXAMPLES:

```
sage: FusionDouble(DihedralGroup(7)).total_q_order()
14
```

```
>>> from sage.all import *
>>> FusionDouble(DihedralGroup(Integer(7))).total_q_order()
14
```

5.8.4 F-Matrix Backend

Fast F-Matrix Methods

`sage.algebras.fusion_rings.fast_parallel_fmats_methods.executor(params)`

Execute a function defined in this module (`sage.algebras.fusion_rings.`

`fast_parallel_fmats_methods`) in a worker process, and supply the factory parameter by constructing a reference to the `FMatrix` object in the worker's memory address space from its `id`.

INPUT:

- `params` – tuple `((fn_name, fmats_id), fn_args)` where `fn_name` is the name of the function to be executed, `fmats_id` is the `id` of the `FMatrix` object, and `fn_args` is a tuple containing all arguments to be passed to the function `fn_name`.

Note

When the parent process is forked, each worker gets a copy of every global variable. The virtual memory address of object X in the parent process equals the *virtual* memory address of the copy of object X in each worker, so we may construct references to forked copies of X using an `id` obtained in the parent process.

Fast Fusion Ring Methods for Computing Braid Group Representations

`sage.algebras.fusion_rings.fast_parallel_fusion_ring_braid_repn.executor(params)`

Execute a function registered in this module's `mappers` in a worker process, and supply the `FusionRing` parameter by constructing a reference to the `FMatrix` object in the worker's memory address space from its `id`.

Note

When the parent process is forked, each worker gets a copy of every global variable. The virtual memory address of object X in the parent process equals the *virtual* memory address of the copy of object X in each worker, so we may construct references to forked copies of X .

Arithmetic Engine for Polynomials as Tuples

`sage.algebras.fusion_rings.poly_tup_engine.apply_coeff_map(eq_tup, coeff_map)`

Apply `coeff_map` to coefficients.

EXAMPLES:

```
sage: from sage.algebras.fusion_rings.poly_tup_engine import apply_coeff_map
sage: sq = lambda x : x**2
sage: R.<x, y, z> = PolynomialRing(ZZ)
sage: from sage.algebras.fusion_rings.poly_tup_engine import poly_to_tup, _tup_to_
      ↪poly
sage: _tup_to_poly(apply_coeff_map(poly_to_tup(x + 2*y + 3*z), sq), parent=R)
x + 4*y + 9*z
```

```
>>> from sage.all import *
>>> from sage.algebras.fusion_rings.poly_tup_engine import apply_coeff_map
>>> sq = lambda x : x**Integer(2)
>>> R = PolynomialRing(ZZ, names=('x', 'y', 'z')); (x, y, z) = R._first_ngens(3)
>>> from sage.algebras.fusion_rings.poly_tup_engine import poly_to_tup, _tup_to_
      ↪poly
>>> _tup_to_poly(apply_coeff_map(poly_to_tup(x + Integer(2)*y + Integer(3)*z), ↪
      ↪sq), parent=R)
x + 4*y + 9*z
```

`sage.algebras.fusion_rings.poly_tup_engine.compute_known_powers(max_degs, val_dict, one)`

Pre-compute powers of known values for efficiency when preparing to substitute into a list of polynomials.

INPUT:

- `max_deg` – an ETuple indicating the maximal degree of each variable
- `val_dict` – dictionary of (var_idx, poly_tup) key-value pairs
- `poly_tup` – tuple of (ETuple, coeff) pairs representing a multivariate polynomial

EXAMPLES:

```
sage: from sage.algebras.fusion_rings.poly_tup_engine import compute_known_powers
sage: R.<x, y, z> = PolynomialRing(QQ)
sage: polys = [x**3 + 1, x**2*y + z**3, y**2 - 3*y]
sage: from sage.algebras.fusion_rings.poly_tup_engine import poly_to_tup
sage: known_val = { 0 : poly_to_tup(R(-1)), 2 : poly_to_tup(y**2) }
sage: from sage.algebras.fusion_rings.poly_tup_engine import get_variables_degrees
sage: max_deg = get_variables_degrees([poly_to_tup(p) for p in polys], 3)
sage: compute_known_powers(max_deg, known_val, R.base_ring().one())
{0: [(((0, 0, 0), 1)),
      (((0, 0, 0), -1)),
      (((0, 0, 0), 1)),
      (((0, 0, 0), -1))],
 2: [(((0, 0, 0), 1)),
      (((0, 2, 0), 1)),
      (((0, 4, 0), 1)),
      (((0, 6, 0), 1))}]
```

```
>>> from sage.all import *
>>> from sage.algebras.fusion_rings.poly_tup_engine import compute_known_powers
>>> R = PolynomialRing(QQ, names=('x', 'y', 'z')); (x, y, z) = R._first_ngens(3)
>>> polys = [x**Integer(3) + Integer(1), x**Integer(2)*y + z**Integer(3),
↳ y**Integer(2) - Integer(3)*y]
>>> from sage.algebras.fusion_rings.poly_tup_engine import poly_to_tup
>>> known_val = { Integer(0) : poly_to_tup(R(-Integer(1))), Integer(2) : poly_to_
↳ tup(y**Integer(2)) }
>>> from sage.algebras.fusion_rings.poly_tup_engine import get_variables_degrees
>>> max_deg = get_variables_degrees([poly_to_tup(p) for p in polys], Integer(3))
>>> compute_known_powers(max_deg, known_val, R.base_ring().one())
{0: [(((0, 0, 0), 1)),
      (((0, 0, 0), -1)),
      (((0, 0, 0), 1)),
      (((0, 0, 0), -1))],
 2: [(((0, 0, 0), 1)),
      (((0, 2, 0), 1)),
      (((0, 4, 0), 1)),
      (((0, 6, 0), 1))}]
```

`sage.algebras.fusion_rings.poly_tup_engine.constant_coeff(eq_tup, field)`

Return the constant coefficient of the polynomial represented by given tuple.

EXAMPLES:

```

sage: from sage.algebras.fusion_rings.poly_tup_engine import constant_coeff
sage: from sage.rings.polynomial.polydict import ETuple
sage: poly_tup = ((ETuple([0, 3, 0]), 2), (ETuple([0, 1, 0]), -1), (ETuple([0, 0, 0]), -2/3))
sage: constant_coeff(poly_tup, QQ)
-2/3
sage: R.<x, y, z> = PolynomialRing(QQ)
sage: from sage.algebras.fusion_rings.poly_tup_engine import poly_to_tup
sage: constant_coeff(poly_to_tup(x**5 + x*y*z - 9), QQ)
-9

```

```

>>> from sage.all import *
>>> from sage.algebras.fusion_rings.poly_tup_engine import constant_coeff
>>> from sage.rings.polynomial.polydict import ETuple
>>> poly_tup = ((ETuple([Integer(0), Integer(3), Integer(0)]), Integer(2)),
→ (ETuple([Integer(0), Integer(1), Integer(0)]), -Integer(1)),
→ (ETuple([Integer(0), Integer(0), Integer(0)]), -Integer(2)/Integer(3)))
>>> constant_coeff(poly_tup, QQ)
-2/3
>>> R = PolynomialRing(QQ, names=('x', 'y', 'z')); (x, y, z) = R._first_ngens(3)
>>> from sage.algebras.fusion_rings.poly_tup_engine import poly_to_tup
>>> constant_coeff(poly_to_tup(x**Integer(5) + x*y*z - Integer(9)), QQ)
-9

```

`sage.algebras.fusion_rings.poly_tup_engine.get_variables_degrees` (*eqns*, *nvars*)

Find maximum degrees for each variable in equations.

EXAMPLES:

```

sage: from sage.algebras.fusion_rings.poly_tup_engine import get_variables_degrees
sage: R.<x, y, z> = PolynomialRing(QQ)
sage: polys = [x**2 + 1, x*y*z**2 - 4*x*y, x*z**3 - 4/3*y + 1]
sage: from sage.algebras.fusion_rings.poly_tup_engine import poly_to_tup
sage: get_variables_degrees([poly_to_tup(p) for p in polys], 3)
[2, 1, 3]

```

```

>>> from sage.all import *
>>> from sage.algebras.fusion_rings.poly_tup_engine import get_variables_degrees
>>> R = PolynomialRing(QQ, names=('x', 'y', 'z')); (x, y, z) = R._first_ngens(3)
>>> polys = [x**Integer(2) + Integer(1), x*y*z**Integer(2) - Integer(4)*x*y,
→ x*z**Integer(3) - Integer(4)/Integer(3)*y + Integer(1)]
>>> from sage.algebras.fusion_rings.poly_tup_engine import poly_to_tup
>>> get_variables_degrees([poly_to_tup(p) for p in polys], Integer(3))
[2, 1, 3]

```

`sage.algebras.fusion_rings.poly_tup_engine.poly_to_tup` (*poly*)

Convert a polynomial object into the internal representation as tuple of (ETuple exp, NumberFieldElement coeff) pairs.

EXAMPLES:

```

sage: from sage.algebras.fusion_rings.poly_tup_engine import poly_to_tup
sage: R.<x, y> = PolynomialRing(QQ)

```

(continues on next page)

(continued from previous page)

```
sage: poly_to_tup(x**2 + 1)
((2, 0), 1), ((0, 0), 1)
sage: poly_to_tup(x**2*y**4 - 4/5*x*y**2 + 1/3 * y)
((2, 4), 1), ((1, 2), -4/5), ((0, 1), 1/3))
```

```
>>> from sage.all import *
>>> from sage.algebras.fusion_rings.poly_tup_engine import poly_to_tup
>>> R = PolynomialRing(QQ, names=('x', 'y',)); (x, y,) = R._first_ngens(2)
>>> poly_to_tup(x**Integer(2) + Integer(1))
((2, 0), 1), ((0, 0), 1)
>>> poly_to_tup(x**Integer(2)*y**Integer(4) - Integer(4)/
↳ Integer(5)*x*y**Integer(2) + Integer(1)/Integer(3) * y)
((2, 4), 1), ((1, 2), -4/5), ((0, 1), 1/3))
```

sage.algebras.fusion_rings.poly_tup_engine.**poly_tup_sortkey**(*eq_tup*)

Return the sortkey of a polynomial represented as a tuple of (ETuple, coeff) pairs with respect to the degree lexicographical term order.

Using this key to sort polynomial tuples results in comparing polynomials term by term (we assume the tuple representation is sorted so that the leading term with respect to the degree reverse lexicographical order comes first). For each term, we first compare degrees, then the monomials themselves. Different polynomials can have the same sortkey.

EXAMPLES:

```
sage: F = CyclotomicField(20)
sage: zeta20 = F.gen()
sage: R.<x, y, z> = PolynomialRing(F)
sage: from sage.algebras.fusion_rings.poly_tup_engine import poly_tup_sortkey,
↳ poly_to_tup
sage: p = (zeta20 + 1)*x^2 + (zeta20^3 + 6)*x*z + (zeta20^2 + 7*zeta20)*z^2 + (2/
↳ 3*zeta20 + 1/4)*x + y
sage: p1 = poly_to_tup(p); p1
((2, 0, 0), zeta20 + 1),
((1, 0, 1), zeta20^3 + 6),
((0, 0, 2), zeta20^2 + 7*zeta20),
((1, 0, 0), 2/3*zeta20 + 1/4),
((0, 1, 0), 1))
sage: poly_tup_sortkey(p1)
(2, 0, 2, 2, 0, 1, -2, 1, 2, -2, 2, 1, 0, 1, 1, -1, 1)
```

```
>>> from sage.all import *
>>> F = CyclotomicField(Integer(20))
>>> zeta20 = F.gen()
>>> R = PolynomialRing(F, names=('x', 'y', 'z',)); (x, y, z,) = R._first_ngens(3)
>>> from sage.algebras.fusion_rings.poly_tup_engine import poly_tup_sortkey, poly_
↳ to_tup
>>> p = (zeta20 + Integer(1))*x**Integer(2) + (zeta20**Integer(3) +
↳ Integer(6))*x*z + (zeta20**Integer(2) + Integer(7)*zeta20)*z**Integer(2) +
↳ (Integer(2)/Integer(3)*zeta20 + Integer(1)/Integer(4))*x + y
>>> p1 = poly_to_tup(p); p1
((2, 0, 0), zeta20 + 1),
```

(continues on next page)

(continued from previous page)

```

((1, 0, 1), zeta20^3 + 6),
((0, 0, 2), zeta20^2 + 7*zeta20),
((1, 0, 0), 2/3*zeta20 + 1/4),
((0, 1, 0), 1))
>>> poly_tup_sortkey(p1)
(2, 0, 2, 2, 0, 1, -2, 1, 2, -2, 2, 1, 0, 1, 1, -1, 1)

```

sage.algebras.fusion_rings.poly_tup_engine.**resize**(eq_tup, idx_map, nvars)

Return a tuple representing a polynomial in a ring with `len(sorted_vars)` generators.

This method is used for creating polynomial objects with the “right number” of variables for computing Groebner bases of the partitioned equations graph and for adding constraints ensuring certain F-symbols are nonzero.

EXAMPLES:

```

sage: from sage.algebras.fusion_rings.poly_tup_engine import resize
sage: from sage.rings.polynomial.polydict import ETuple
sage: K = CyclotomicField(56)
sage: poly_tup = ((ETuple([0, 3, 0, 2]), K(2)), (ETuple([0, 1, 0, 1]), K(-1)),
↳ (ETuple([0, 0, 0, 0]), K(-2/3)))
sage: idx_map = {1: 0, 3: 1}
sage: resize(poly_tup, idx_map, 2)
((3, 2), 2), ((1, 1), -1), ((0, 0), -2/3))

sage: R = PolynomialRing(K, 'fx', 20)
sage: R.inject_variables()
Defining fx0, fx1, fx2, fx3, fx4, fx5, fx6, fx7, fx8, fx9, fx10, fx11, fx12, fx13,
↳ fx14, fx15, fx16, fx17, fx18, fx19
sage: sparse_poly = R(fx0**2 * fx17 + fx3)
sage: from sage.algebras.fusion_rings.poly_tup_engine import poly_to_tup, _tup_to_
↳ poly
sage: S.<x, y, z> = PolynomialRing(K)
sage: _tup_to_poly(resize(poly_to_tup(sparse_poly), {0:0, 3:1, 17:2}, 3),
↳ parent=S)
x^2*z + y

```

```

>>> from sage.all import *
>>> from sage.algebras.fusion_rings.poly_tup_engine import resize
>>> from sage.rings.polynomial.polydict import ETuple
>>> K = CyclotomicField(Integer(56))
>>> poly_tup = ((ETuple([Integer(0), Integer(3), Integer(0), Integer(2)]),
↳ K(Integer(2))), (ETuple([Integer(0), Integer(1), Integer(0), Integer(1)]), K(-
↳ Integer(1))), (ETuple([Integer(0), Integer(0), Integer(0), Integer(0)]), K(-
↳ Integer(2)/Integer(3))))
>>> idx_map = {Integer(1): Integer(0), Integer(3): Integer(1)}
>>> resize(poly_tup, idx_map, Integer(2))
((3, 2), 2), ((1, 1), -1), ((0, 0), -2/3))

>>> R = PolynomialRing(K, 'fx', Integer(20))
>>> R.inject_variables()
Defining fx0, fx1, fx2, fx3, fx4, fx5, fx6, fx7, fx8, fx9, fx10, fx11, fx12, fx13,
↳ fx14, fx15, fx16, fx17, fx18, fx19
>>> sparse_poly = R(fx0**Integer(2) * fx17 + fx3)

```

(continues on next page)

(continued from previous page)

```

>>> from sage.algebras.fusion_rings.poly_tup_engine import poly_to_tup, _tup_to_
↳poly
>>> S = PolynomialRing(K, names=('x', 'y', 'z')); (x, y, z,) = S._first_ngens(3)
>>> _tup_to_poly(resize(poly_to_tup(sparse_poly), {Integer(0):Integer(0),
↳Integer(3):Integer(1), Integer(17):Integer(2)}, Integer(3)), parent=S)
x^2*z + y

```

sage.algebras.fusion_rings.poly_tup_engine.**tup_to_univ_poly**(*eq_tup*, *univ_poly_ring*)

Given a tuple of pairs representing a univariate polynomial and a univariate polynomial ring, return a univariate polynomial object.

Each pair in the tuple is assumed to be of the form (ETuple, coeff), where coeff is an element of univ_poly_ring.base_ring().

EXAMPLES:

```

sage: from sage.algebras.fusion_rings.poly_tup_engine import tup_to_univ_poly
sage: from sage.rings.polynomial.polydict import ETuple
sage: K = CyclotomicField(56)
sage: poly_tup = ((ETuple([0, 3, 0]), K(2)), (ETuple([0, 1, 0]), K(-1)),
↳(ETuple([0, 0, 0]), K(-2/3)))
sage: R = K['b']
sage: tup_to_univ_poly(poly_tup, R)
2*b^3 - b - 2/3

```

```

>>> from sage.all import *
>>> from sage.algebras.fusion_rings.poly_tup_engine import tup_to_univ_poly
>>> from sage.rings.polynomial.polydict import ETuple
>>> K = CyclotomicField(Integer(56))
>>> poly_tup = ((ETuple([Integer(0), Integer(3), Integer(0)]), K(Integer(2))),
↳(ETuple([Integer(0), Integer(1), Integer(0)]), K(-Integer(1))),
↳(ETuple([Integer(0), Integer(0), Integer(0)]), K(-Integer(2)/Integer(3))))
>>> R = K['b']
>>> tup_to_univ_poly(poly_tup, R)
2*b^3 - b - 2/3

```

sage.algebras.fusion_rings.poly_tup_engine.**variables**(*eq_tup*)

Return indices of all variables appearing in eq_tup.

EXAMPLES:

```

sage: from sage.algebras.fusion_rings.poly_tup_engine import variables
sage: from sage.rings.polynomial.polydict import ETuple
sage: poly_tup = ((ETuple([0, 3, 0]), 2), (ETuple([0, 1, 0]), -1), (ETuple([0, 0,
↳0]), -2/3))
sage: variables(poly_tup)
[1]
sage: from sage.algebras.fusion_rings.poly_tup_engine import poly_to_tup
sage: R.<x, y, z> = PolynomialRing(QQ)
sage: variables(poly_to_tup(x^2*y + y**3 - 4/3*x))
[0, 1]
sage: variables(poly_to_tup(R(1/4)))
[]

```

```

>>> from sage.all import *
>>> from sage.algebras.fusion_rings.poly_tup_engine import variables
>>> from sage.rings.polynomial.polydict import ETuple
>>> poly_tup = ((ETuple([Integer(0), Integer(3), Integer(0)]), Integer(2)),
↳ (ETuple([Integer(0), Integer(1), Integer(0)]), -Integer(1)),
↳ (ETuple([Integer(0), Integer(0), Integer(0)]), -Integer(2)/Integer(3)))
>>> variables(poly_tup)
[1]
>>> from sage.algebras.fusion_rings.poly_tup_engine import poly_to_tup
>>> R = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = R._first_ngens(3)
>>> variables(poly_to_tup(x*Integer(2)*y + y**Integer(3) - Integer(4)/
↳ Integer(3)*x))
[0, 1]
>>> variables(poly_to_tup(R(Integer(1)/Integer(4))))
[]

```

Shared Memory Managers for F-Symbol Attributes

This module provides an implementation for shared dictionary like state attributes required by the orthogonal F-matrix solver.

Currently, the attributes only work when the base field of the `FMatrix` factory is a cyclotomic field.

class `sage.algebras.fusion_rings.shm_managers.FvarsHandler`

Bases: `object`

A shared memory backed dict-like structure to manage the `_fvars` attribute of an F-matrix.

This structure implements a representation of the F-symbols dictionary using a structured NumPy array backed by a contiguous shared memory object.

The monomial data is stored in the `exp_data` structure. Monomial exponent data is stored contiguously and ticks are used to indicate different monomials.

Coefficient data is stored in the `coeff_nums` and `coeff_denom` arrays. The `coeff_denom` array stores the value `d = coeff.denominator()` for each cyclotomic coefficient. The `coeff_nums` array stores the values `c.numerator() * d` for `c` in `coeff._coefficients()`, the abridged list representation of the cyclotomic coefficient `coeff`.

Each entry also has a boolean `modified` attribute, indicating whether it has been modified by the parent process. Entry retrieval is cached in each process, so each process must check whether entries have been modified before attempting retrieval.

The parent process should construct this object without a `name` attribute. Children processes use the `name` attribute, accessed via `self.shm.name` to attach to the shared memory block.

INPUT:

- `n_slots` – number of generators of the underlying polynomial ring
- `field` – base field for polynomial ring
- `idx_to_sextuple` – map relating a single integer index to a sextuple of `FusionRing` elements
- `init_data` – dictionary or `FvarsHandler` object containing known squares for initialization, e.g., from a solver checkpoint
- `use_mp` – integer indicating the number of child processes used for multiprocessing; if running serially, use 0

- `pids_name` – the name of a `ShareableList` containing the process `pid`'s for every process in the pool (including the parent process)
- `name` – the name of a shared memory object (used by child processes for attaching)
- `max_terms` – maximum number of terms in each entry; since we use contiguous C-style memory blocks, the size of the block must be known in advance
- `n_bytes` – the number of bytes that should be allocated for each numerator and each denominator stored by the structure

Note

To properly dispose of shared memory resources, `self.shm.unlink()` must be called before exiting.

Note

If you ever encounter an `OverflowError` when running the `FMatrix.find_orthogonal_solution()` solver, consider increasing the parameter `n_bytes`.

Warning

The current data structure supports up to 2^{16} entries, with each monomial in each entry having at most 254 nonzero terms. On average, each of the `max_terms` monomials can have at most 30 terms.

EXAMPLES:

```
sage: from sage.algebras.fusion_rings.shm_managers import FvarsHandler
sage: # Create shared data structure
sage: f = FusionRing("A2", 1).get_fmatrix(inject_variables=True, new=True)
creating variables fx1..fx8
Defining fx0, fx1, fx2, fx3, fx4, fx5, fx6, fx7
sage: f.start_worker_pool()
sage: n_proc = f.pool._processes
sage: pids_name = f._pid_list.shm.name
sage: fvars = FvarsHandler(8, f._field, f._idx_to_sextuple, use_mp=n_proc, pids_
↪name=pids_name)
sage: # In the same shell or in a different shell, attach to fvars
sage: name = fvars.shm.name
sage: fvars2 = FvarsHandler(8, f._field, f._idx_to_sextuple, name=name, use_mp=n_
↪proc, pids_name=pids_name)
sage: from sage.algebras.fusion_rings.poly_tup_engine import poly_to_tup
sage: rhs = tuple((exp, tuple(c._coefficients())) for exp, c in poly_to_
↪tup(fx5**5))
sage: fvars[f2, f1, f2, f2, f0, f0] = rhs
sage: f._tup_to_fpoly(fvars2[f2, f1, f2, f2, f0, f0])
fx5^5
sage: fvars.shm.unlink()
sage: f.shutdown_worker_pool()
```

```

>>> from sage.all import *
>>> from sage.algebras.fusion_rings.shm_managers import FvarsHandler
>>> # Create shared data structure
>>> f = FusionRing("A2", Integer(1)).get_fmatrix(inject_variables=True, new=True)
creating variables fx1..fx8
Defining fx0, fx1, fx2, fx3, fx4, fx5, fx6, fx7
>>> f.start_worker_pool()
>>> n_proc = f.pool._processes
>>> pids_name = f._pid_list.shm.name
>>> fvars = FvarsHandler(Integer(8), f._field, f._idx_to_sextuple, use_mp=n_proc,
↳pids_name=pids_name)
>>> # In the same shell or in a different shell, attach to fvars
>>> name = fvars.shm.name
>>> fvars2 = FvarsHandler(Integer(8), f._field, f._idx_to_sextuple, name=name,
↳use_mp=n_proc, pids_name=pids_name)
>>> from sage.algebras.fusion_rings.poly_tup_engine import poly_to_tup
>>> rhs = tuple((exp, tuple(c._coefficients())) for exp, c in poly_to_
↳tup(fx5**Integer(5)))
>>> fvars[f2, f1, f2, f2, f0, f0] = rhs
>>> f._tup_to_fpoly(fvars2[f2, f1, f2, f2, f0, f0])
fx5^5
>>> fvars.shm.unlink()
>>> f.shutdown_worker_pool()

```

items()

Iterate through key-value pairs in the data structure as if it were a Python dict.

As in a Python dict, the key-value pairs are yielded in no particular order.

EXAMPLES:

```

sage: f = FusionRing("G2", 1).get_fmatrix(inject_variables=True, new=True)
creating variables fx1..fx5
Defining fx0, fx1, fx2, fx3, fx4
sage: from sage.algebras.fusion_rings.shm_managers import FvarsHandler
sage: shared_fvars = FvarsHandler(5, f._field, f._idx_to_sextuple, init_
↳data=f._fvars)
sage: for sextuple, fvar in shared_fvars.items():
.....:     if sextuple == (f1, f1, f1, f1, f1, f1):
.....:         f._tup_to_fpoly(fvar)
.....:
fx4

```

```

>>> from sage.all import *
>>> f = FusionRing("G2", Integer(1)).get_fmatrix(inject_variables=True,
↳new=True)
creating variables fx1..fx5
Defining fx0, fx1, fx2, fx3, fx4
>>> from sage.algebras.fusion_rings.shm_managers import FvarsHandler
>>> shared_fvars = FvarsHandler(Integer(5), f._field, f._idx_to_sextuple,
↳init_data=f._fvars)
>>> for sextuple, fvar in shared_fvars.items():
...     if sextuple == (f1, f1, f1, f1, f1, f1):

```

(continues on next page)

(continued from previous page)

```
...      f._tup_to_fpoly(fvar)
.....:
fx4
```

shm**class** sage.algebras.fusion_rings.shm_managers.**KSHandler**

Bases: object

A shared memory backed dict-like structure to manage the `_ks` attribute of an F-matrix.

This structure implements a representation of the known squares dictionary using a structured NumPy array backed by a contiguous shared memory object.

The structure mimics a dictionary of `(idx, known_sq)` pairs. Each integer index corresponds to a variable and each `known_sq` is an element of the F-matrix factory's base cyclotomic field.

Each cyclotomic coefficient is stored as a list of numerators and a list of denominators representing the rational coefficients. The structured array also maintains `known` attribute that indicates whether the structure contains an entry corresponding to the given index.

The parent process should construct this object without a `name` attribute. Children processes use the `name` attribute, accessed via `self.shm.name` to attach to the shared memory block.

INPUT:

- `n_slots` – the total number of F-symbols
- `field` – F-matrix's base cyclotomic field
- `use_mp` – boolean indicating whether to construct a shared memory block to back `self`
- `init_data` – dictionary or *KSHandler* object containing known squares for initialization, e.g., from a solver checkpoint
- **name** – the name of a shared memory object (used by child processes for attaching)

Note

To properly dispose of shared memory resources, `self.shm.unlink()` must be called before exiting.

Warning

This structure *cannot* modify an entry that has already been set.

EXAMPLES:

```
sage: from sage.algebras.fusion_rings.shm_managers import KSHandler
sage: # Create shared data structure
sage: f = FusionRing("A1", 2).get_fmatrix(inject_variables=True, new=True)
creating variables fx1..fx14
Defining fx0, fx1, fx2, fx3, fx4, fx5, fx6, fx7, fx8, fx9, fx10, fx11, fx12, fx13
sage: n = f._poly_ring.ngens()
sage: f.start_worker_pool()
```

(continues on next page)

(continued from previous page)

```

sage: ks = KSHandler(n, f._field, use_mp=True)
sage: # In the same shell or in a different shell, attach to fvars
sage: name = ks.shm.name
sage: ks2 = KSHandler(n, f._field, name=name, use_mp=True)
sage: from sage.algebras.fusion_rings.poly_tup_engine import poly_to_tup
sage: eqns = [fx1**2 - 4, fx3**2 + f._field.gen()**4 - 1/19*f._field.gen()**2]
sage: ks.update([poly_to_tup(p) for p in eqns])
sage: for idx, sq in ks.items():
.....:     print("Index: {}, square: {}".format(idx, sq))
.....:
Index: 1, square: 4
Index: 3, square: -zeta32^4 + 1/19*zeta32^2
sage: ks.shm.unlink()
sage: f.shutdown_worker_pool()

```

```

>>> from sage.all import *
>>> from sage.algebras.fusion_rings.shm_managers import KSHandler
>>> # Create shared data structure
>>> f = FusionRing("A1", Integer(2)).get_fmatrix(inject_variables=True, new=True)
creating variables fx1..fx14
Defining fx0, fx1, fx2, fx3, fx4, fx5, fx6, fx7, fx8, fx9, fx10, fx11, fx12, fx13
>>> n = f._poly_ring.ngens()
>>> f.start_worker_pool()
>>> ks = KSHandler(n, f._field, use_mp=True)
>>> # In the same shell or in a different shell, attach to fvars
>>> name = ks.shm.name
>>> ks2 = KSHandler(n, f._field, name=name, use_mp=True)
>>> from sage.algebras.fusion_rings.poly_tup_engine import poly_to_tup
>>> eqns = [fx1**Integer(2) - Integer(4), fx3**Integer(2) + f._field.
↳ gen()**Integer(4) - Integer(1)/Integer(19)*f._field.gen()**Integer(2)]
>>> ks.update([poly_to_tup(p) for p in eqns])
>>> for idx, sq in ks.items():
...     print("Index: {}, square: {}".format(idx, sq))
.....:
Index: 1, square: 4
Index: 3, square: -zeta32^4 + 1/19*zeta32^2
>>> ks.shm.unlink()
>>> f.shutdown_worker_pool()

```

items()

Iterate through existing entries using Python dict-style syntax.

EXAMPLES:

```

sage: f = FusionRing("A3", 1).get_fmatrix()
sage: f._reset_solver_state()
sage: f.get_orthogonality_constraints(output=False)
sage: f._ks.update(f.ideal_basis)
sage: for idx, sq in f._ks.items():
.....:     print("Index: {}, sq: {}".format(idx, sq))
.....:
Index: 0, sq: 1

```

(continues on next page)

(continued from previous page)

```

Index: 1, sq: 1
Index: 2, sq: 1
Index: 3, sq: 1
Index: 4, sq: 1
...
Index: 25, sq: 1
Index: 26, sq: 1

```

```

>>> from sage.all import *
>>> f = FusionRing("A3", Integer(1)).get_fmatrix()
>>> f._reset_solver_state()
>>> f.get_orthogonality_constraints(output=False)
>>> f._ks.update(f.ideal_basis)
>>> for idx, sq in f._ks.items():
...     print("Index: {}, sq: {}".format(idx, sq))
.....:
Index: 0, sq: 1
Index: 1, sq: 1
Index: 2, sq: 1
Index: 3, sq: 1
Index: 4, sq: 1
...
Index: 25, sq: 1
Index: 26, sq: 1

```

shm**update** (*eqns*)

Update self's shared_memory-backed dictionary of known squares. Keys are variable indices and corresponding values are the squares.

EXAMPLES:

```

sage: f = FusionRing("B5", 1).get_fmatrix()
sage: f._reset_solver_state()
sage: for idx, sq in f._ks.items():
.....:     k
.....:
sage: f.get_orthogonality_constraints()
[fx0^2 - 1,
 fx1^2 - 1,
 fx2^2 - 1,
 fx3^2 - 1,
 fx4^2 - 1,
 fx5^2 - 1,
 fx6^2 - 1,
 fx7^2 - 1,
 fx8^2 - 1,
 fx9^2 - 1,
 fx10^2 + fx12^2 - 1,
 fx10*fx11 + fx12*fx13,
 fx10*fx11 + fx12*fx13,

```

(continues on next page)

(continued from previous page)

```

fx11^2 + fx13^2 - 1]
sage: f.get_orthogonality_constraints(output=False)
sage: f._ks.update(f.ideal_basis)
sage: for idx, sq in f._ks.items():
....:     print(idx, "-->", sq)
....:
0 --> 1
1 --> 1
2 --> 1
3 --> 1
4 --> 1
5 --> 1
6 --> 1
7 --> 1
8 --> 1
9 --> 1

```

```

>>> from sage.all import *
>>> f = FusionRing("B5", Integer(1)).get_fmatrix()
>>> f._reset_solver_state()
>>> for idx, sq in f._ks.items():
...     k
....:
>>> f.get_orthogonality_constraints()
[fx0^2 - 1,
 fx1^2 - 1,
 fx2^2 - 1,
 fx3^2 - 1,
 fx4^2 - 1,
 fx5^2 - 1,
 fx6^2 - 1,
 fx7^2 - 1,
 fx8^2 - 1,
 fx9^2 - 1,
 fx10^2 + fx12^2 - 1,
 fx10*fx11 + fx12*fx13,
 fx10*fx11 + fx12*fx13,
 fx11^2 + fx13^2 - 1]
>>> f.get_orthogonality_constraints(output=False)
>>> f._ks.update(f.ideal_basis)
>>> for idx, sq in f._ks.items():
...     print(idx, "-->", sq)
....:
0 --> 1
1 --> 1
2 --> 1
3 --> 1
4 --> 1
5 --> 1
6 --> 1
7 --> 1
8 --> 1

```

(continues on next page)

(continued from previous page)

9 --> 1

WarningThis method assumes every polynomial in eqns is *monic*.

```
sage.algebras.fusion_rings.shm_managers.make_FvarsHandler(n, field, idx_map, init_data)
```

Provide pickling / unpickling support for `FvarsHandler`.

```
sage.algebras.fusion_rings.shm_managers.make_KSHandler(n_slots, field, init_data)
```

Provide pickling / unpickling support for `KSHandler`.

5.9 Hall Algebras

AUTHORS:

- Travis Scrimshaw (2013-10-17): Initial version

```
class sage.algebras.hall_algebra.HallAlgebra(base_ring, q, prefix='H')
```

Bases: `CombinatorialFreeModule`

The (classical) Hall algebra.

The (classical) Hall algebra over a commutative ring R with a parameter $q \in R$ is defined to be the free R -module with basis (I_λ) , where λ runs over all integer partitions. The algebra structure is given by a product defined by

$$I_\mu \cdot I_\lambda = \sum_{\nu} P_{\mu,\lambda}^{\nu}(q) I_{\nu},$$

where $P_{\mu,\lambda}^{\nu}$ is a Hall polynomial (see `hall_polynomial()`). The unity of this algebra is $I_\emptyset$.

The (classical) Hall algebra is also known as the Hall-Steinitz algebra.

We can define an R -algebra isomorphism Φ from the R -algebra of symmetric functions (see `SymmetricFunctions`) to the (classical) Hall algebra by sending the r -th elementary symmetric function e_r to $q^{r(r-1)/2} I_{(1^r)}$ for every positive integer r . This isomorphism used to transport the Hopf algebra structure from the R -algebra of symmetric functions to the Hall algebra, thus making the latter a connected graded Hopf algebra. If λ is a partition, then the preimage of the basis element I_λ under this isomorphism is $q^{n(\lambda)} P_\lambda(x; q^{-1})$, where P_λ denotes the λ -th Hall-Littlewood P -function, and where $n(\lambda) = \sum_i (i-1)\lambda_i$.

See section 2.3 in [Sch2006], and sections II.2 and III.3 in [Mac1995] (where our I_λ is called u_λ).

EXAMPLES:

```
sage: R.<q> = ZZ[]
sage: H = HallAlgebra(R, q)
sage: H[2,1]*H[1,1]
H[3, 2] + (q+1)*H[3, 1, 1] + (q^2+q)*H[2, 2, 1] + (q^4+q^3+q^2)*H[2, 1, 1, 1]
sage: H[2]*H[2,1]
H[4, 1] + q*H[3, 2] + (q^2-1)*H[3, 1, 1] + (q^3+q^2)*H[2, 2, 1]
sage: H[3]*H[1,1]
H[4, 1] + q^2*H[3, 1, 1]
sage: H[3]*H[2,1]
H[5, 1] + q*H[4, 2] + (q^2-1)*H[4, 1, 1] + q^3*H[3, 2, 1]
```

```

>>> from sage.all import *
>>> R = ZZ['q']; (q,) = R._first_ngens(1)
>>> H = HallAlgebra(R, q)
>>> H[Integer(2), Integer(1)]*H[Integer(1), Integer(1)]
H[3, 2] + (q+1)*H[3, 1, 1] + (q^2+q)*H[2, 2, 1] + (q^4+q^3+q^2)*H[2, 1, 1, 1]
>>> H[Integer(2)]*H[Integer(2), Integer(1)]
H[4, 1] + q*H[3, 2] + (q^2-1)*H[3, 1, 1] + (q^3+q^2)*H[2, 2, 1]
>>> H[Integer(3)]*H[Integer(1), Integer(1)]
H[4, 1] + q^2*H[3, 1, 1]
>>> H[Integer(3)]*H[Integer(2), Integer(1)]
H[5, 1] + q*H[4, 2] + (q^2-1)*H[4, 1, 1] + q^3*H[3, 2, 1]

```

We can rewrite the Hall algebra in terms of monomials of the elements $I_{(1^r)}$:

```

sage: I = H.monomial_basis()
sage: H(I[2,1,1])
H[3, 1] + (q+1)*H[2, 2] + (2*q^2+2*q+1)*H[2, 1, 1]
+ (q^5+2*q^4+3*q^3+3*q^2+2*q+1)*H[1, 1, 1, 1]
sage: I(H[2,1,1])
I[3, 1] + (-q^3-q^2-q-1)*I[4]

```

```

>>> from sage.all import *
>>> I = H.monomial_basis()
>>> H(I[Integer(2), Integer(1), Integer(1)])
H[3, 1] + (q+1)*H[2, 2] + (2*q^2+2*q+1)*H[2, 1, 1]
+ (q^5+2*q^4+3*q^3+3*q^2+2*q+1)*H[1, 1, 1, 1]
>>> I(H[Integer(2), Integer(1), Integer(1)])
I[3, 1] + (-q^3-q^2-q-1)*I[4]

```

The isomorphism between the Hall algebra and the symmetric functions described above is implemented as a coercion:

```

sage: R = PolynomialRing(ZZ, 'q').fraction_field()
sage: q = R.gen()
sage: H = HallAlgebra(R, q)
sage: e = SymmetricFunctions(R).e()
sage: e(H[1,1,1])
1/q^3*e[3]

```

```

>>> from sage.all import *
>>> R = PolynomialRing(ZZ, 'q').fraction_field()
>>> q = R.gen()
>>> H = HallAlgebra(R, q)
>>> e = SymmetricFunctions(R).e()
>>> e(H[Integer(1), Integer(1), Integer(1)])
1/q^3*e[3]

```

We can also do computations with any special value of q , such as 0 or 1 or (most commonly) a prime power. Here is an example using a prime:

```

sage: H = HallAlgebra(ZZ, 2)
sage: H[2,1]*H[1,1]
H[3, 2] + 3*H[3, 1, 1] + 6*H[2, 2, 1] + 28*H[2, 1, 1, 1]

```

(continues on next page)

(continued from previous page)

```

sage: H[3,1]*H[2]
H[5, 1] + H[4, 2] + 6*H[3, 3] + 3*H[4, 1, 1] + 8*H[3, 2, 1]
sage: H[2,1,1]*H[3,1]
H[5, 2, 1] + 2*H[4, 3, 1] + 6*H[4, 2, 2] + 7*H[5, 1, 1, 1]
+ 19*H[4, 2, 1, 1] + 24*H[3, 3, 1, 1] + 48*H[3, 2, 2, 1]
+ 105*H[4, 1, 1, 1, 1] + 224*H[3, 2, 1, 1, 1]
sage: I = H.monomial_basis()
sage: H(I[2,1,1])
H[3, 1] + 3*H[2, 2] + 13*H[2, 1, 1] + 105*H[1, 1, 1, 1]
sage: I(H[2,1,1])
I[3, 1] - 15*I[4]

```

```

>>> from sage.all import *
>>> H = HallAlgebra(ZZ, Integer(2))
>>> H[Integer(2),Integer(1)]*H[Integer(1),Integer(1)]
H[3, 2] + 3*H[3, 1, 1] + 6*H[2, 2, 1] + 28*H[2, 1, 1, 1]
>>> H[Integer(3),Integer(1)]*H[Integer(2)]
H[5, 1] + H[4, 2] + 6*H[3, 3] + 3*H[4, 1, 1] + 8*H[3, 2, 1]
>>> H[Integer(2),Integer(1),Integer(1)]*H[Integer(3),Integer(1)]
H[5, 2, 1] + 2*H[4, 3, 1] + 6*H[4, 2, 2] + 7*H[5, 1, 1, 1]
+ 19*H[4, 2, 1, 1] + 24*H[3, 3, 1, 1] + 48*H[3, 2, 2, 1]
+ 105*H[4, 1, 1, 1, 1] + 224*H[3, 2, 1, 1, 1]
>>> I = H.monomial_basis()
>>> H(I[Integer(2),Integer(1),Integer(1)])
H[3, 1] + 3*H[2, 2] + 13*H[2, 1, 1] + 105*H[1, 1, 1, 1]
>>> I(H[Integer(2),Integer(1),Integer(1)])
I[3, 1] - 15*I[4]

```

If q is set to 1, the coercion to the symmetric functions sends I_λ to m_λ :

```

sage: H = HallAlgebra(QQ, 1)
sage: H[2,1] * H[2,1]
H[4, 2] + 2*H[3, 3] + 2*H[4, 1, 1] + 2*H[3, 2, 1] + 6*H[2, 2, 2] + 4*H[2, 2, 1, 1]
sage: m = SymmetricFunctions(QQ).m()
sage: m[2,1] * m[2,1]
4*m[2, 2, 1, 1] + 6*m[2, 2, 2] + 2*m[3, 2, 1] + 2*m[3, 3] + 2*m[4, 1, 1] + m[4, 2]
sage: m(H[3,1])
m[3, 1]

```

```

>>> from sage.all import *
>>> H = HallAlgebra(QQ, Integer(1))
>>> H[Integer(2),Integer(1)] * H[Integer(2),Integer(1)]
H[4, 2] + 2*H[3, 3] + 2*H[4, 1, 1] + 2*H[3, 2, 1] + 6*H[2, 2, 2] + 4*H[2, 2, 1, 1]
>>> m = SymmetricFunctions(QQ).m()
>>> m[Integer(2),Integer(1)] * m[Integer(2),Integer(1)]
4*m[2, 2, 1, 1] + 6*m[2, 2, 2] + 2*m[3, 2, 1] + 2*m[3, 3] + 2*m[4, 1, 1] + m[4, 2]
>>> m(H[Integer(3),Integer(1)])
m[3, 1]

```

We can set q to 0 (but should keep in mind that we don't get the Schur functions this way):

```
sage: H = HallAlgebra(QQ, 0)
sage: H[2,1] * H[2,1]
H[4, 2] + H[3, 3] + H[4, 1, 1] - H[3, 2, 1] - H[3, 1, 1, 1]
```

```
>>> from sage.all import *
>>> H = HallAlgebra(QQ, Integer(0))
>>> H[Integer(2), Integer(1)] * H[Integer(2), Integer(1)]
H[4, 2] + H[3, 3] + H[4, 1, 1] - H[3, 2, 1] - H[3, 1, 1, 1]
```

class Element

Bases: `IndexedFreeModuleElement`

scalar(y)

Return the scalar product of `self` and `y`.

The scalar product is given by

$$(I_\lambda, I_\mu) = \delta_{\lambda, \mu} \frac{1}{a_\lambda},$$

where a_λ is given by

$$a_\lambda = q^{|\lambda|+2n(\lambda)} \prod_k \prod_{i=1}^{l_k} (1 - q^{-i})$$

where $n(\lambda) = \sum_i (i-1)\lambda_i$ and $\lambda = (1^{l_1}, 2^{l_2}, \dots, m^{l_m})$.

Note that a_λ can be interpreted as the number of automorphisms of a certain object in a category corresponding to λ . See Lemma 2.8 in [Sch2006] for details.

EXAMPLES:

```
sage: R.<q> = ZZ[]
sage: H = HallAlgebra(R, q)
sage: H[1].scalar(H[1])
1/(q - 1)
sage: H[2].scalar(H[2])
1/(q^2 - q)
sage: H[2,1].scalar(H[2,1])
1/(q^5 - 2*q^4 + q^3)
sage: H[1,1,1,1].scalar(H[1,1,1,1])
1/(q^16 - q^15 - q^14 + 2*q^11 - q^8 - q^7 + q^6)
sage: H.an_element().scalar(H.an_element())
(4*q^2 + 9)/(q^2 - q)
```

```
>>> from sage.all import *
>>> R = ZZ['q']; (q,) = R._first_ngens(1)
>>> H = HallAlgebra(R, q)
>>> H[Integer(1)].scalar(H[Integer(1)])
1/(q - 1)
>>> H[Integer(2)].scalar(H[Integer(2)])
1/(q^2 - q)
>>> H[Integer(2), Integer(1)].scalar(H[Integer(2), Integer(1)])
1/(q^5 - 2*q^4 + q^3)
```

(continues on next page)

(continued from previous page)

```

>>> H[Integer(1), Integer(1), Integer(1), Integer(1)].scalar(H[Integer(1),
↳ Integer(1), Integer(1), Integer(1)])
1/(q^16 - q^15 - q^14 + 2*q^11 - q^8 - q^7 + q^6)
>>> H.an_element().scalar(H.an_element())
(4*q^2 + 9)/(q^2 - q)

```

antipode_on_basis (*la*)Return the antipode of the basis element indexed by *la*.

EXAMPLES:

```

sage: R = PolynomialRing(ZZ, 'q').fraction_field()
sage: q = R.gen()
sage: H = HallAlgebra(R, q)
sage: H.antipode_on_basis(Partition([1,1]))
1/q*H[2] + 1/q*H[1, 1]
sage: H.antipode_on_basis(Partition([2]))
-1/q*H[2] + ((q^2-1)/q)*H[1, 1]

sage: R.<q> = LaurentPolynomialRing(ZZ)
sage: H = HallAlgebra(R, q)
sage: H.antipode_on_basis(Partition([1,1]))
(q^-1)*H[2] + (q^-1)*H[1, 1]
sage: H.antipode_on_basis(Partition([2]))
-(q^-1)*H[2] - (q^-1-q)*H[1, 1]

```

```

>>> from sage.all import *
>>> R = PolynomialRing(ZZ, 'q').fraction_field()
>>> q = R.gen()
>>> H = HallAlgebra(R, q)
>>> H.antipode_on_basis(Partition([Integer(1), Integer(1)]))
1/q*H[2] + 1/q*H[1, 1]
>>> H.antipode_on_basis(Partition([Integer(2)]))
-1/q*H[2] + ((q^2-1)/q)*H[1, 1]

>>> R = LaurentPolynomialRing(ZZ, names=('q',)); (q,) = R._first_ngens(1)
>>> H = HallAlgebra(R, q)
>>> H.antipode_on_basis(Partition([Integer(1), Integer(1)]))
(q^-1)*H[2] + (q^-1)*H[1, 1]
>>> H.antipode_on_basis(Partition([Integer(2)]))
-(q^-1)*H[2] - (q^-1-q)*H[1, 1]

```

coproduct_on_basis (*la*)Return the coproduct of the basis element indexed by *la*.

EXAMPLES:

```

sage: R = PolynomialRing(ZZ, 'q').fraction_field()
sage: q = R.gen()
sage: H = HallAlgebra(R, q)
sage: H.coproduct_on_basis(Partition([1,1]))
H[] # H[1, 1] + 1/q*H[1] # H[1] + H[1, 1] # H[]
sage: H.coproduct_on_basis(Partition([2]))

```

(continues on next page)

(continued from previous page)

```

H[] # H[2] + ((q-1)/q)*H[1] # H[1] + H[2] # H[]
sage: H.coproduct_on_basis(Partition([2,1]))
H[] # H[2, 1] + ((q^2-1)/q^2)*H[1] # H[1, 1] + 1/q*H[1] # H[2]
+ ((q^2-1)/q^2)*H[1, 1] # H[1] + 1/q*H[2] # H[1] + H[2, 1] # H[]

sage: R.<q> = LaurentPolynomialRing(ZZ)
sage: H = HallAlgebra(R, q)
sage: H.coproduct_on_basis(Partition([2]))
H[] # H[2] - (q^-1-1)*H[1] # H[1] + H[2] # H[]
sage: H.coproduct_on_basis(Partition([2,1]))
H[] # H[2, 1] - (q^-2-1)*H[1] # H[1, 1] + (q^-1)*H[1] # H[2]
- (q^-2-1)*H[1, 1] # H[1] + (q^-1)*H[2] # H[1] + H[2, 1] # H[]

```

```

>>> from sage.all import *
>>> R = PolynomialRing(ZZ, 'q').fraction_field()
>>> q = R.gen()
>>> H = HallAlgebra(R, q)
>>> H.coproduct_on_basis(Partition([Integer(1), Integer(1)]))
H[] # H[1, 1] + 1/q*H[1] # H[1] + H[1, 1] # H[]
>>> H.coproduct_on_basis(Partition([Integer(2)]))
H[] # H[2] + ((q-1)/q)*H[1] # H[1] + H[2] # H[]
>>> H.coproduct_on_basis(Partition([Integer(2), Integer(1)]))
H[] # H[2, 1] + ((q^2-1)/q^2)*H[1] # H[1, 1] + 1/q*H[1] # H[2]
+ ((q^2-1)/q^2)*H[1, 1] # H[1] + 1/q*H[2] # H[1] + H[2, 1] # H[]

>>> R = LaurentPolynomialRing(ZZ, names=('q',)); (q,) = R._first_ngens(1)
>>> H = HallAlgebra(R, q)
>>> H.coproduct_on_basis(Partition([Integer(2)]))
H[] # H[2] - (q^-1-1)*H[1] # H[1] + H[2] # H[]
>>> H.coproduct_on_basis(Partition([Integer(2), Integer(1)]))
H[] # H[2, 1] - (q^-2-1)*H[1] # H[1, 1] + (q^-1)*H[1] # H[2]
- (q^-2-1)*H[1, 1] # H[1] + (q^-1)*H[2] # H[1] + H[2, 1] # H[]

```

counit(*x*)Return the counit of the element *x*.

EXAMPLES:

```

sage: R = PolynomialRing(ZZ, 'q').fraction_field()
sage: q = R.gen()
sage: H = HallAlgebra(R, q)
sage: H.counit(H.an_element())
2

```

```

>>> from sage.all import *
>>> R = PolynomialRing(ZZ, 'q').fraction_field()
>>> q = R.gen()
>>> H = HallAlgebra(R, q)
>>> H.counit(H.an_element())
2

```

monomial_basis()Return the basis of the Hall algebra given by monomials in the $I_{(1^r)}$.

EXAMPLES:

```
sage: R.<q> = ZZ[]
sage: H = HallAlgebra(R, q)
sage: H.monomial_basis()
Hall algebra with q=q over Univariate Polynomial Ring in q over
Integer Ring in the monomial basis
```

```
>>> from sage.all import *
>>> R = ZZ['q']; (q,) = R._first_ngens(1)
>>> H = HallAlgebra(R, q)
>>> H.monomial_basis()
Hall algebra with q=q over Univariate Polynomial Ring in q over
Integer Ring in the monomial basis
```

one_basis()

Return the index of the basis element 1.

EXAMPLES:

```
sage: R.<q> = ZZ[]
sage: H = HallAlgebra(R, q)
sage: H.one_basis()
[]
```

```
>>> from sage.all import *
>>> R = ZZ['q']; (q,) = R._first_ngens(1)
>>> H = HallAlgebra(R, q)
>>> H.one_basis()
[]
```

product_on_basis(mu, la)

Return the product of the two basis elements indexed by μ and λ .

EXAMPLES:

```
sage: R.<q> = ZZ[]
sage: H = HallAlgebra(R, q)
sage: H.product_on_basis(Partition([1,1]), Partition([1]))
H[2, 1] + (q^2+q+1)*H[1, 1, 1]
sage: H.product_on_basis(Partition([2,1]), Partition([1,1]))
H[3, 2] + (q+1)*H[3, 1, 1] + (q^2+q)*H[2, 2, 1] + (q^4+q^3+q^2)*H[2, 1, 1, 1]
sage: H.product_on_basis(Partition([3,2]), Partition([2,1]))
H[5, 3] + (q+1)*H[4, 4] + q*H[5, 2, 1] + (2*q^2-1)*H[4, 3, 1]
+ (q^3+q^2)*H[4, 2, 2] + (q^4+q^3)*H[3, 3, 2]
+ (q^4-q^2)*H[4, 2, 1, 1] + (q^5+q^4-q^3-q^2)*H[3, 3, 1, 1]
+ (q^6+q^5)*H[3, 2, 2, 1]
sage: H.product_on_basis(Partition([3,1,1]), Partition([2,1]))
H[5, 2, 1] + q*H[4, 3, 1] + (q^2-1)*H[4, 2, 2]
+ (q^3+q^2)*H[3, 3, 2] + (q^2+q+1)*H[5, 1, 1, 1]
+ (2*q^3+q^2-q-1)*H[4, 2, 1, 1] + (q^4+2*q^3+q^2)*H[3, 3, 1, 1]
+ (q^5+q^4)*H[3, 2, 2, 1] + (q^6+q^5+q^4-q^2-q-1)*H[4, 1, 1, 1, 1]
+ (q^7+q^6+q^5)*H[3, 2, 1, 1, 1]
```

```

>>> from sage.all import *
>>> R = ZZ['q']; (q,) = R._first_ngens(1)
>>> H = HallAlgebra(R, q)
>>> H.product_on_basis(Partition([Integer(1), Integer(1)]), ↵
↵ Partition([Integer(1)]))
H[2, 1] + (q^2+q+1)*H[1, 1, 1]
>>> H.product_on_basis(Partition([Integer(2), Integer(1)]), ↵
↵ Partition([Integer(1), Integer(1)]))
H[3, 2] + (q+1)*H[3, 1, 1] + (q^2+q)*H[2, 2, 1] + (q^4+q^3+q^2)*H[2, 1, 1, 1]
>>> H.product_on_basis(Partition([Integer(3), Integer(2)]), ↵
↵ Partition([Integer(2), Integer(1)]))
H[5, 3] + (q+1)*H[4, 4] + q*H[5, 2, 1] + (2*q^2-1)*H[4, 3, 1]
+ (q^3+q^2)*H[4, 2, 2] + (q^4+q^3)*H[3, 3, 2]
+ (q^4-q^2)*H[4, 2, 1, 1] + (q^5+q^4-q^3-q^2)*H[3, 3, 1, 1]
+ (q^6+q^5)*H[3, 2, 2, 1]
>>> H.product_on_basis(Partition([Integer(3), Integer(1), Integer(1)]), ↵
↵ Partition([Integer(2), Integer(1)]))
H[5, 2, 1] + q*H[4, 3, 1] + (q^2-1)*H[4, 2, 2]
+ (q^3+q^2)*H[3, 3, 2] + (q^2+q+1)*H[5, 1, 1, 1]
+ (2*q^3+q^2-q-1)*H[4, 2, 1, 1] + (q^4+2*q^3+q^2)*H[3, 3, 1, 1]
+ (q^5+q^4)*H[3, 2, 2, 1] + (q^6+q^5+q^4-q^2-q-1)*H[4, 1, 1, 1, 1]
+ (q^7+q^6+q^5)*H[3, 2, 1, 1, 1]

```

class sage.algebras.hall_algebra.HallAlgebraMonomials(*base_ring, q, prefix='I'*)

Bases: CombinatorialFreeModule

The classical Hall algebra given in terms of monomials in the $I_{(1^r)}$.

We first associate a monomial $I_{(1^{r_1})}I_{(1^{r_2})}\cdots I_{(1^{r_k})}$ with the composition $(r_1, r_2, \dots, r_k)$. However since $I_{(1^r)}$ commutes with $I_{(1^s)}$, the basis is indexed by partitions.

EXAMPLES:

We use the fraction field of $\mathbb{Z}[q]$ for our initial example:

```

sage: R = PolynomialRing(ZZ, 'q').fraction_field()
sage: q = R.gen()
sage: H = HallAlgebra(R, q)
sage: I = H.monomial_basis()

```

```

>>> from sage.all import *
>>> R = PolynomialRing(ZZ, 'q').fraction_field()
>>> q = R.gen()
>>> H = HallAlgebra(R, q)
>>> I = H.monomial_basis()

```

We check that the basis conversions are mutually inverse:

```

sage: all(H(I(H[p])) == H[p] for i in range(7) for p in Partitions(i))
True
sage: all(I(H(I[p])) == I[p] for i in range(7) for p in Partitions(i))
True

```

```
>>> from sage.all import *
>>> all(H(I(H[p])) == H[p] for i in range(Integer(7)) for p in Partitions(i))
True
>>> all(I(H(I[p])) == I[p] for i in range(Integer(7)) for p in Partitions(i))
True
```

Since Laurent polynomials are sufficient, we run the same check with the Laurent polynomial ring $\mathbb{Z}[q, q^{-1}]$:

```
sage: R.<q> = LaurentPolynomialRing(ZZ)
sage: H = HallAlgebra(R, q)
sage: I = H.monomial_basis()
sage: all(H(I(H[p])) == H[p] for i in range(6) for p in Partitions(i)) # long time
True
sage: all(I(H(I[p])) == I[p] for i in range(6) for p in Partitions(i)) # long time
True
```

```
>>> from sage.all import *
>>> R = LaurentPolynomialRing(ZZ, names=('q',)); (q,) = R._first_ngens(1)
>>> H = HallAlgebra(R, q)
>>> I = H.monomial_basis()
>>> all(H(I(H[p])) == H[p] for i in range(Integer(6)) for p in Partitions(i)) #_
↳long time
True
>>> all(I(H(I[p])) == I[p] for i in range(Integer(6)) for p in Partitions(i)) #_
↳long time
True
```

We can also convert to the symmetric functions. The natural basis corresponds to the Hall-Littlewood basis (up to a renormalization and an inversion of the q parameter), and this basis corresponds to the elementary basis (up to a renormalization):

```
sage: Sym = SymmetricFunctions(R)
sage: e = Sym.e()
sage: e(I[2,1])
(q^-1)*e[2, 1]
sage: e(I[4,2,2,1])
(q^-8)*e[4, 2, 2, 1]
sage: HLP = Sym.hall_littlewood(q).P()
sage: H(I[2,1])
H[2, 1] + (1+q+q^2)*H[1, 1, 1]
sage: HLP(e[2,1])
(1+q+q^2)*HLP[1, 1, 1] + HLP[2, 1]
sage: all( e(H[lam]) == q**(-sum([i * x for i, x in enumerate(lam)]))
.....:      * e(HLP[lam]).map_coefficients(lambda p: p(q**(-1)))
.....:      for lam in Partitions(4) )
True
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(R)
>>> e = Sym.e()
>>> e(I[Integer(2), Integer(1)])
(q^-1)*e[2, 1]
>>> e(I[Integer(4), Integer(2), Integer(2), Integer(1)])
```

(continues on next page)

(continued from previous page)

```

(q^-8)*e[4, 2, 2, 1]
>>> HLP = Sym.hall_littlewood(q).P()
>>> H(I[Integer(2),Integer(1)])
H[2, 1] + (1+q+q^2)*H[1, 1, 1]
>>> HLP(e[Integer(2),Integer(1)])
(1+q+q^2)*HLP[1, 1, 1] + HLP[2, 1]
>>> all( e(H[lam]) == q**(-sum([i * x for i, x in enumerate(lam)]))
...      * e(HLP[lam]).map_coefficients(lambda p: p(q**(-Integer(1))))
...      for lam in Partitions(Integer(4)) )
True

```

We can also do computations using a prime power:

```

sage: H = HallAlgebra(ZZ, 3)
sage: I = H.monomial_basis()
sage: i_elt = I[2,1]*I[1,1]; i_elt
I[2, 1, 1, 1]
sage: H(i_elt)
H[4, 1] + 7*H[3, 2] + 37*H[3, 1, 1] + 136*H[2, 2, 1]
+ 1495*H[2, 1, 1, 1] + 62920*H[1, 1, 1, 1, 1]

```

```

>>> from sage.all import *
>>> H = HallAlgebra(ZZ, Integer(3))
>>> I = H.monomial_basis()
>>> i_elt = I[Integer(2),Integer(1)]*I[Integer(1),Integer(1)]; i_elt
I[2, 1, 1, 1]
>>> H(i_elt)
H[4, 1] + 7*H[3, 2] + 37*H[3, 1, 1] + 136*H[2, 2, 1]
+ 1495*H[2, 1, 1, 1] + 62920*H[1, 1, 1, 1, 1]

```

class Element

Bases: `IndexedFreeModuleElement`

scalar(y)

Return the scalar product of `self` and `y`.

The scalar product is computed by converting into the natural basis.

EXAMPLES:

```

sage: R.<q> = ZZ[]
sage: I = HallAlgebra(R, q).monomial_basis()
sage: I[1].scalar(I[1])
1/(q - 1)
sage: I[2].scalar(I[2])
1/(q^4 - q^3 - q^2 + q)
sage: I[2,1].scalar(I[2,1])
(2*q + 1)/(q^6 - 2*q^5 + 2*q^3 - q^2)
sage: I[1,1,1,1].scalar(I[1,1,1,1])
24/(q^4 - 4*q^3 + 6*q^2 - 4*q + 1)
sage: I.an_element().scalar(I.an_element())
(4*q^4 - 4*q^2 + 9)/(q^4 - q^3 - q^2 + q)

```

```

>>> from sage.all import *
>>> R = ZZ['q']; (q,) = R._first_ngens(1)
>>> I = HallAlgebra(R, q).monomial_basis()
>>> I[Integer(1)].scalar(I[Integer(1)])
1/(q - 1)
>>> I[Integer(2)].scalar(I[Integer(2)])
1/(q^4 - q^3 - q^2 + q)
>>> I[Integer(2), Integer(1)].scalar(I[Integer(2), Integer(1)])
(2*q + 1)/(q^6 - 2*q^5 + 2*q^3 - q^2)
>>> I[Integer(1), Integer(1), Integer(1), Integer(1)].scalar(I[Integer(1),
↪ Integer(1), Integer(1), Integer(1)])
24/(q^4 - 4*q^3 + 6*q^2 - 4*q + 1)
>>> I.an_element().scalar(I.an_element())
(4*q^4 - 4*q^2 + 9)/(q^4 - q^3 - q^2 + q)

```

antipode_on_basis(a)

Return the antipode of the basis element indexed by a.

EXAMPLES:

```

sage: R = PolynomialRing(ZZ, 'q').fraction_field()
sage: q = R.gen()
sage: I = HallAlgebra(R, q).monomial_basis()
sage: I.antipode_on_basis(Partition([1]))
-I[1]
sage: I.antipode_on_basis(Partition([2]))
1/q*I[1, 1] - I[2]
sage: I.antipode_on_basis(Partition([2, 1]))
-1/q*I[1, 1, 1] + I[2, 1]

sage: R.<q> = LaurentPolynomialRing(ZZ)
sage: I = HallAlgebra(R, q).monomial_basis()
sage: I.antipode_on_basis(Partition([2, 1]))
-(q^-1)*I[1, 1, 1] + I[2, 1]

```

```

>>> from sage.all import *
>>> R = PolynomialRing(ZZ, 'q').fraction_field()
>>> q = R.gen()
>>> I = HallAlgebra(R, q).monomial_basis()
>>> I.antipode_on_basis(Partition([Integer(1)]))
-I[1]
>>> I.antipode_on_basis(Partition([Integer(2)]))
1/q*I[1, 1] - I[2]
>>> I.antipode_on_basis(Partition([Integer(2), Integer(1)]))
-1/q*I[1, 1, 1] + I[2, 1]

>>> R = LaurentPolynomialRing(ZZ, names=('q',)); (q,) = R._first_ngens(1)
>>> I = HallAlgebra(R, q).monomial_basis()
>>> I.antipode_on_basis(Partition([Integer(2), Integer(1)]))
-(q^-1)*I[1, 1, 1] + I[2, 1]

```

coproduct_on_basis(a)

Return the coproduct of the basis element indexed by a.

EXAMPLES:

```

sage: R = PolynomialRing(ZZ, 'q').fraction_field()
sage: q = R.gen()
sage: I = HallAlgebra(R, q).monomial_basis()
sage: I.coproduct_on_basis(Partition([1]))
I[] # I[1] + I[1] # I[]
sage: I.coproduct_on_basis(Partition([2]))
I[] # I[2] + 1/q*I[1] # I[1] + I[2] # I[]
sage: I.coproduct_on_basis(Partition([2,1]))
I[] # I[2, 1] + 1/q*I[1] # I[1, 1] + I[1] # I[2]
      + 1/q*I[1, 1] # I[1] + I[2] # I[1] + I[2, 1] # I[]

sage: R.<q> = LaurentPolynomialRing(ZZ)
sage: I = HallAlgebra(R, q).monomial_basis()
sage: I.coproduct_on_basis(Partition([2,1]))
I[] # I[2, 1] + (q^-1)*I[1] # I[1, 1] + I[1] # I[2]
      + (q^-1)*I[1, 1] # I[1] + I[2] # I[1] + I[2, 1] # I[]

```

```

>>> from sage.all import *
>>> R = PolynomialRing(ZZ, 'q').fraction_field()
>>> q = R.gen()
>>> I = HallAlgebra(R, q).monomial_basis()
>>> I.coproduct_on_basis(Partition([Integer(1)]))
I[] # I[1] + I[1] # I[]
>>> I.coproduct_on_basis(Partition([Integer(2)]))
I[] # I[2] + 1/q*I[1] # I[1] + I[2] # I[]
>>> I.coproduct_on_basis(Partition([Integer(2), Integer(1)]))
I[] # I[2, 1] + 1/q*I[1] # I[1, 1] + I[1] # I[2]
      + 1/q*I[1, 1] # I[1] + I[2] # I[1] + I[2, 1] # I[]

>>> R = LaurentPolynomialRing(ZZ, names=('q',)); (q,) = R._first_ngens(1)
>>> I = HallAlgebra(R, q).monomial_basis()
>>> I.coproduct_on_basis(Partition([Integer(2), Integer(1)]))
I[] # I[2, 1] + (q^-1)*I[1] # I[1, 1] + I[1] # I[2]
      + (q^-1)*I[1, 1] # I[1] + I[2] # I[1] + I[2, 1] # I[]

```

counit (*x*)

Return the counit of the element *x*.

EXAMPLES:

```

sage: R = PolynomialRing(ZZ, 'q').fraction_field()
sage: q = R.gen()
sage: I = HallAlgebra(R, q).monomial_basis()
sage: I.counit(I.an_element())
2

```

```

>>> from sage.all import *
>>> R = PolynomialRing(ZZ, 'q').fraction_field()
>>> q = R.gen()
>>> I = HallAlgebra(R, q).monomial_basis()
>>> I.counit(I.an_element())
2

```

one_basis()

Return the index of the basis element l .

EXAMPLES:

```
sage: R.<q> = ZZ[]
sage: I = HallAlgebra(R, q).monomial_basis()
sage: I.one_basis()
[]
```

```
>>> from sage.all import *
>>> R = ZZ['q']; (q,) = R._first_ngens(1)
>>> I = HallAlgebra(R, q).monomial_basis()
>>> I.one_basis()
[]
```

product_on_basis(a, b)

Return the product of the two basis elements indexed by a and b .

EXAMPLES:

```
sage: R.<q> = ZZ[]
sage: I = HallAlgebra(R, q).monomial_basis()
sage: I.product_on_basis(Partition([4, 2, 1]), Partition([3, 2, 1]))
I[4, 3, 2, 2, 1, 1]
```

```
>>> from sage.all import *
>>> R = ZZ['q']; (q,) = R._first_ngens(1)
>>> I = HallAlgebra(R, q).monomial_basis()
>>> I.product_on_basis(Partition([Integer(4), Integer(2), Integer(1)]),
↳ Partition([Integer(3), Integer(2), Integer(1)]))
I[4, 3, 2, 2, 1, 1]
```

`sage.algebras.hall_algebra.transpose_cmp(x, y)`

Compare partitions x and y in transpose dominance order.

We say partitions μ and λ satisfy $\mu \prec \lambda$ in transpose dominance order if for all $i \geq 1$ we have:

$$l_1 + 2l_2 + \cdots + (i-1)l_{i-1} + i(l_i + l_{i+1} + \cdots) \leq m_1 + 2m_2 + \cdots + (i-1)m_{i-1} + i(m_i + m_{i+1} + \cdots),$$

where l_k denotes the number of appearances of k in λ , and m_k denotes the number of appearances of k in μ .

Equivalently, $\mu \prec \lambda$ if the conjugate of the partition μ dominates the conjugate of the partition λ .

Since this is a partial ordering, we fallback to lex ordering $\mu <_L \lambda$ if we cannot compare in the transpose order.

EXAMPLES:

```
sage: from sage.algebras.hall_algebra import transpose_cmp
sage: transpose_cmp(Partition([4, 3, 1]), Partition([3, 2, 2, 1]))
-1
sage: transpose_cmp(Partition([2, 2, 1]), Partition([3, 2]))
1
sage: transpose_cmp(Partition([4, 1, 1]), Partition([4, 1, 1]))
0
```

```

>>> from sage.all import *
>>> from sage.algebras.hall_algebra import transpose_cmp
>>> transpose_cmp(Partition([Integer(4), Integer(3), Integer(1)]),
↳ Partition([Integer(3), Integer(2), Integer(2), Integer(1)]))
-1
>>> transpose_cmp(Partition([Integer(2), Integer(2), Integer(1)]),
↳ Partition([Integer(3), Integer(2)]))
1
>>> transpose_cmp(Partition([Integer(4), Integer(1), Integer(1)]),
↳ Partition([Integer(4), Integer(1), Integer(1)]))
0

```

5.10 Incidence Algebras

class sage.combinat.posets.incidence_algebras.**IncidenceAlgebra** (R, P , prefix='I')

Bases: `CombinatorialFreeModule`

The incidence algebra of a poset.

Let P be a poset and R be a commutative unital associative ring. The *incidence algebra* I_P is the algebra of functions $\alpha: P \times P \rightarrow R$ such that $\alpha(x, y) = 0$ if $x \not\leq y$ where multiplication is given by convolution:

$$(\alpha * \beta)(x, y) = \sum_{x \leq k \leq y} \alpha(x, k) \beta(k, y).$$

This has a natural basis given by indicator functions for the interval $[a, b]$, i.e. $X_{a,b}(x, y) = \delta_{ax} \delta_{by}$. The incidence algebra is a unital algebra with the identity given by the Kronecker delta $\delta(x, y) = \delta_{xy}$. The Möbius function of P is another element of I_P whose inverse is the ζ function of the poset (so $\zeta(x, y) = 1$ for every interval $[x, y]$).

Todo

Implement the incidence coalgebra.

REFERENCES:

- [Wikipedia article Incidence_algebra](#)

class **Element**

Bases: `IndexedFreeModuleElement`

An element of an incidence algebra.

is_unit()

Return if `self` is a unit.

EXAMPLES:

```

sage: P = posets.BooleanLattice(2)
sage: I = P.incidence_algebra(QQ)
sage: mu = I.moebius()
sage: mu.is_unit()
True
sage: zeta = I.zeta()
sage: zeta.is_unit()

```

(continues on next page)

(continued from previous page)

```

True
sage: x = mu - I.zeta() + I[2,2]
sage: x.is_unit()
False
sage: y = I.moebius() + I.zeta()
sage: y.is_unit()
True

```

```

>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(2))
>>> I = P.incidence_algebra(QQ)
>>> mu = I.moebius()
>>> mu.is_unit()
True
>>> zeta = I.zeta()
>>> zeta.is_unit()
True
>>> x = mu - I.zeta() + I[Integer(2), Integer(2)]
>>> x.is_unit()
False
>>> y = I.moebius() + I.zeta()
>>> y.is_unit()
True

```

This depends on the base ring:

```

sage: I = P.incidence_algebra(ZZ)
sage: y = I.moebius() + I.zeta()
sage: y.is_unit()
False

```

```

>>> from sage.all import *
>>> I = P.incidence_algebra(ZZ)
>>> y = I.moebius() + I.zeta()
>>> y.is_unit()
False

```

to_matrix()

Return `self` as a matrix.

We define a matrix $M_{xy} = \alpha(x, y)$ for some element $\alpha \in I_P$ in the incidence algebra I_P and we order the elements $x, y \in P$ by some linear extension of P . This defines an algebra (iso)morphism; in particular, multiplication in the incidence algebra goes to matrix multiplication.

EXAMPLES:

```

sage: P = posets.BooleanLattice(2)
sage: I = P.incidence_algebra(QQ)
sage: I.moebius().to_matrix()
[ 1 -1 -1  1]
[ 0  1  0 -1]
[ 0  0  1 -1]
[ 0  0  0  1]

```

(continues on next page)

(continued from previous page)

```
sage: I.zeta().to_matrix()
[1 1 1 1]
[0 1 0 1]
[0 0 1 1]
[0 0 0 1]
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(2))
>>> I = P.incidence_algebra(QQ)
>>> I.moebius().to_matrix()
[ 1 -1 -1  1]
[ 0  1  0 -1]
[ 0  0  1 -1]
[ 0  0  0  1]
>>> I.zeta().to_matrix()
[1 1 1 1]
[0 1 0 1]
[0 0 1 1]
[0 0 0 1]
```

delta()

Return the element 1 in `self` (which is the Kronecker delta $\delta(x, y)$).

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: I = P.incidence_algebra(QQ)
sage: I.one()
I[0, 0] + I[1, 1] + I[2, 2] + I[3, 3] + I[4, 4] + I[5, 5]
+ I[6, 6] + I[7, 7] + I[8, 8] + I[9, 9] + I[10, 10]
+ I[11, 11] + I[12, 12] + I[13, 13] + I[14, 14] + I[15, 15]
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> I = P.incidence_algebra(QQ)
>>> I.one()
I[0, 0] + I[1, 1] + I[2, 2] + I[3, 3] + I[4, 4] + I[5, 5]
+ I[6, 6] + I[7, 7] + I[8, 8] + I[9, 9] + I[10, 10]
+ I[11, 11] + I[12, 12] + I[13, 13] + I[14, 14] + I[15, 15]
```

moebius()

Return the Möbius function of `self`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(2)
sage: I = P.incidence_algebra(QQ)
sage: I.moebius()
I[0, 0] - I[0, 1] - I[0, 2] + I[0, 3] + I[1, 1]
- I[1, 3] + I[2, 2] - I[2, 3] + I[3, 3]
```

```

>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(2))
>>> I = P.incidence_algebra(QQ)
>>> I.moebius()
I[0, 0] - I[0, 1] - I[0, 2] + I[0, 3] + I[1, 1]
- I[1, 3] + I[2, 2] - I[2, 3] + I[3, 3]

```

one()

Return the element 1 in self (which is the Kronecker delta $\delta(x, y)$).

EXAMPLES:

```

sage: P = posets.BooleanLattice(4)
sage: I = P.incidence_algebra(QQ)
sage: I.one()
I[0, 0] + I[1, 1] + I[2, 2] + I[3, 3] + I[4, 4] + I[5, 5]
+ I[6, 6] + I[7, 7] + I[8, 8] + I[9, 9] + I[10, 10]
+ I[11, 11] + I[12, 12] + I[13, 13] + I[14, 14] + I[15, 15]

```

```

>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> I = P.incidence_algebra(QQ)
>>> I.one()
I[0, 0] + I[1, 1] + I[2, 2] + I[3, 3] + I[4, 4] + I[5, 5]
+ I[6, 6] + I[7, 7] + I[8, 8] + I[9, 9] + I[10, 10]
+ I[11, 11] + I[12, 12] + I[13, 13] + I[14, 14] + I[15, 15]

```

poset()

Return the defining poset of self.

EXAMPLES:

```

sage: P = posets.BooleanLattice(4)
sage: I = P.incidence_algebra(QQ)
sage: I.poset()
Finite lattice containing 16 elements
sage: I.poset() == P
True

```

```

>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> I = P.incidence_algebra(QQ)
>>> I.poset()
Finite lattice containing 16 elements
>>> I.poset() == P
True

```

product_on_basis(A, B)

Return the product of basis elements indexed by A and B.

EXAMPLES:

```

sage: P = posets.BooleanLattice(4)
sage: I = P.incidence_algebra(QQ)

```

(continues on next page)

(continued from previous page)

```
sage: I.product_on_basis((1, 3), (3, 11))
I[1, 11]
sage: I.product_on_basis((1, 3), (2, 2))
0
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> I = P.incidence_algebra(QQ)
>>> I.product_on_basis((Integer(1), Integer(3)), (Integer(3), Integer(11)))
I[1, 11]
>>> I.product_on_basis((Integer(1), Integer(3)), (Integer(2), Integer(2)))
0
```

reduced_subalgebra (*prefix*='R')

Return the reduced incidence subalgebra.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: I = P.incidence_algebra(QQ)
sage: I.reduced_subalgebra()
Reduced incidence algebra of Finite lattice containing 16 elements
over Rational Field
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> I = P.incidence_algebra(QQ)
>>> I.reduced_subalgebra()
Reduced incidence algebra of Finite lattice containing 16 elements
over Rational Field
```

some_elements()

Return a list of elements of self.

EXAMPLES:

```
sage: P = posets.BooleanLattice(1)
sage: I = P.incidence_algebra(QQ)
sage: Ielts = I.some_elements(); Ielts # random
[2*I[0, 0] + 2*I[0, 1] + 3*I[1, 1],
 I[0, 0] - I[0, 1] + I[1, 1],
 I[0, 0] + I[0, 1] + I[1, 1]]
sage: [a in I for a in Ielts]
[True, True, True]
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(1))
>>> I = P.incidence_algebra(QQ)
>>> Ielts = I.some_elements(); Ielts # random
[2*I[0, 0] + 2*I[0, 1] + 3*I[1, 1],
 I[0, 0] - I[0, 1] + I[1, 1],
 I[0, 0] + I[0, 1] + I[1, 1]]
```

(continues on next page)

(continued from previous page)

```
>>> [a in I for a in Ielts]
[True, True, True]
```

zeta()Return the ζ function in `self`.The ζ function on a poset P is given by

$$\zeta(x, y) = \begin{cases} 1 & x \leq y, \\ 0 & x \not\leq y. \end{cases}$$

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: I = P.incidence_algebra(QQ)
sage: I.zeta() * I.moebius() == I.one()
True
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> I = P.incidence_algebra(QQ)
>>> I.zeta() * I.moebius() == I.one()
True
```

class `sage.combinat.posets.incidence_algebras.ReducedIncidenceAlgebra` (I , *prefix*='R')

Bases: `CombinatorialFreeModule`

The reduced incidence algebra of a poset.

The reduced incidence algebra R_P is a subalgebra of the incidence algebra I_P where $\alpha(x, y) = \alpha(x', y')$ when $[x, y]$ is isomorphic to $[x', y']$ as posets. Thus the delta, Möbius, and zeta functions are all elements of R_P .

class `Element`Bases: `IndexedFreeModuleElement`

An element of a reduced incidence algebra.

is_unit()Return if `self` is a unit.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: x = R.an_element()
sage: x.is_unit()
True
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> R = P.incidence_algebra(QQ).reduced_subalgebra()
>>> x = R.an_element()
>>> x.is_unit()
True
```

lift()Return the lift of `self` to the ambient space.

EXAMPLES:

```

sage: P = posets.BooleanLattice(2)
sage: I = P.incidence_algebra(QQ)
sage: R = I.reduced_subalgebra()
sage: x = R.an_element(); x
2*R[(0, 0)] + 2*R[(0, 1)] + 3*R[(0, 3)]
sage: x.lift()
2*I[0, 0] + 2*I[0, 1] + 2*I[0, 2] + 3*I[0, 3] + 2*I[1, 1]
+ 2*I[1, 3] + 2*I[2, 2] + 2*I[2, 3] + 2*I[3, 3]

```

```

>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(2))
>>> I = P.incidence_algebra(QQ)
>>> R = I.reduced_subalgebra()
>>> x = R.an_element(); x
2*R[(0, 0)] + 2*R[(0, 1)] + 3*R[(0, 3)]
>>> x.lift()
2*I[0, 0] + 2*I[0, 1] + 2*I[0, 2] + 3*I[0, 3] + 2*I[1, 1]
+ 2*I[1, 3] + 2*I[2, 2] + 2*I[2, 3] + 2*I[3, 3]

```

to_matrix()Return `self` as a matrix.

EXAMPLES:

```

sage: P = posets.BooleanLattice(2)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: mu = R.moebius()
sage: mu.to_matrix()
[ 1 -1 -1  1]
[ 0  1  0 -1]
[ 0  0  1 -1]
[ 0  0  0  1]

```

```

>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(2))
>>> R = P.incidence_algebra(QQ).reduced_subalgebra()
>>> mu = R.moebius()
>>> mu.to_matrix()
[ 1 -1 -1  1]
[ 0  1  0 -1]
[ 0  0  1 -1]
[ 0  0  0  1]

```

delta()Return the Kronecker delta function in `self`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.delta()
R[(0, 0)]
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> R = P.incidence_algebra(QQ).reduced_subalgebra()
>>> R.delta()
R[(0, 0)]
```

lift()

Return the lift morphism from `self` to the ambient space.

EXAMPLES:

```
sage: P = posets.BooleanLattice(2)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.lift
Generic morphism:
  From: Reduced incidence algebra of Finite lattice containing 4 elements_
  ↪over Rational Field
  To:   Incidence algebra of Finite lattice containing 4 elements over_
  ↪Rational Field
sage: R.an_element() - R.one()
R[(0, 0)] + 2*R[(0, 1)] + 3*R[(0, 3)]
sage: R.lift(R.an_element() - R.one())
I[0, 0] + 2*I[0, 1] + 2*I[0, 2] + 3*I[0, 3] + I[1, 1]
+ 2*I[1, 3] + I[2, 2] + 2*I[2, 3] + I[3, 3]
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(2))
>>> R = P.incidence_algebra(QQ).reduced_subalgebra()
>>> R.lift
Generic morphism:
  From: Reduced incidence algebra of Finite lattice containing 4 elements_
  ↪over Rational Field
  To:   Incidence algebra of Finite lattice containing 4 elements over_
  ↪Rational Field
>>> R.an_element() - R.one()
R[(0, 0)] + 2*R[(0, 1)] + 3*R[(0, 3)]
>>> R.lift(R.an_element() - R.one())
I[0, 0] + 2*I[0, 1] + 2*I[0, 2] + 3*I[0, 3] + I[1, 1]
+ 2*I[1, 3] + I[2, 2] + 2*I[2, 3] + I[3, 3]
```

moebius()

Return the Möbius function of `self`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
```

(continues on next page)

(continued from previous page)

```
sage: R.moebius()
R[(0, 0)] - R[(0, 1)] + R[(0, 3)] - R[(0, 7)] + R[(0, 15)]
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> R = P.incidence_algebra(QQ).reduced_subalgebra()
>>> R.moebius()
R[(0, 0)] - R[(0, 1)] + R[(0, 3)] - R[(0, 7)] + R[(0, 15)]
```

one_basis()

Return the index of the element 1 in self.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.one_basis()
(0, 0)
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> R = P.incidence_algebra(QQ).reduced_subalgebra()
>>> R.one_basis()
(0, 0)
```

poset()

Return the defining poset of self.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.poset()
Finite lattice containing 16 elements
sage: R.poset() == P
True
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> R = P.incidence_algebra(QQ).reduced_subalgebra()
>>> R.poset()
Finite lattice containing 16 elements
>>> R.poset() == P
True
```

some_elements()

Return a list of elements of self.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
```

(continues on next page)

(continued from previous page)

```
sage: R.some_elements()
[2*R[(0, 0)] + 2*R[(0, 1)] + 3*R[(0, 3)],
 R[(0, 0)] - R[(0, 1)] + R[(0, 3)] - R[(0, 7)] + R[(0, 15)],
 R[(0, 0)] + R[(0, 1)] + R[(0, 3)] + R[(0, 7)] + R[(0, 15)]]
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> R = P.incidence_algebra(QQ).reduced_subalgebra()
>>> R.some_elements()
[2*R[(0, 0)] + 2*R[(0, 1)] + 3*R[(0, 3)],
 R[(0, 0)] - R[(0, 1)] + R[(0, 3)] - R[(0, 7)] + R[(0, 15)],
 R[(0, 0)] + R[(0, 1)] + R[(0, 3)] + R[(0, 7)] + R[(0, 15)]]
```

zeta()Return the ζ function in self.The ζ function on a poset P is given by

$$\zeta(x, y) = \begin{cases} 1 & x \leq y, \\ 0 & x \not\leq y. \end{cases}$$

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.zeta()
R[(0, 0)] + R[(0, 1)] + R[(0, 3)] + R[(0, 7)] + R[(0, 15)]
```

```
>>> from sage.all import *
>>> P = posets.BooleanLattice(Integer(4))
>>> R = P.incidence_algebra(QQ).reduced_subalgebra()
>>> R.zeta()
R[(0, 0)] + R[(0, 1)] + R[(0, 3)] + R[(0, 7)] + R[(0, 15)]
```

5.11 Group algebras

This functionality has been moved to `sage.categories.algebra_functor`.`sage.algebras.group_algebra.GroupAlgebra(G, R=Integer Ring)`Return the group algebra of G over R .

INPUT:

- G – a group
- R – (default: $\mathbf{Z}$) a ring

EXAMPLES:

The *group algebra* $A = RG$ is the space of formal linear combinations of elements of G with coefficients in R :

```
sage: G = DihedralGroup(3)
sage: R = QQ
sage: A = GroupAlgebra(G, R); A
```

(continues on next page)

(continued from previous page)

```
Algebra of Dihedral group of order 6 as a permutation group over Rational Field
sage: a = A.an_element(); a
() + (1,2) + 3*(1,2,3) + 2*(1,3,2)
```

```
>>> from sage.all import *
>>> G = DihedralGroup(Integer(3))
>>> R = QQ
>>> A = GroupAlgebra(G, R); A
Algebra of Dihedral group of order 6 as a permutation group over Rational Field
>>> a = A.an_element(); a
() + (1,2) + 3*(1,2,3) + 2*(1,3,2)
```

This space is endowed with an algebra structure, obtained by extending by bilinearity the multiplication of G to a multiplication on RG :

```
sage: A in Algebras
True
sage: a * a
14*() + 5*(2,3) + 2*(1,2) + 10*(1,2,3) + 13*(1,3,2) + 5*(1,3)
```

```
>>> from sage.all import *
>>> A in Algebras
True
>>> a * a
14*() + 5*(2,3) + 2*(1,2) + 10*(1,2,3) + 13*(1,3,2) + 5*(1,3)
```

`GroupAlgebra()` is just a short hand for a more general construction that covers, e.g., monoid algebras, additive group algebras and so on:

```
sage: G.algebra(QQ)
Algebra of Dihedral group of order 6 as a permutation group over Rational Field

sage: GroupAlgebra(G,QQ) is G.algebra(QQ)
True

sage: M = Monoids().example(); M
An example of a monoid:
the free monoid generated by ('a', 'b', 'c', 'd')
sage: M.algebra(QQ)
Algebra of An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd'
↪)
over Rational Field
```

```
>>> from sage.all import *
>>> G.algebra(QQ)
Algebra of Dihedral group of order 6 as a permutation group over Rational Field

>>> GroupAlgebra(G,QQ) is G.algebra(QQ)
True

>>> M = Monoids().example(); M
An example of a monoid:
```

(continues on next page)

(continued from previous page)

```

the free monoid generated by ('a', 'b', 'c', 'd')
>>> M.algebra(QQ)
Algebra of An example of a monoid: the free monoid generated by ('a', 'b', 'c', 'd
→')
      over Rational Field

```

See the documentation of `sage.categories.algebra_functor` for details.

```

class sage.algebras.group_algebra.GroupAlgebra_class(R, basis_keys=None, element_class=None,
category=None, prefix=None, names=None,
**kwds)

```

Bases: `CombinatorialFreeModule`

5.12 Grossman-Larson Hopf Algebras

AUTHORS:

- Frédéric Chapoton (2017)

```

class sage.combinat.grossman_larson_algebras.GrossmanLarsonAlgebra(R, names=None)

```

Bases: `CombinatorialFreeModule`

The Grossman-Larson Hopf Algebra.

The Grossman-Larson Hopf Algebras are Hopf algebras with a basis indexed by forests of decorated rooted trees. They are the universal enveloping algebras of free pre-Lie algebras, seen as Lie algebras.

The Grossman-Larson Hopf algebra on a given set E has an explicit description using rooted forests. The underlying vector space has a basis indexed by finite rooted forests endowed with a map from their vertices to E (called the “labeling”). In this basis, the product of two (decorated) rooted forests $S * T$ is a sum over all maps from the set of roots of T to the union of a singleton $\{\#\}$ and the set of vertices of S . Given such a map, one defines a new forest as follows. Starting from the disjoint union of all rooted trees of S and T , one adds an edge from every root of T to its image when this image is not the fake vertex labelled $\#$. The coproduct sends a rooted forest T to the sum of all tensors $T_1 \otimes T_2$ obtained by splitting the connected components of T into two subsets and letting T_1 be the forest formed by the first subset and T_2 the forest formed by the second. This yields a connected graded Hopf algebra (the degree of a forest is its number of vertices).

See [Pana2002] (Section 2) and [GroLar1]. (Note that both references use rooted trees rather than rooted forests, so think of each rooted forest grafted onto a new root. Also, the product is reversed, so they are defining the opposite algebra structure.)

Warning

For technical reasons, instead of using forests as labels for the basis, we use rooted trees. Their root vertex should be considered as a fake vertex. This fake root vertex is labelled ' $\#$ ' when labels are present.

EXAMPLES:

```

sage: G = algebras.GrossmanLarson(QQ, 'xy')
sage: x, y = G.single_vertex_all()
sage: ascii_art(x*y)
B   + B
#       #_

```

(continues on next page)

(continued from previous page)

```

|      / /
x      x y
|
y

sage: ascii_art(x*x*x)
B  + B      + 3*B      + B
#      #      #_      _#_
|      |      / /      / / /
x      x_      x x      x x x
|      / /      |
x      x x      x
|
x

```

```

>>> from sage.all import *
>>> G = algebras.GrossmanLarson(QQ, 'xy')
>>> x, y = G.single_vertex_all()
>>> ascii_art(x*y)
B  + B
#      #_
|      / /
x      x y
|
y

>>> ascii_art(x*x*x)
B  + B      + 3*B      + B
#      #      #_      _#_
|      |      / /      / / /
x      x_      x x      x x x
|      / /      |
x      x x      x
|
x

```

The Grossman-Larson algebra is associative:

```

sage: z = x * y
sage: x * (y * z) == (x * y) * z
True

```

```

>>> from sage.all import *
>>> z = x * y
>>> x * (y * z) == (x * y) * z
True

```

It is not commutative:

```

sage: x * y == y * x
False

```

```
>>> from sage.all import *
>>> x * y == y * x
False
```

When `None` is given as input, unlabelled forests are used instead; this corresponds to a 1-element set E :

```
sage: G = algebras.GrossmanLarson(QQ, None)
sage: x = G.single_vertex_all()[0]
sage: ascii_art(x*x)
B + B
  o      o_
  |      / /
  o      o o
  |
  o
```

```
>>> from sage.all import *
>>> G = algebras.GrossmanLarson(QQ, None)
>>> x = G.single_vertex_all()[Integer(0)]
>>> ascii_art(x*x)
B + B
  o      o_
  |      / /
  o      o o
  |
  o
```

Note

Variables names can be `None`, a list of strings, a string or an integer. When `None` is given, unlabelled rooted forests are used. When a single string is given, each letter is taken as a variable. See `sage.combinat.words.alphabet.build_alphabet()`.

Warning

Beware that the underlying combinatorial free module is based either on `RootedTrees` or on `Labelled-RootedTrees`, with no restriction on the labellings. This means that all code calling the `basis()` method would not give meaningful results, since `basis()` returns many “chaff” elements that do not belong to the algebra.

REFERENCES:

- [Pana2002]
- [GroLar1]

`an_element()`

Return an element of `self`.

EXAMPLES:

```
sage: A = algebras.GrossmanLarson(QQ, 'xy')
sage: A.an_element()
B[#[x[]]] + 2*B[#[x[x[]]]] + 2*B[#[x[], x[]]]
```

```
>>> from sage.all import *
>>> A = algebras.GrossmanLarson(QQ, 'xy')
>>> A.an_element()
B[#[x[]]] + 2*B[#[x[x[]]]] + 2*B[#[x[], x[]]]
```

antipode_on_basis(*x*)

Return the antipode of a forest.

EXAMPLES:

```
sage: G = algebras.GrossmanLarson(QQ, 2)
sage: x, y = G.single_vertex_all()
sage: G.antipode(x) # indirect doctest
-B[#[0[]]]

sage: G.antipode(y*x) # indirect doctest
B[#[0[1[]]]] + B[#[0[], 1[]]]
```

```
>>> from sage.all import *
>>> G = algebras.GrossmanLarson(QQ, Integer(2))
>>> x, y = G.single_vertex_all()
>>> G.antipode(x) # indirect doctest
-B[#[0[]]]

>>> G.antipode(y*x) # indirect doctest
B[#[0[1[]]]] + B[#[0[], 1[]]]
```

change_ring(*R*)

Return the Grossman-Larson algebra in the same variables over *R*.

INPUT:

- *R* – a ring

EXAMPLES:

```
sage: A = algebras.GrossmanLarson(ZZ, 'fgh')
sage: A.change_ring(QQ)
Grossman-Larson Hopf algebra on 3 generators ['f', 'g', 'h']
over Rational Field
```

```
>>> from sage.all import *
>>> A = algebras.GrossmanLarson(ZZ, 'fgh')
>>> A.change_ring(QQ)
Grossman-Larson Hopf algebra on 3 generators ['f', 'g', 'h']
over Rational Field
```

coproduct_on_basis(*x*)

Return the coproduct of a forest.

EXAMPLES:

```

sage: G = algebras.GrossmanLarson(QQ, 2)
sage: x, y = G.single_vertex_all()
sage: ascii_art(G.coproduct(x)) # indirect doctest
1 # B + B # 1
  #      #
  |      |
  0      0

sage: Delta_xy = G.coproduct(y*x)
sage: ascii_art(Delta_xy) # random indirect doctest
1 # B      + 1 # B + B # B + B      # 1 + B # B + B # 1
  #_          #      #      #      #_          #      #      #
  / /          |      |      |      / /          |      |      |
0 1            1      0      1      0 1            1      0      1
                |
                0
                                |
                                0

```

```

>>> from sage.all import *
>>> G = algebras.GrossmanLarson(QQ, Integer(2))
>>> x, y = G.single_vertex_all()
>>> ascii_art(G.coproduct(x)) # indirect doctest
1 # B + B # 1
  #      #
  |      |
  0      0

>>> Delta_xy = G.coproduct(y*x)
>>> ascii_art(Delta_xy) # random indirect doctest
1 # B      + 1 # B + B # B + B      # 1 + B # B + B # 1
  #_          #      #      #      #_          #      #      #
  / /          |      |      |      / /          |      |      |
0 1            1      0      1      0 1            1      0      1
                |
                0
                                |
                                0

```

counit_on_basis(x)

Return the counit on a basis element.

This is zero unless the forest x is empty.

EXAMPLES:

```

sage: A = algebras.GrossmanLarson(QQ, 'xy')
sage: RT = A.basis().keys()
sage: x = RT([RT([], 'x')], '#')
sage: A.counit_on_basis(x)
0
sage: A.counit_on_basis(RT([], '#'))
1

```

```

>>> from sage.all import *
>>> A = algebras.GrossmanLarson(QQ, 'xy')
>>> RT = A.basis().keys()

```

(continues on next page)

(continued from previous page)

```

>>> x = RT([RT([], 'x')], '#')
>>> A.counit_on_basis(x)
0
>>> A.counit_on_basis(RT([], '#'))
1

```

degree_on_basis(*t*)

Return the degree of a rooted forest in the Grossman-Larson algebra.

This is the total number of vertices of the forest.

EXAMPLES:

```

sage: A = algebras.GrossmanLarson(QQ, '@')
sage: RT = A.basis().keys()
sage: A.degree_on_basis(RT([RT([])]))
1

```

```

>>> from sage.all import *
>>> A = algebras.GrossmanLarson(QQ, '@')
>>> RT = A.basis().keys()
>>> A.degree_on_basis(RT([RT([])]))
1

```

one_basis()

Return the empty rooted forest.

EXAMPLES:

```

sage: A = algebras.GrossmanLarson(QQ, 'ab')
sage: A.one_basis()
# []

sage: A = algebras.GrossmanLarson(QQ, None)
sage: A.one_basis()
[]

```

```

>>> from sage.all import *
>>> A = algebras.GrossmanLarson(QQ, 'ab')
>>> A.one_basis()
# []

>>> A = algebras.GrossmanLarson(QQ, None)
>>> A.one_basis()
[]

```

product_on_basis(*x*, *y*)

Return the product of two forests *x* and *y*.

This is the sum over all possible ways for the components of the forest *y* to either fall side-by-side with components of *x* or be grafted on a vertex of *x*.

EXAMPLES:


```
sage: A = algebras.GrossmanLarson(QQ, None)
sage: RT = A.basis().keys()
sage: x = RT([RT([])])
sage: A.product_on_basis(x, x)
B[[[[]]]] + B[[[]], []]
```

```
>>> from sage.all import *
>>> A = algebras.GrossmanLarson(QQ, None)
>>> RT = A.basis().keys()
>>> x = RT([RT([])])
>>> A.product_on_basis(x, x)
B[[[[]]]] + B[[[]], []]
```

Check that the product is the correct one:

```
sage: A = algebras.GrossmanLarson(QQ, 'uv')
sage: RT = A.basis().keys()
sage: Tu = RT([RT([], 'u')], '#')
sage: Tv = RT([RT([], 'v')], '#')
sage: A.product_on_basis(Tu, Tv)
B[#[u[v[]]]] + B[#[u[], v[]]]
```

```
>>> from sage.all import *
>>> A = algebras.GrossmanLarson(QQ, 'uv')
>>> RT = A.basis().keys()
>>> Tu = RT([RT([], 'u')], '#')
>>> Tv = RT([RT([], 'v')], '#')
>>> A.product_on_basis(Tu, Tv)
B[#[u[v[]]]] + B[#[u[], v[]]]
```

`single_vertex(i)`

Return the i -th rooted forest with one vertex.

This is the rooted forest with just one vertex, labelled by the i -th element of the label list.

See also

`single_vertex_all()`.

INPUT:

- i – nonnegative integer

EXAMPLES:

```
sage: F = algebras.GrossmanLarson(ZZ, 'xyz')
sage: F.single_vertex(0)
B[#[x[]]]

sage: F.single_vertex(4)
Traceback (most recent call last):
...
IndexError: argument i (= 4) must be between 0 and 2
```

```

>>> from sage.all import *
>>> F = algebras.GrossmanLarson(ZZ, 'xyz')
>>> F.single_vertex(Integer(0))
B[#[x[]]]

>>> F.single_vertex(Integer(4))
Traceback (most recent call last):
...
IndexError: argument i (= 4) must be between 0 and 2

```

single_vertex_all()

Return the rooted forests with one vertex in self.

They freely generate the Lie algebra of primitive elements as a pre-Lie algebra.

See also

single_vertex().

EXAMPLES:

```

sage: A = algebras.GrossmanLarson(ZZ, 'fgh')
sage: A.single_vertex_all()
(B[#[f[]]], B[#[g[]]], B[#[h[]]])

sage: A = algebras.GrossmanLarson(QQ, ['x1', 'x2'])
sage: A.single_vertex_all()
(B[#[x1[]]], B[#[x2[]]])

sage: A = algebras.GrossmanLarson(ZZ, None)
sage: A.single_vertex_all()
(B[[[]]],)

```

```

>>> from sage.all import *
>>> A = algebras.GrossmanLarson(ZZ, 'fgh')
>>> A.single_vertex_all()
(B[#[f[]]], B[#[g[]]], B[#[h[]]])

>>> A = algebras.GrossmanLarson(QQ, ['x1', 'x2'])
>>> A.single_vertex_all()
(B[#[x1[]]], B[#[x2[]]])

>>> A = algebras.GrossmanLarson(ZZ, None)
>>> A.single_vertex_all()
(B[[[]]],)

```

some_elements()

Return some elements of the Grossman-Larson Hopf algebra.

EXAMPLES:

```

sage: A = algebras.GrossmanLarson(QQ, None)
sage: A.some_elements()

```

(continues on next page)

(continued from previous page)

```
[B[[[]]], B[[]] + B[[[[]]]] + B[[[]], []],
4*B[[[[]]]] + 4*B[[[]], []]]
```

```
>>> from sage.all import *
>>> A = algebras.GrossmanLarson(QQ, None)
>>> A.some_elements()
[B[[[]]], B[[]] + B[[[[]]]] + B[[[]], []],
4*B[[[[]]]] + 4*B[[[]], []]]
```

With several generators:

```
sage: A = algebras.GrossmanLarson(QQ, 'xy')
sage: A.some_elements()
[B[#x[]],
 B[#[] + B[#x[x[]]]] + B[#x[], x[]],
 B[#x[x[]]] + 3*B[#x[y[]]] + B[#x[], x[]] + 3*B[#x[], y[]]]
```

```
>>> from sage.all import *
>>> A = algebras.GrossmanLarson(QQ, 'xy')
>>> A.some_elements()
[B[#x[]],
 B[#[] + B[#x[x[]]]] + B[#x[], x[]],
 B[#x[x[]]] + 3*B[#x[y[]]] + B[#x[], x[]] + 3*B[#x[], y[]]]
```

variable_names()

Return the names of the variables.

This returns the set E (as a family).

EXAMPLES:

```
sage: R = algebras.GrossmanLarson(QQ, 'xy')
sage: R.variable_names()
{'x', 'y'}

sage: R = algebras.GrossmanLarson(QQ, ['a', 'b'])
sage: R.variable_names()
{'a', 'b'}

sage: R = algebras.GrossmanLarson(QQ, 2)
sage: R.variable_names()
{0, 1}

sage: R = algebras.GrossmanLarson(QQ, None)
sage: R.variable_names()
{'o'}
```

```
>>> from sage.all import *
>>> R = algebras.GrossmanLarson(QQ, 'xy')
>>> R.variable_names()
{'x', 'y'}
```

(continues on next page)

(continued from previous page)

```

>>> R = algebras.GrossmanLarson(QQ, ['a', 'b'])
>>> R.variable_names()
{'a', 'b'}

>>> R = algebras.GrossmanLarson(QQ, Integer(2))
>>> R.variable_names()
{0, 1}

>>> R = algebras.GrossmanLarson(QQ, None)
>>> R.variable_names()
{'o'}

```

5.13 Möbius Algebras

```

class sage.combinat.posets.moebius_algebra.BasisAbstract (R, basis_keys=None,
                                                         element_class=None, category=None,
                                                         prefix=None, names=None, **kws)

```

Bases: `CombinatorialFreeModule`, `BindableClass`

Abstract base class for a basis.

```

class sage.combinat.posets.moebius_algebra.MoebiusAlgebra (R, L)

```

Bases: `Parent`, `UniqueRepresentation`

The Möbius algebra of a lattice.

Let L be a lattice. The *Möbius algebra* M_L was originally constructed by Solomon [Solomon67] and has a natural basis $\{E_x \mid x \in L\}$ with multiplication given by $E_x \cdot E_y = E_{x \vee y}$. Moreover this has a basis given by orthogonal idempotents $\{I_x \mid x \in L\}$ (so $I_x I_y = \delta_{xy} I_x$ where δ is the Kronecker delta) related to the natural basis by

$$I_x = \sum_{x \leq y} \mu_L(x, y) E_y,$$

where μ_L is the Möbius function of L .

Note

We use the join $\vee$ for our multiplication, whereas [Greene73] and [Etienne98] define the Möbius algebra using the meet $\wedge$. This is done for compatibility with `QuantumMoebiusAlgebra`.

REFERENCES:

```

class E (M, prefix='E')

```

Bases: `BasisAbstract`

The natural basis of a Möbius algebra.

Let E_x and E_y be basis elements of M_L for some lattice L . Multiplication is given by $E_x E_y = E_{x \vee y}$.

```

one ()

```

Return the element 1 of `self`.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: E = L.moebius_algebra(QQ).E()
sage: E.one()
E[0]
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> E = L.moebius_algebra(QQ).E()
>>> E.one()
E[0]
```

product_on_basis(x, y)

Return the product of basis elements indexed by x and y .

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: E = L.moebius_algebra(QQ).E()
sage: E.product_on_basis(5, 14)
E[15]
sage: E.product_on_basis(2, 8)
E[10]
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> E = L.moebius_algebra(QQ).E()
>>> E.product_on_basis(Integer(5), Integer(14))
E[15]
>>> E.product_on_basis(Integer(2), Integer(8))
E[10]
```

class I(M , $prefix='I'$)

Bases: *BasisAbstract*

The (orthogonal) idempotent basis of a Möbius algebra.

Let I_x and I_y be basis elements of M_L for some lattice L . Multiplication is given by $I_x I_y = \delta_{xy} I_x$ where δ_{xy} is the Kronecker delta.

one()

Return the element 1 of self.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: I = L.moebius_algebra(QQ).I()
sage: I.one()
I[0] + I[1] + I[2] + I[3] + I[4] + I[5] + I[6] + I[7] + I[8]
+ I[9] + I[10] + I[11] + I[12] + I[13] + I[14] + I[15]
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> I = L.moebius_algebra(QQ).I()
>>> I.one()
```

(continues on next page)

(continued from previous page)

```
I[0] + I[1] + I[2] + I[3] + I[4] + I[5] + I[6] + I[7] + I[8]
+ I[9] + I[10] + I[11] + I[12] + I[13] + I[14] + I[15]
```

product_on_basis(x, y)Return the product of basis elements indexed by x and y .

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: I = L.moebius_algebra(QQ).I()
sage: I.product_on_basis(5, 14)
0
sage: I.product_on_basis(2, 2)
I[2]
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> I = L.moebius_algebra(QQ).I()
>>> I.product_on_basis(Integer(5), Integer(14))
0
>>> I.product_on_basis(Integer(2), Integer(2))
I[2]
```

a_realization()Return a particular realization of *self* (the B -basis).

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: M = L.moebius_algebra(QQ)
sage: M.a_realization()
Moebius algebra of Finite lattice containing 16 elements
over Rational Field in the natural basis
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> M = L.moebius_algebra(QQ)
>>> M.a_realization()
Moebius algebra of Finite lattice containing 16 elements
over Rational Field in the natural basis
```

idempotentalias of I **lattice**()Return the defining lattice of *self*.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: M = L.moebius_algebra(QQ)
sage: M.lattice()
Finite lattice containing 16 elements
```

(continues on next page)

(continued from previous page)

```
sage: M.lattice() == L
True
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> M = L.moebius_algebra(QQ)
>>> M.lattice()
Finite lattice containing 16 elements
>>> M.lattice() == L
True
```

naturalalias of E

class sage.combinat.posets.moebius_algebra.**MoebiusAlgebraBases** (*parent_with_realization*)

Bases: *Category_realization_of_parent*

The category of bases of a Möbius algebra.

INPUT:

- base – a Möbius algebra

class **ElementMethods**

Bases: object

class **ParentMethods**

Bases: object

one()

Return the element 1 of self.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: C = L.quantum_moebius_algebra().C()
sage: all(C.one() * b == b for b in C.basis())
True
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> C = L.quantum_moebius_algebra().C()
>>> all(C.one() * b == b for b in C.basis())
True
```

product_on_basis (x, y)Return the product of basis elements indexed by x and y .

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: C = L.quantum_moebius_algebra().C()
sage: C.product_on_basis(5, 14)
q^3*C[15]
sage: C.product_on_basis(2, 8)
q^4*C[10]
```

```

>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> C = L.quantum_moebius_algebra().C()
>>> C.product_on_basis(Integer(5), Integer(14))
q^3*C[15]
>>> C.product_on_basis(Integer(2), Integer(8))
q^4*C[10]

```

super_categories()

The super categories of self.

EXAMPLES:

```

sage: from sage.combinat.posets.moebius_algebra import MoebiusAlgebraBases
sage: M = posets.BooleanLattice(4).moebius_algebra(QQ)
sage: bases = MoebiusAlgebraBases(M)
sage: bases.super_categories()
[Category of finite dimensional commutative algebras with basis over Rational_
↪Field,
Category of realizations of Moebius algebra of Finite lattice
containing 16 elements over Rational Field]

```

```

>>> from sage.all import *
>>> from sage.combinat.posets.moebius_algebra import MoebiusAlgebraBases
>>> M = posets.BooleanLattice(Integer(4)).moebius_algebra(QQ)
>>> bases = MoebiusAlgebraBases(M)
>>> bases.super_categories()
[Category of finite dimensional commutative algebras with basis over Rational_
↪Field,
Category of realizations of Moebius algebra of Finite lattice
containing 16 elements over Rational Field]

```

class sage.combinat.posets.moebius_algebra.**QuantumMoebiusAlgebra**($L, q=None$)

Bases: `Parent`, `UniqueRepresentation`

The quantum Möbius algebra of a lattice.

Let L be a lattice, and we define the *quantum Möbius algebra* $M_L(q)$ as the algebra with basis $\{E_x \mid x \in L\}$ with multiplication given by

$$E_x E_y = \sum_{z \geq a \geq x \vee y} \mu_L(a, z) q^{\text{crk } a} E_z,$$

where μ_L is the Möbius function of L and crk is the corank function (i.e., $\text{crk } a = \text{rank } L - \text{rank } a$). At $q = 1$, this reduces to the multiplication formula originally given by Solomon.

class **C**($M, \text{prefix}='C'$)

Bases: `BasisAbstract`

The characteristic basis of a quantum Möbius algebra.

The characteristic basis $\{C_x \mid x \in L\}$ of M_L for some lattice L is defined by

$$C_x = \sum_{a \geq x} P(F^x; q) E_a,$$

where $F^x = \{y \in L \mid y \geq x\}$ is the principal order filter of x and $P(F^x; q)$ is the characteristic polynomial of the (sub)poset F^x .

class **E** (*M*, *prefix*='E')

Bases: *BasisAbstract*

The natural basis of a quantum Möbius algebra.

Let E_x and E_y be basis elements of M_L for some lattice L . Multiplication is given by

$$E_x E_y = \sum_{z \geq a \geq x \vee y} \mu_L(a, z) q^{\text{crk } a} E_z,$$

where μ_L is the Möbius function of L and crk is the corank function (i.e., $\text{crk } a = \text{rank } L - \text{rank } a$).

one ()

Return the element 1 of self.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: E = L.quantum_moebius_algebra().E()
sage: all(E.one() * b == b for b in E.basis())
True
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> E = L.quantum_moebius_algebra().E()
>>> all(E.one() * b == b for b in E.basis())
True
```

product_on_basis (*x*, *y*)

Return the product of basis elements indexed by *x* and *y*.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: E = L.quantum_moebius_algebra().E()
sage: E.product_on_basis(5, 14)
E[15]
sage: E.product_on_basis(2, 8)
q^2*E[10] + (q-q^2)*E[11] + (q-q^2)*E[14] + (1-2*q+q^2)*E[15]
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> E = L.quantum_moebius_algebra().E()
>>> E.product_on_basis(Integer(5), Integer(14))
E[15]
>>> E.product_on_basis(Integer(2), Integer(8))
q^2*E[10] + (q-q^2)*E[11] + (q-q^2)*E[14] + (1-2*q+q^2)*E[15]
```

class **KL** (*M*, *prefix*='KL')

Bases: *BasisAbstract*

The Kazhdan-Lusztig basis of a quantum Möbius algebra.

The Kazhdan-Lusztig basis $\{B_x \mid x \in L\}$ of M_L for some lattice L is defined by

$$B_x = \sum_{y \geq x} P_{x,y}(q) E_a,$$

where $P_{x,y}(q)$ is the Kazhdan-Lusztig polynomial of L , following the definition given in [EPW14].

EXAMPLES:

We construct some examples of Proposition 4.5 of [EPW14]:

```
sage: M = posets.BooleanLattice(4).quantum_moebius_algebra()
sage: KL = M.KL()
sage: KL[4] * KL[5]
(q^2+q^3)*KL[5] + (q+2*q^2+q^3)*KL[7] + (q+2*q^2+q^3)*KL[13]
+ (1+3*q+3*q^2+q^3)*KL[15]
sage: KL[4] * KL[15]
(1+3*q+3*q^2+q^3)*KL[15]
sage: KL[4] * KL[10]
(q+3*q^2+3*q^3+q^4)*KL[14] + (1+4*q+6*q^2+4*q^3+q^4)*KL[15]
```

```
>>> from sage.all import *
>>> M = posets.BooleanLattice(Integer(4)).quantum_moebius_algebra()
>>> KL = M.KL()
>>> KL[Integer(4)] * KL[Integer(5)]
(q^2+q^3)*KL[5] + (q+2*q^2+q^3)*KL[7] + (q+2*q^2+q^3)*KL[13]
+ (1+3*q+3*q^2+q^3)*KL[15]
>>> KL[Integer(4)] * KL[Integer(15)]
(1+3*q+3*q^2+q^3)*KL[15]
>>> KL[Integer(4)] * KL[Integer(10)]
(q+3*q^2+3*q^3+q^4)*KL[14] + (1+4*q+6*q^2+4*q^3+q^4)*KL[15]
```

a_realization()

Return a particular realization of `self` (the B -basis).

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: M = L.quantum_moebius_algebra()
sage: M.a_realization()
Quantum Moebius algebra of Finite lattice containing 16 elements
with q=q over Univariate Laurent Polynomial Ring in q
over Integer Ring in the natural basis
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> M = L.quantum_moebius_algebra()
>>> M.a_realization()
Quantum Moebius algebra of Finite lattice containing 16 elements
with q=q over Univariate Laurent Polynomial Ring in q
over Integer Ring in the natural basis
```

characteristic_basis

alias of C

kazhdan_lusztig

alias of KL

lattice()

Return the defining lattice of `self`.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: M = L.quantum_moebius_algebra()
sage: M.lattice()
Finite lattice containing 16 elements
sage: M.lattice() == L
True
```

```
>>> from sage.all import *
>>> L = posets.BooleanLattice(Integer(4))
>>> M = L.quantum_moebius_algebra()
>>> M.lattice()
Finite lattice containing 16 elements
>>> M.lattice() == L
True
```

natural

alias of *E*

5.14 Orlik-Terao Algebras

class sage.algebras.orlik_terao.**OrlikTeraoAlgebra** (*R*, *M*, *ordering=None*)

Bases: `CombinatorialFreeModule`

An Orlik-Terao algebra.

Let R be a commutative ring. Let M be a matroid with groundset X with some fixed ordering and representation $A = (a_x)_{x \in X}$ (so a_x is a (column) vector). Let $C(M)$ denote the set of circuits of M . Let P denote the quotient algebra $R[e_x \mid x \in X] / \langle e_x^2 \rangle$, i.e., the polynomial algebra with squares being zero. The *Orlik-Terao ideal* $J(M)$ is the ideal of P generated by

$$\partial e_S := \sum_{i=1}^t (-1)^i \chi(S \setminus \{j_i\}) e_{S \setminus \{j_i\}}$$

for all $S = \{j_1 < j_2 < \dots < j_t\} \in C(M)$, where $\chi(T)$ is defined as follows. If T is linearly dependent, then $\chi(T) = 0$. Otherwise, let $T = \{x_1 < \dots < x_{|T|}\}$, and for every flat F of M , choose a basis Θ_F . Then define $\chi(T) = \det(b_1, \dots, b_{|T|})$, where b_i is a_{x_i} expressed in the basis Θ_F .

It is easy to see that $\partial e_S \in J(M)$ not only for circuits S , but also for any dependent set S of M . Moreover, every dependent set S of M satisfies $e_S \in J(M)$.

The *Orlik-Terao algebra* $A(M)$ is the quotient $E/J(M)$. This is a graded finite-dimensional commutative R -algebra. The non-broken circuit (NBC) sets of M (that is, the subsets of X containing no broken circuit of M) form a basis of $A(M)$. (Recall that a broken circuit of M is defined to be the result of removing the smallest element from a circuit of M .)

In the current implementation, the basis of $A(M)$ is indexed by the NBC sets, which are implemented as frozensets.

INPUT:

- *R* – the base ring
- *M* – the defining matroid
- *ordering* – (optional) an ordering of the groundset

EXAMPLES:

We create the Orlik-Terao algebra of the wheel matroid $W(3)$ and do some basic computations:

```
sage: M = matroids.Wheel(3)
sage: OT = M.orlik_terao_algebra(QQ)
sage: OT.dimension()
24
sage: G = OT.algebra_generators()
sage: sorted(map(sorted, M.broken_circuits()))
[[1, 3], [1, 4, 5], [2, 3, 4], [2, 3, 5], [2, 4], [2, 5], [4, 5]]
sage: G[1] * G[2] * G[3]
OT{0, 1, 2} + OT{0, 2, 3}
sage: G[1] * G[4] * G[5]
-OT{0, 1, 4} - OT{0, 1, 5} - OT{0, 3, 4} - OT{0, 3, 5}
```

```
>>> from sage.all import *
>>> M = matroids.Wheel(Integer(3))
>>> OT = M.orlik_terao_algebra(QQ)
>>> OT.dimension()
24
>>> G = OT.algebra_generators()
>>> sorted(map(sorted, M.broken_circuits()))
[[1, 3], [1, 4, 5], [2, 3, 4], [2, 3, 5], [2, 4], [2, 5], [4, 5]]
>>> G[Integer(1)] * G[Integer(2)] * G[Integer(3)]
OT{0, 1, 2} + OT{0, 2, 3}
>>> G[Integer(1)] * G[Integer(4)] * G[Integer(5)]
-OT{0, 1, 4} - OT{0, 1, 5} - OT{0, 3, 4} - OT{0, 3, 5}
```

We create an example of a linear matroid and do a basic computation:

```
sage: R = ZZ['t'].fraction_field()
sage: t = R.gen()
sage: mat = matrix(R, [[1-3*t/(t+2), t, 5], [-2, 1, 3/(7-t)]])
sage: M = Matroid(mat)
sage: OT = M.orlik_terao_algebra()
sage: G = OT.algebra_generators()
sage: G[1] * G[2]
((2*t^3-12*t^2-12*t-14)/(8*t^2-19*t-70))*OT{0, 1}
+ ((10*t^2-44*t-146)/(-8*t^2+19*t+70))*OT{0, 2}
```

```
>>> from sage.all import *
>>> R = ZZ['t'].fraction_field()
>>> t = R.gen()
>>> mat = matrix(R, [[Integer(1)-Integer(3)*t/(t+Integer(2)), t, Integer(5)], [-
Integer(2), Integer(1), Integer(3)/(Integer(7)-t)]])
>>> M = Matroid(mat)
>>> OT = M.orlik_terao_algebra()
>>> G = OT.algebra_generators()
>>> G[Integer(1)] * G[Integer(2)]
((2*t^3-12*t^2-12*t-14)/(8*t^2-19*t-70))*OT{0, 1}
+ ((10*t^2-44*t-146)/(-8*t^2+19*t+70))*OT{0, 2}
```

REFERENCES:

- [OT1994]
- [FL2001]
- [CF2005]

algebra_generators()

Return the algebra generators of `self`.

These form a family indexed by the groundset X of M . For each $x \in X$, the x -th element is e_x .

EXAMPLES:

```
sage: M = matroids.Whirl(2)
sage: OT = M.orlik_terao_algebra()
sage: OT.algebra_generators()
Finite family {0: OT{0}, 1: OT{1}, 2: OT{2}, 3: OT{3}}

sage: M = matroids.catalog.Fano()
sage: OT = M.orlik_terao_algebra()
sage: OT.algebra_generators()
Finite family {'a': OT{a}, 'b': OT{b}, 'c': OT{c}, 'd': OT{d},
              'e': OT{e}, 'f': OT{f}, 'g': OT{g}}

sage: M = matroids.catalog.NonFano()
sage: OT = M.orlik_terao_algebra(GF(3) ['t'])
sage: OT.algebra_generators()
Finite family {'a': OT{a}, 'b': OT{b}, 'c': OT{c}, 'd': OT{d},
              'e': OT{e}, 'f': OT{f}, 'g': OT{g}}
```

```
>>> from sage.all import *
>>> M = matroids.Whirl(Integer(2))
>>> OT = M.orlik_terao_algebra()
>>> OT.algebra_generators()
Finite family {0: OT{0}, 1: OT{1}, 2: OT{2}, 3: OT{3}}

>>> M = matroids.catalog.Fano()
>>> OT = M.orlik_terao_algebra()
>>> OT.algebra_generators()
Finite family {'a': OT{a}, 'b': OT{b}, 'c': OT{c}, 'd': OT{d},
              'e': OT{e}, 'f': OT{f}, 'g': OT{g}}

>>> M = matroids.catalog.NonFano()
>>> OT = M.orlik_terao_algebra(GF(Integer(3)) ['t'])
>>> OT.algebra_generators()
Finite family {'a': OT{a}, 'b': OT{b}, 'c': OT{c}, 'd': OT{d},
              'e': OT{e}, 'f': OT{f}, 'g': OT{g}}
```

degree_on_basis(m)

Return the degree of the basis element indexed by `m`.

EXAMPLES:

```
sage: M = matroids.Wheel(3)
sage: OT = M.orlik_terao_algebra(QQ)
sage: OT.degree_on_basis(frozenset([1]))
```

(continues on next page)

(continued from previous page)

```

1
sage: OT.degree_on_basis(frozenset([0, 2, 3]))
3

```

```

>>> from sage.all import *
>>> M = matroids.Wheel(Integer(3))
>>> OT = M.orlik_terao_algebra(QQ)
>>> OT.degree_on_basis(frozenset([Integer(1)]))
1
>>> OT.degree_on_basis(frozenset([Integer(0), Integer(2), Integer(3)]))
3

```

one_basis()

Return the index of the basis element corresponding to 1 in self.

EXAMPLES:

```

sage: M = matroids.Wheel(3)
sage: OT = M.orlik_terao_algebra(QQ)
sage: OT.one_basis() == frozenset()
True

```

```

>>> from sage.all import *
>>> M = matroids.Wheel(Integer(3))
>>> OT = M.orlik_terao_algebra(QQ)
>>> OT.one_basis() == frozenset()
True

```

product_on_basis(a, b)

Return the product in self of the basis elements indexed by a and b.

EXAMPLES:

```

sage: M = matroids.Wheel(3)
sage: OT = M.orlik_terao_algebra(QQ)
sage: OT.product_on_basis(frozenset([2]), frozenset([3, 4]))
OT{0, 1, 2} + OT{0, 1, 4} + OT{0, 2, 3} + OT{0, 3, 4}

```

```

>>> from sage.all import *
>>> M = matroids.Wheel(Integer(3))
>>> OT = M.orlik_terao_algebra(QQ)
>>> OT.product_on_basis(frozenset([Integer(2)]), frozenset([Integer(3),
↳ Integer(4)]))
OT{0, 1, 2} + OT{0, 1, 4} + OT{0, 2, 3} + OT{0, 3, 4}

```

```

sage: G = OT.algebra_generators()
sage: prod(G)
0
sage: G[2] * G[4]
OT{1, 2} + OT{1, 4}
sage: G[3] * G[4] * G[2]
OT{0, 1, 2} + OT{0, 1, 4} + OT{0, 2, 3} + OT{0, 3, 4}

```

(continues on next page)

(continued from previous page)

```
sage: G[2] * G[3] * G[4]
OT{0, 1, 2} + OT{0, 1, 4} + OT{0, 2, 3} + OT{0, 3, 4}
sage: G[3] * G[2] * G[4]
OT{0, 1, 2} + OT{0, 1, 4} + OT{0, 2, 3} + OT{0, 3, 4}
```

```
>>> from sage.all import *
>>> G = OT.algebra_generators()
>>> prod(G)
0
>>> G[Integer(2)] * G[Integer(4)]
OT{1, 2} + OT{1, 4}
>>> G[Integer(3)] * G[Integer(4)] * G[Integer(2)]
OT{0, 1, 2} + OT{0, 1, 4} + OT{0, 2, 3} + OT{0, 3, 4}
>>> G[Integer(2)] * G[Integer(3)] * G[Integer(4)]
OT{0, 1, 2} + OT{0, 1, 4} + OT{0, 2, 3} + OT{0, 3, 4}
>>> G[Integer(3)] * G[Integer(2)] * G[Integer(4)]
OT{0, 1, 2} + OT{0, 1, 4} + OT{0, 2, 3} + OT{0, 3, 4}
```

subset_image(S)

Return the element e_S of self corresponding to a subset S of the groundset of the defining matroid.

INPUT:

- S – frozenset which is a subset of the groundset of M

EXAMPLES:

```
sage: M = matroids.Wheel(3)
sage: OT = M.orlik_terao_algebra()
sage: BC = sorted(M.broken_circuits(), key=sorted)
sage: for bc in BC: (sorted(bc), OT.subset_image(bc))
([1, 3], OT{0, 1} + OT{0, 3})
([1, 4, 5], -OT{0, 1, 4} - OT{0, 1, 5} - OT{0, 3, 4} - OT{0, 3, 5})
([2, 3, 4], OT{0, 1, 2} + OT{0, 1, 4} + OT{0, 2, 3} + OT{0, 3, 4})
([2, 3, 5], -OT{0, 2, 3} + OT{0, 3, 5})
([2, 4], OT{1, 2} + OT{1, 4})
([2, 5], -OT{0, 2} + OT{0, 5})
([4, 5], -OT{3, 4} - OT{3, 5})

sage: # needs sage.graphs
sage: M4 = matroids.CompleteGraphic(4).ternary_matroid()
sage: OT = M4.orlik_terao_algebra()
sage: OT.subset_image(frozenset({2, 3, 4}))
OT{0, 2, 3} + 2*OT{0, 3, 4}
```

```
>>> from sage.all import *
>>> M = matroids.Wheel(Integer(3))
>>> OT = M.orlik_terao_algebra()
>>> BC = sorted(M.broken_circuits(), key=sorted)
>>> for bc in BC: (sorted(bc), OT.subset_image(bc))
([1, 3], OT{0, 1} + OT{0, 3})
([1, 4, 5], -OT{0, 1, 4} - OT{0, 1, 5} - OT{0, 3, 4} - OT{0, 3, 5})
([2, 3, 4], OT{0, 1, 2} + OT{0, 1, 4} + OT{0, 2, 3} + OT{0, 3, 4})
```

(continues on next page)

(continued from previous page)

```

([2, 3, 5], -OT{0, 2, 3} + OT{0, 3, 5})
([2, 4], OT{1, 2} + OT{1, 4})
([2, 5], -OT{0, 2} + OT{0, 5})
([4, 5], -OT{3, 4} - OT{3, 5})

>>> # needs sage.graphs
>>> M4 = matroids.CompleteGraphic(Integer(4)).ternary_matroid()
>>> OT = M4.orlik_terao_algebra()
>>> OT.subset_image(frozenset({Integer(2), Integer(3), Integer(4)}))
OT{0, 2, 3} + 2*OT{0, 3, 4}

```

An example of a custom ordering:

```

sage: # needs sage.graphs
sage: G = Graph([[3, 4], [4, 1], [1, 2], [2, 3], [3, 5], [5, 6], [6, 3]])
sage: M = Matroid(G).regular_matroid()
sage: s = [(5, 6), (1, 2), (3, 5), (2, 3), (1, 4), (3, 6), (3, 4)]
sage: sorted([sorted(c) for c in M.circuits()])
[[ (1, 2), (1, 4), (2, 3), (3, 4)],
 [ (3, 5), (3, 6), (5, 6)]]
sage: OT = M.orlik_terao_algebra(QQ, ordering=s)
sage: OT.subset_image(frozenset())
OT{}
sage: OT.subset_image(frozenset([(1, 2), (3, 4), (1, 4), (2, 3)]))
0
sage: OT.subset_image(frozenset([(2, 3), (1, 2), (3, 4)]))
OT{(1, 2), (2, 3), (3, 4)}
sage: OT.subset_image(frozenset([(1, 4), (3, 4), (2, 3), (3, 6), (5, 6)]))
-OT{(1, 2), (1, 4), (2, 3), (3, 6), (5, 6)}
- OT{(1, 2), (1, 4), (3, 4), (3, 6), (5, 6)}
+ OT{(1, 2), (2, 3), (3, 4), (3, 6), (5, 6)}
sage: OT.subset_image(frozenset([(1, 4), (3, 4), (2, 3), (3, 6), (3, 5)]))
-OT{(1, 2), (1, 4), (2, 3), (3, 5), (5, 6)}
+ OT{(1, 2), (1, 4), (2, 3), (3, 6), (5, 6)}
- OT{(1, 2), (1, 4), (3, 4), (3, 5), (5, 6)}
+ OT{(1, 2), (1, 4), (3, 4), (3, 6), (5, 6)}
+ OT{(1, 2), (2, 3), (3, 4), (3, 5), (5, 6)}
- OT{(1, 2), (2, 3), (3, 4), (3, 6), (5, 6)}

```

```

>>> from sage.all import *
>>> # needs sage.graphs
>>> G = Graph([[Integer(3), Integer(4)], [Integer(4), Integer(1)],
↳ [Integer(1), Integer(2)], [Integer(2), Integer(3)], [Integer(3),
↳ Integer(5)], [Integer(5), Integer(6)], [Integer(6), Integer(3)]])
>>> M = Matroid(G).regular_matroid()
>>> s = [(Integer(5), Integer(6)), (Integer(1), Integer(2)), (Integer(3),
↳ Integer(5)), (Integer(2), Integer(3)), (Integer(1), Integer(4)),
↳ (Integer(3), Integer(6)), (Integer(3), Integer(4))]
>>> sorted([sorted(c) for c in M.circuits()])
[[ (1, 2), (1, 4), (2, 3), (3, 4)],
 [ (3, 5), (3, 6), (5, 6)]]
>>> OT = M.orlik_terao_algebra(QQ, ordering=s)

```

(continues on next page)

(continued from previous page)

```

>>> OT.subset_image(frozenset())
OT{}
>>> OT.subset_image(frozenset([(Integer(1), Integer(2)), (Integer(3),
↪ Integer(4)), (Integer(1), Integer(4)), (Integer(2), Integer(3))]))
0
>>> OT.subset_image(frozenset([(Integer(2), Integer(3)), (Integer(1),
↪ Integer(2)), (Integer(3), Integer(4))]))
OT{(1, 2), (2, 3), (3, 4)}
>>> OT.subset_image(frozenset([(Integer(1), Integer(4)), (Integer(3),
↪ Integer(4)), (Integer(2), Integer(3)), (Integer(3), Integer(6)), (Integer(5),
↪ Integer(6))]))
-OT{(1, 2), (1, 4), (2, 3), (3, 6), (5, 6)}
- OT{(1, 2), (1, 4), (3, 4), (3, 6), (5, 6)}
+ OT{(1, 2), (2, 3), (3, 4), (3, 6), (5, 6)}
>>> OT.subset_image(frozenset([(Integer(1), Integer(4)), (Integer(3),
↪ Integer(4)), (Integer(2), Integer(3)), (Integer(3), Integer(6)), (Integer(3),
↪ Integer(5))]))
-OT{(1, 2), (1, 4), (2, 3), (3, 5), (5, 6)}
+ OT{(1, 2), (1, 4), (2, 3), (3, 6), (5, 6)}
- OT{(1, 2), (1, 4), (3, 4), (3, 5), (5, 6)}
+ OT{(1, 2), (1, 4), (3, 4), (3, 6), (5, 6)}
+ OT{(1, 2), (2, 3), (3, 4), (3, 5), (5, 6)}
- OT{(1, 2), (2, 3), (3, 4), (3, 6), (5, 6)}

```

class sage.algebras.orlik_terao.**OrlikTeraoInvariantAlgebra**(*R, M, G, action_on_groundset=None, *args, **kwargs*)

Bases: `FiniteDimensionalInvariantModule`

Give the invariant algebra of the Orlik-Terao algebra from the action on $A(M)$ which is induced from the `action_on_groundset`.

INPUT:

- *R* – the ring of coefficients
- *M* – a matroid
- *G* – a semigroup
- `action_on_groundset` – a function defining the action of *G* on the elements of the groundset of *M* default

OUTPUT:

- The invariant algebra of the Orlik-Terao algebra induced by the action of `action_on_groundset`

EXAMPLES:

Lets start with the action of S_3 on the rank-2 braid matroid:

```

sage: A = matrix([[1, 1, 0], [-1, 0, 1], [0, -1, -1]])
sage: M = Matroid(A)
sage: M.groundset()
frozenset({0, 1, 2})
sage: G = SymmetricGroup(3)
↪needs sage.groups

```

#_

```

>>> from sage.all import *
>>> A = matrix([[Integer(1), Integer(1), Integer(0)], [-Integer(1), Integer(0),
↳ Integer(1)], [Integer(0), -Integer(1), -Integer(1)]])
>>> M = Matroid(A)
>>> M.groundset()
frozenset({0, 1, 2})
>>> G = SymmetricGroup(Integer(3))
↳ # needs sage.groups

```

Calling elements g of G on an element i of $\{1, 2, 3\}$ defines the action we want, but since the groundset is $\{0, 1, 2\}$ we first add 1 and then subtract 1:

```

sage: def on_groundset(g, x):
.....:     return g(x+1)-1

```

```

>>> from sage.all import *
>>> def on_groundset(g, x):
...     return g(x+Integer(1))-Integer(1)

```

Now that we have defined an action we can create the invariant, and get its basis:

```

sage: # needs sage.groups
sage: OTG = M.orlik_terao_algebra(QQ, invariant=(G, on_groundset))
sage: OTG.basis()
Finite family {0: B[0], 1: B[1]}
sage: [OTG.lift(b) for b in OTG.basis()]
[OT{ }, OT{0} + OT{1} + OT{2}]

```

```

>>> from sage.all import *
>>> # needs sage.groups
>>> OTG = M.orlik_terao_algebra(QQ, invariant=(G, on_groundset))
>>> OTG.basis()
Finite family {0: B[0], 1: B[1]}
>>> [OTG.lift(b) for b in OTG.basis()]
[OT{ }, OT{0} + OT{1} + OT{2}]

```

Since it is invariant, the action of any g in G is trivial:

```

sage: # needs sage.groups
sage: x = OTG.an_element(); x
2*B[0] + 2*B[1]
sage: g = G.an_element(); g
(2, 3)
sage: g*x
2*B[0] + 2*B[1]

sage: # needs sage.groups
sage: x = OTG.random_element()
sage: g = G.random_element()
sage: g*x == x
True

```

```

>>> from sage.all import *
>>> # needs sage.groups
>>> x = OTG.an_element(); x
2*B[0] + 2*B[1]
>>> g = G.an_element(); g
(2, 3)
>>> g*x
2*B[0] + 2*B[1]

>>> # needs sage.groups
>>> x = OTG.random_element()
>>> g = G.random_element()
>>> g*x == x
True

```

The underlying ambient module is the Orlik-Terao algebra, which is accessible via `ambient()`:

```

sage: M.orlik_terao_algebra(QQ) is OTG.ambient()
↪needs sage.groups
True

```

```

>>> from sage.all import *
>>> M.orlik_terao_algebra(QQ) is OTG.ambient()
↪needs sage.groups
True

```

For a bigger example, here we will look at the rank-3 braid matroid:

```

sage: # needs sage.groups
sage: A = matrix([[1, 1, 1, 0, 0, 0], [-1, 0, 0, 1, 1, 0],
....:             [0, -1, 0, -1, 0, 1], [0, 0, -1, 0, -1, -1]]); A
[ 1  1  1  0  0  0]
[-1  0  0  1  1  0]
[ 0 -1  0 -1  0  1]
[ 0  0 -1  0 -1 -1]
sage: M = Matroid(A); M.groundset()
frozenset({0, 1, 2, 3, 4, 5})
sage: G = SymmetricGroup(6)
sage: OTG = M.orlik_terao_algebra(QQ, invariant=(G, on_groundset))
sage: OTG.ambient()
Orlik-Terao algebra of
Linear matroid of rank 3 on 6 elements represented over the Rational Field
over Rational Field
sage: OTG.basis()
Finite family {0: B[0], 1: B[1]}
sage: [OTG.lift(b) for b in OTG.basis()]
[OT{ }, OT{0} + OT{1} + OT{2} + OT{3} + OT{4} + OT{5}]

```

```

>>> from sage.all import *
>>> # needs sage.groups
>>> A = matrix([[Integer(1), Integer(1), Integer(1), Integer(0), Integer(0),
↪Integer(0)], [-Integer(1), Integer(0), Integer(0), Integer(1), Integer(1),
↪Integer(0)],

```

(continues on next page)

(continued from previous page)

```

...      [Integer(0), -Integer(1), Integer(0), -Integer(1), Integer(0),
→ Integer(1)], [Integer(0), Integer(0), -Integer(1), Integer(0), -Integer(1), -
→ Integer(1)]]; A
[ 1  1  1  0  0  0]
[-1  0  0  1  1  0]
[ 0 -1  0 -1  0  1]
[ 0  0 -1  0 -1 -1]
>>> M = Matroid(A); M.groundset()
frozenset({0, 1, 2, 3, 4, 5})
>>> G = SymmetricGroup(Integer(6))
>>> OTG = M.orlik_terao_algebra(QQ, invariant=(G, on_groundset))
>>> OTG.ambient()
Orlik-Terao algebra of
  Linear matroid of rank 3 on 6 elements represented over the Rational Field
  over Rational Field
>>> OTG.basis()
Finite family {0: B[0], 1: B[1]}
>>> [OTG.lift(b) for b in OTG.basis()]
[OT{ }, OT{0} + OT{1} + OT{2} + OT{3} + OT{4} + OT{5}]

```

construction()Return the functorial construction of `self`.This implementation of the method only returns `None`.

5.15 Orlik-Solomon Algebras

class sage.algebras.orlik_solomon.**OrlikSolomonAlgebra** (*R, M, ordering=None*)

Bases: `CombinatorialFreeModule`

An Orlik-Solomon algebra.

Let R be a commutative ring. Let M be a matroid with groundset X . Let $C(M)$ denote the set of circuits of M . Let E denote the exterior algebra over R generated by $\{e_x \mid x \in X\}$. The *Orlik-Solomon ideal* $J(M)$ is the ideal of E generated by

$$\partial e_S := \sum_{i=1}^t (-1)^{i-1} e_{j_1} \wedge e_{j_2} \wedge \cdots \wedge \widehat{e_{j_i}} \wedge \cdots \wedge e_{j_t}$$

for all $S = \{j_1 < j_2 < \cdots < j_t\} \in C(M)$, where $\widehat{e_{j_i}}$ means that the term e_{j_i} is being omitted. The notation ∂e_S is not a coincidence, as ∂e_S is actually the image of $e_S := e_{j_1} \wedge e_{j_2} \wedge \cdots \wedge e_{j_t}$ under the unique derivation ∂ of E which sends all e_x to 1.

It is easy to see that $\partial e_S \in J(M)$ not only for circuits S , but also for any dependent set S of M . Moreover, every dependent set S of M satisfies $e_S \in J(M)$.

The *Orlik-Solomon algebra* $A(M)$ is the quotient $E/J(M)$. This is a graded finite-dimensional skew-commutative R -algebra. Fix some ordering on X ; then, the NBC sets of M (that is, the subsets of X containing no broken circuit of M) form a basis of $A(M)$. (Here, a *broken circuit* of M is defined to be the result of removing the smallest element from a circuit of M .)

In the current implementation, the basis of $A(M)$ is indexed by the NBC sets, which are implemented as frozensets.

INPUT:

- R – the base ring

- `M` – the defining matroid
- `ordering` – (optional) an ordering of the groundset

EXAMPLES:

We create the Orlik-Solomon algebra of the uniform matroid $U(3, 4)$ and do some basic computations:

```
sage: M = matroids.Uniform(3, 4)
sage: OS = M.orlik_solomon_algebra(QQ)
sage: OS.dimension()
14
sage: G = OS.algebra_generators()
sage: M.broken_circuits()
SetSystem of 1 sets over 4 elements
sage: M.broken_circuits()[0]
frozenset({1, 2, 3})
sage: G[1] * G[2] * G[3]
OS{0, 1, 2} - OS{0, 1, 3} + OS{0, 2, 3}
```

```
>>> from sage.all import *
>>> M = matroids.Uniform(Integer(3), Integer(4))
>>> OS = M.orlik_solomon_algebra(QQ)
>>> OS.dimension()
14
>>> G = OS.algebra_generators()
>>> M.broken_circuits()
SetSystem of 1 sets over 4 elements
>>> M.broken_circuits()[Integer(0)]
frozenset({1, 2, 3})
>>> G[Integer(1)] * G[Integer(2)] * G[Integer(3)]
OS{0, 1, 2} - OS{0, 1, 3} + OS{0, 2, 3}
```

REFERENCES:

- [Wikipedia article Arrangement_of_hyperplanes#The_Orlik-Solomon_algebra](#)
- [CE2001]

algebra_generators()

Return the algebra generators of `self`.

These form a family indexed by the groundset X of M . For each $x \in X$, the x -th element is e_x .

EXAMPLES:

```
sage: M = matroids.Uniform(2, 2)
sage: OS = M.orlik_solomon_algebra(QQ)
sage: OS.algebra_generators()
Finite family {0: OS{0}, 1: OS{1}}

sage: M = matroids.Uniform(1, 2)
sage: OS = M.orlik_solomon_algebra(QQ)
sage: OS.algebra_generators()
Finite family {0: OS{0}, 1: OS{0}}

sage: M = matroids.Uniform(1, 3)
```

(continues on next page)

(continued from previous page)

```
sage: OS = M.orlik_solomon_algebra(QQ)
sage: OS.algebra_generators()
Finite family {0: OS{0}, 1: OS{0}, 2: OS{0}}
```

```
>>> from sage.all import *
>>> M = matroids.Uniform(Integer(2), Integer(2))
>>> OS = M.orlik_solomon_algebra(QQ)
>>> OS.algebra_generators()
Finite family {0: OS{0}, 1: OS{1}}

>>> M = matroids.Uniform(Integer(1), Integer(2))
>>> OS = M.orlik_solomon_algebra(QQ)
>>> OS.algebra_generators()
Finite family {0: OS{0}, 1: OS{0}}

>>> M = matroids.Uniform(Integer(1), Integer(3))
>>> OS = M.orlik_solomon_algebra(QQ)
>>> OS.algebra_generators()
Finite family {0: OS{0}, 1: OS{0}, 2: OS{0}}
```

aomoto_complex(*omega*)

Return the Aomoto complex of *self* defined by *omega*.

Let $A(M)$ be an Orlik-Solomon algebra of a matroid M . Let $\omega \in A(M)_1$ be an element of (homogeneous) degree 1. The Aomoto complex is the chain complex defined on $A(M)$ with the differential defined by $\omega \wedge$.

EXAMPLES:

```
sage: OS = hyperplane_arrangements.braid(3).orlik_solomon_algebra(QQ)
sage: gens = OS.algebra_generators()
sage: AC = OS.aomoto_complex(gens[0])
sage: ascii_art(AC)

          [0]
    [1 0 0] [0]
    [0 1 0] [1]
0 <-- C_2 <----- C_1 <---- C_0 <-- 0
sage: AC.homology()
{0: Vector space of dimension 0 over Rational Field,
 1: Vector space of dimension 0 over Rational Field,
 2: Vector space of dimension 0 over Rational Field}

sage: AC = OS.aomoto_complex(-2*gens[0] + gens[1] + gens[2]); ascii_art(AC)

          [ 1]
    [-1 -1 -1] [ 1]
    [-1 -1 -1] [-2]
0 <-- C_2 <----- C_1 <---- C_0 <-- 0
sage: AC.homology()
{0: Vector space of dimension 0 over Rational Field,
 1: Vector space of dimension 1 over Rational Field,
 2: Vector space of dimension 1 over Rational Field}
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```

>>> OS = hyperplane_arrangements.braid(Integer(3)).orlik_solomon_algebra(QQ)
>>> gens = OS.algebra_generators()
>>> AC = OS.aomoto_complex(gens[Integer(0)])
>>> ascii_art(AC)
          [0]
      [1 0 0] [0]
      [0 1 0] [1]
0 <-- C_2 <----- C_1 <---- C_0 <-- 0
>>> AC.homology()
{0: Vector space of dimension 0 over Rational Field,
 1: Vector space of dimension 0 over Rational Field,
 2: Vector space of dimension 0 over Rational Field}

>>> AC = OS.aomoto_complex(-Integer(2)*gens[Integer(0)] + gens[Integer(1)] +
↪gens[Integer(2)]); ascii_art(AC)
          [ 1]
      [-1 -1 -1] [ 1]
      [-1 -1 -1] [-2]
0 <-- C_2 <----- C_1 <---- C_0 <-- 0
>>> AC.homology()
{0: Vector space of dimension 0 over Rational Field,
 1: Vector space of dimension 1 over Rational Field,
 2: Vector space of dimension 1 over Rational Field}

```

REFERENCES:

- [BY2016]

as_cdga()

Return the commutative differential graded algebra corresponding to `self` with the trivial differential.

EXAMPLES:

```

sage: # needs sage.geometry.polyhedron sage.graphs
sage: H = hyperplane_arrangements.braid(3)
sage: O = H.orlik_solomon_algebra(QQ)
sage: O.as_cdga()
Commutative Differential Graded Algebra with generators ('e0', 'e1', 'e2')
in degrees (1, 1, 1) with relations [e0*e1 - e0*e2 + e1*e2] over Rational
↪Field
with differential:
  e0 --> 0
  e1 --> 0
  e2 --> 0

```

```

>>> from sage.all import *
>>> # needs sage.geometry.polyhedron sage.graphs
>>> H = hyperplane_arrangements.braid(Integer(3))
>>> O = H.orlik_solomon_algebra(QQ)
>>> O.as_cdga()
Commutative Differential Graded Algebra with generators ('e0', 'e1', 'e2')
in degrees (1, 1, 1) with relations [e0*e1 - e0*e2 + e1*e2] over Rational
↪Field

```

(continues on next page)

(continued from previous page)

```

with differential:
    e0 --> 0
    e1 --> 0
    e2 --> 0

```

as_gca()

Return the graded commutative algebra corresponding to self.

EXAMPLES:

```

sage: # needs sage.geometry.polyhedron sage.graphs
sage: H = hyperplane_arrangements.braid(3)
sage: O = H.orlik_solomon_algebra(QQ)
sage: O.as_gca()
Graded Commutative Algebra with generators ('e0', 'e1', 'e2') in degrees (1,
↪1, 1)
with relations [e0*e1 - e0*e2 + e1*e2] over Rational Field

```

```

>>> from sage.all import *
>>> # needs sage.geometry.polyhedron sage.graphs
>>> H = hyperplane_arrangements.braid(Integer(3))
>>> O = H.orlik_solomon_algebra(QQ)
>>> O.as_gca()
Graded Commutative Algebra with generators ('e0', 'e1', 'e2') in degrees (1,
↪1, 1)
with relations [e0*e1 - e0*e2 + e1*e2] over Rational Field

```

```

sage: N = matroids.catalog.Fano()
sage: O = N.orlik_solomon_algebra(QQ)
sage: O.as_gca()
↪needs sage.libs.singular
Graded Commutative Algebra with generators ('e0', 'e1', 'e2', 'e3', 'e4', 'e5',
↪', 'e6')
in degrees (1, 1, 1, 1, 1, 1, 1) with relations
[e1*e2 - e1*e3 + e2*e3, e0*e1*e3 - e0*e1*e4 + e0*e3*e4 - e1*e3*e4,
e0*e2 - e0*e4 + e2*e4, e3*e4 - e3*e5 + e4*e5,
e1*e2*e4 - e1*e2*e5 + e1*e4*e5 - e2*e4*e5,
e0*e2*e3 - e0*e2*e5 + e0*e3*e5 - e2*e3*e5, e0*e1 - e0*e5 + e1*e5,
e2*e5 - e2*e6 + e5*e6, e1*e3*e5 - e1*e3*e6 + e1*e5*e6 - e3*e5*e6,
e0*e4*e5 - e0*e4*e6 + e0*e5*e6 - e4*e5*e6, e1*e4 - e1*e6 + e4*e6,
e2*e3*e4 - e2*e3*e6 + e2*e4*e6 - e3*e4*e6, e0*e3 - e0*e6 + e3*e6,
e0*e1*e2 - e0*e1*e6 + e0*e2*e6 - e1*e2*e6] over Rational Field

```

```

>>> from sage.all import *
>>> N = matroids.catalog.Fano()
>>> O = N.orlik_solomon_algebra(QQ)
>>> O.as_gca()
↪needs sage.libs.singular
Graded Commutative Algebra with generators ('e0', 'e1', 'e2', 'e3', 'e4', 'e5',
↪', 'e6')
in degrees (1, 1, 1, 1, 1, 1, 1) with relations
[e1*e2 - e1*e3 + e2*e3, e0*e1*e3 - e0*e1*e4 + e0*e3*e4 - e1*e3*e4,

```

(continues on next page)

(continued from previous page)

```

e0*e2 - e0*e4 + e2*e4, e3*e4 - e3*e5 + e4*e5,
e1*e2*e4 - e1*e2*e5 + e1*e4*e5 - e2*e4*e5,
e0*e2*e3 - e0*e2*e5 + e0*e3*e5 - e2*e3*e5, e0*e1 - e0*e5 + e1*e5,
e2*e5 - e2*e6 + e5*e6, e1*e3*e5 - e1*e3*e6 + e1*e5*e6 - e3*e5*e6,
e0*e4*e5 - e0*e4*e6 + e0*e5*e6 - e4*e5*e6, e1*e4 - e1*e6 + e4*e6,
e2*e3*e4 - e2*e3*e6 + e2*e4*e6 - e3*e4*e6, e0*e3 - e0*e6 + e3*e6,
e0*e1*e2 - e0*e1*e6 + e0*e2*e6 - e1*e2*e6] over Rational Field

```

degree_on_basis(m)

Return the degree of the basis element indexed by *m*.

EXAMPLES:

```

sage: M = matroids.Wheel(3)
sage: OS = M.orlik_solomon_algebra(QQ)
sage: OS.degree_on_basis(frozenset([1]))
1
sage: OS.degree_on_basis(frozenset([0, 2, 3]))
3

```

```

>>> from sage.all import *
>>> M = matroids.Wheel(Integer(3))
>>> OS = M.orlik_solomon_algebra(QQ)
>>> OS.degree_on_basis(frozenset([Integer(1)]))
1
>>> OS.degree_on_basis(frozenset([Integer(0), Integer(2), Integer(3)]))
3

```

one_basis()

Return the index of the basis element corresponding to 1 in *self*.

EXAMPLES:

```

sage: M = matroids.Wheel(3)
sage: OS = M.orlik_solomon_algebra(QQ)
sage: OS.one_basis() == frozenset()
True

```

```

>>> from sage.all import *
>>> M = matroids.Wheel(Integer(3))
>>> OS = M.orlik_solomon_algebra(QQ)
>>> OS.one_basis() == frozenset()
True

```

product_on_basis(a, b)

Return the product in *self* of the basis elements indexed by *a* and *b*.

EXAMPLES:

```

sage: M = matroids.Wheel(3)
sage: OS = M.orlik_solomon_algebra(QQ)
sage: OS.product_on_basis(frozenset([2]), frozenset([3, 4]))
OS{0, 1, 2} - OS{0, 1, 4} + OS{0, 2, 3} + OS{0, 3, 4}

```

```
>>> from sage.all import *
>>> M = matroids.Wheel(Integer(3))
>>> OS = M.orlik_solomon_algebra(QQ)
>>> OS.product_on_basis(frozenset([Integer(2)]), frozenset([Integer(3),
↪ Integer(4)]))
OS{0, 1, 2} - OS{0, 1, 4} + OS{0, 2, 3} + OS{0, 3, 4}
```

```
sage: G = OS.algebra_generators()
sage: prod(G)
0
sage: G[2] * G[4]
-OS{1, 2} + OS{1, 4}
sage: G[3] * G[4] * G[2]
OS{0, 1, 2} - OS{0, 1, 4} + OS{0, 2, 3} + OS{0, 3, 4}
sage: G[2] * G[3] * G[4]
OS{0, 1, 2} - OS{0, 1, 4} + OS{0, 2, 3} + OS{0, 3, 4}
sage: G[3] * G[2] * G[4]
-OS{0, 1, 2} + OS{0, 1, 4} - OS{0, 2, 3} - OS{0, 3, 4}
```

```
>>> from sage.all import *
>>> G = OS.algebra_generators()
>>> prod(G)
0
>>> G[Integer(2)] * G[Integer(4)]
-OS{1, 2} + OS{1, 4}
>>> G[Integer(3)] * G[Integer(4)] * G[Integer(2)]
OS{0, 1, 2} - OS{0, 1, 4} + OS{0, 2, 3} + OS{0, 3, 4}
>>> G[Integer(2)] * G[Integer(3)] * G[Integer(4)]
OS{0, 1, 2} - OS{0, 1, 4} + OS{0, 2, 3} + OS{0, 3, 4}
>>> G[Integer(3)] * G[Integer(2)] * G[Integer(4)]
-OS{0, 1, 2} + OS{0, 1, 4} - OS{0, 2, 3} - OS{0, 3, 4}
```

subset_image(*S*)

Return the element e_S of $A(M)$ ($==$ self) corresponding to a subset S of the groundset of M .

INPUT:

- s – frozenset which is a subset of the groundset of M

EXAMPLES:

```
sage: M = matroids.Wheel(3)
sage: OS = M.orlik_solomon_algebra(QQ)
sage: BC = sorted(M.broken_circuits(), key=sorted)
sage: for bc in BC: (sorted(bc), OS.subset_image(bc))
([1, 3], -OS{0, 1} + OS{0, 3})
([1, 4, 5], OS{0, 1, 4} - OS{0, 1, 5} - OS{0, 3, 4} + OS{0, 3, 5})
([2, 3, 4], OS{0, 1, 2} - OS{0, 1, 4} + OS{0, 2, 3} + OS{0, 3, 4})
([2, 3, 5], OS{0, 2, 3} + OS{0, 3, 5})
([2, 4], -OS{1, 2} + OS{1, 4})
([2, 5], -OS{0, 2} + OS{0, 5})
([4, 5], -OS{3, 4} + OS{3, 5})

sage: # needs sage.graphs
```

(continues on next page)

(continued from previous page)

```

sage: M4 = matroids.CompleteGraphic(4)
sage: OSM4 = M4.orlik_solomon_algebra(QQ)
sage: OSM4.subset_image(frozenset({2,3,4}))
OS{0, 2, 3} + OS{0, 3, 4}

```

```

>>> from sage.all import *
>>> M = matroids.Wheel(Integer(3))
>>> OS = M.orlik_solomon_algebra(QQ)
>>> BC = sorted(M.broken_circuits(), key=sorted)
>>> for bc in BC: (sorted(bc), OS.subset_image(bc))
([1, 3], -OS{0, 1} + OS{0, 3})
([1, 4, 5], OS{0, 1, 4} - OS{0, 1, 5} - OS{0, 3, 4} + OS{0, 3, 5})
([2, 3, 4], OS{0, 1, 2} - OS{0, 1, 4} + OS{0, 2, 3} + OS{0, 3, 4})
([2, 3, 5], OS{0, 2, 3} + OS{0, 3, 5})
([2, 4], -OS{1, 2} + OS{1, 4})
([2, 5], -OS{0, 2} + OS{0, 5})
([4, 5], -OS{3, 4} + OS{3, 5})

>>> # needs sage.graphs
>>> M4 = matroids.CompleteGraphic(Integer(4))
>>> OSM4 = M4.orlik_solomon_algebra(QQ)
>>> OSM4.subset_image(frozenset({Integer(2), Integer(3), Integer(4)}))
OS{0, 2, 3} + OS{0, 3, 4}

```

An example of a custom ordering:

```

sage: # needs sage.graphs
sage: G = Graph([[3, 4], [4, 1], [1, 2], [2, 3], [3, 5], [5, 6], [6, 3]])
sage: MG = Matroid(G)
sage: s = [(5, 6), (1, 2), (3, 5), (2, 3), (1, 4), (3, 6), (3, 4)]
sage: sorted([sorted(c) for c in MG.circuits()])
[[ (1, 2), (1, 4), (2, 3), (3, 4) ],
 [ (3, 5), (3, 6), (5, 6) ]]
sage: OSMG = MG.orlik_solomon_algebra(QQ, ordering=s)
sage: OSMG.subset_image(frozenset())
OS{}
sage: OSMG.subset_image(frozenset([(1,2), (3,4), (1,4), (2,3)]))
0
sage: OSMG.subset_image(frozenset([(2,3), (1,2), (3,4)]))
OS{(1, 2), (2, 3), (3, 4)}
sage: OSMG.subset_image(frozenset([(1,4), (3,4), (2,3), (3,6), (5,6)]))
-OS{(1, 2), (1, 4), (2, 3), (3, 6), (5, 6)}
+ OS{(1, 2), (1, 4), (3, 4), (3, 6), (5, 6)}
- OS{(1, 2), (2, 3), (3, 4), (3, 6), (5, 6)}
sage: OSMG.subset_image(frozenset([(1,4), (3,4), (2,3), (3,6), (3,5)]))
OS{(1, 2), (1, 4), (2, 3), (3, 5), (5, 6)}
- OS{(1, 2), (1, 4), (2, 3), (3, 6), (5, 6)}
+ OS{(1, 2), (1, 4), (3, 4), (3, 5), (5, 6)}
+ OS{(1, 2), (1, 4), (3, 4), (3, 6), (5, 6)}
- OS{(1, 2), (2, 3), (3, 4), (3, 5), (5, 6)}
- OS{(1, 2), (2, 3), (3, 4), (3, 6), (5, 6)}

```

```

>>> from sage.all import *
>>> # needs sage.graphs
>>> G = Graph([[Integer(3), Integer(4)], [Integer(4), Integer(1)],
↳ [Integer(1), Integer(2)], [Integer(2), Integer(3)], [Integer(3),
↳ Integer(5)], [Integer(5), Integer(6)], [Integer(6), Integer(3)]])
>>> MG = Matroid(G)
>>> s = [(Integer(5), Integer(6)), (Integer(1), Integer(2)), (Integer(3),
↳ Integer(5)), (Integer(2), Integer(3)), (Integer(1), Integer(4)),
↳ (Integer(3), Integer(6)), (Integer(3), Integer(4))]
>>> sorted([sorted(c) for c in MG.circuits()])
[[ (1, 2), (1, 4), (2, 3), (3, 4)],
 [ (3, 5), (3, 6), (5, 6)]]
>>> OSMG = MG.orlik_solomon_algebra(QQ, ordering=s)
>>> OSMG.subset_image(frozenset())
OS{}
>>> OSMG.subset_image(frozenset([(Integer(1), Integer(2)), (Integer(3),
↳ Integer(4)), (Integer(1), Integer(4)), (Integer(2), Integer(3))]))
0
>>> OSMG.subset_image(frozenset([(Integer(2), Integer(3)), (Integer(1),
↳ Integer(2)), (Integer(3), Integer(4))]))
OS{(1, 2), (2, 3), (3, 4)}
>>> OSMG.subset_image(frozenset([(Integer(1), Integer(4)), (Integer(3),
↳ Integer(4)), (Integer(2), Integer(3)), (Integer(3), Integer(6)), (Integer(5),
↳ Integer(6))]))
-OS{(1, 2), (1, 4), (2, 3), (3, 6), (5, 6)}
+ OS{(1, 2), (1, 4), (3, 4), (3, 6), (5, 6)}
- OS{(1, 2), (2, 3), (3, 4), (3, 6), (5, 6)}
>>> OSMG.subset_image(frozenset([(Integer(1), Integer(4)), (Integer(3),
↳ Integer(4)), (Integer(2), Integer(3)), (Integer(3), Integer(6)), (Integer(3),
↳ Integer(5))]))
OS{(1, 2), (1, 4), (2, 3), (3, 5), (5, 6)}
- OS{(1, 2), (1, 4), (2, 3), (3, 6), (5, 6)}
+ OS{(1, 2), (1, 4), (3, 4), (3, 5), (5, 6)}
+ OS{(1, 2), (1, 4), (3, 4), (3, 6), (5, 6)}
- OS{(1, 2), (2, 3), (3, 4), (3, 5), (5, 6)}
- OS{(1, 2), (2, 3), (3, 4), (3, 6), (5, 6)}

```

```

class sage.algebras.orlik_solomon.OrlikSolomonInvariantAlgebra(R, M, G,
                                                                action_on_groundset=None,
                                                                *args, **kwargs)

```

Bases: `FiniteDimensionalInvariantModule`

The invariant algebra of the Orlik-Solomon algebra from the action on $A(M)$ induced from the `action_on_groundset`.

INPUT:

- R – the ring of coefficients
- M – a matroid
- G – a semigroup
- `action_on_groundset` – (optional) a function defining the action of G on the elements of the groundset of M ; default is $g(x)$

EXAMPLES:

Lets start with the action of S_3 on the rank 2 braid matroid:

```
sage: # needs sage.graphs
sage: M = matroids.CompleteGraphic(3)
sage: M.groundset()
frozenset({0, 1, 2})
sage: G = SymmetricGroup(3)
↪needs sage.groups
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> M = matroids.CompleteGraphic(Integer(3))
>>> M.groundset()
frozenset({0, 1, 2})
>>> G = SymmetricGroup(Integer(3))
↪ # needs sage.groups
```

Calling elements g of G on an element i of $\{1, 2, 3\}$ defines the action we want, but since the groundset is $\{0, 1, 2\}$ we first add 1 and then subtract 1:

```
sage: def on_groundset(g, x):
.....:     return g(x+1) - 1
```

```
>>> from sage.all import *
>>> def on_groundset(g, x):
...     return g(x+Integer(1)) - Integer(1)
```

Now that we have defined an action we can create the invariant, and get its basis:

```
sage: # needs sage.graphs sage.groups
sage: OSG = M.orlik_solomon_algebra(QQ, invariant=(G, on_groundset))
sage: OSG.basis()
Finite family {0: B[0], 1: B[1]}
sage: [OSG.lift(b) for b in OSG.basis()]
[OS{0}, OS{0} + OS{1} + OS{2}]
```

```
>>> from sage.all import *
>>> # needs sage.graphs sage.groups
>>> OSG = M.orlik_solomon_algebra(QQ, invariant=(G, on_groundset))
>>> OSG.basis()
Finite family {0: B[0], 1: B[1]}
>>> [OSG.lift(b) for b in OSG.basis()]
[OS{0}, OS{0} + OS{1} + OS{2}]
```

Since it is invariant, the action of any g in G is trivial:

```
sage: # needs sage.graphs sage.groups
sage: x = OSG.an_element(); x
2*B[0] + 2*B[1]
sage: g = G.an_element(); g
(2, 3)
sage: g * x
2*B[0] + 2*B[1]
```

(continues on next page)

(continued from previous page)

```
sage: # needs sage.graphs sage.groups
sage: x = OSG.random_element()
sage: g = G.random_element()
sage: g * x == x
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs sage.groups
>>> x = OSG.an_element(); x
2*B[0] + 2*B[1]
>>> g = G.an_element(); g
(2, 3)
>>> g * x
2*B[0] + 2*B[1]

>>> # needs sage.graphs sage.groups
>>> x = OSG.random_element()
>>> g = G.random_element()
>>> g * x == x
True
```

The underlying ambient module is the Orlik-Solomon algebra, which is accessible via `ambient()`:

```
sage: M.orlik_solomon_algebra(QQ) is OSG.ambient() #_
↪needs sage.graphs sage.groups
True
```

```
>>> from sage.all import *
>>> M.orlik_solomon_algebra(QQ) is OSG.ambient() #_
↪needs sage.graphs sage.groups
True
```

There is not much structure here, so let's look at a bigger example. Here we will look at the rank 3 braid matroid, and to make things easier, we'll start the indexing at 1 so that the S_6 action on the groundset is simply calling `g`:

```
sage: # needs sage.graphs sage.groups
sage: M = matroids.CompleteGraphic(4); M.groundset()
frozenset({0, 1, 2, 3, 4, 5})
sage: new_bases = [frozenset(i+1 for i in j) for j in M.bases()]
sage: M = Matroid(bases=new_bases); M.groundset()
frozenset({1, 2, 3, 4, 5, 6})
sage: G = SymmetricGroup(6)
sage: OSG = M.orlik_solomon_algebra(QQ, invariant=G)
sage: OSG.basis()
Finite family {0: B[0], 1: B[1]}
sage: [OSG.lift(b) for b in OSG.basis()]
[OS{ }, OS{1} + OS{2} + OS{3} + OS{4} + OS{5} + OS{6}]
sage: (OSG.basis()[1])^2
0
sage: 5 * OSG.basis()[1]
5*B[1]
```

```

>>> from sage.all import *
>>> # needs sage.graphs sage.groups
>>> M = matroids.CompleteGraphic(Integer(4)); M.groundset()
frozenset({0, 1, 2, 3, 4, 5})
>>> new_bases = [frozenset(i+Integer(1) for i in j) for j in M.bases()]
>>> M = Matroid(bases=new_bases); M.groundset()
frozenset({1, 2, 3, 4, 5, 6})
>>> G = SymmetricGroup(Integer(6))
>>> OSG = M.orlik_solomon_algebra(QQ, invariant=G)
>>> OSG.basis()
Finite family {0: B[0], 1: B[1]}
>>> [OSG.lift(b) for b in OSG.basis()]
[OS{ }, OS{1} + OS{2} + OS{3} + OS{4} + OS{5} + OS{6}]
>>> (OSG.basis()[Integer(1)])**Integer(2)
0
>>> Integer(5) * OSG.basis()[Integer(1)]
5*B[1]

```

Next, we look at the same matroid but with an $S_3 \times S_3$ action (here realized as a Young subgroup of S_6):

```

sage: # needs sage.graphs sage.groups
sage: H = G.young_subgroup([3, 3])
sage: OSH = M.orlik_solomon_algebra(QQ, invariant=H)
sage: OSH.basis()
Finite family {0: B[0], 1: B[1], 2: B[2]}
sage: [OSH.lift(b) for b in OSH.basis()]
[OS{ }, OS{4} + OS{5} + OS{6}, OS{1} + OS{2} + OS{3}]

```

```

>>> from sage.all import *
>>> # needs sage.graphs sage.groups
>>> H = G.young_subgroup([Integer(3), Integer(3)])
>>> OSH = M.orlik_solomon_algebra(QQ, invariant=H)
>>> OSH.basis()
Finite family {0: B[0], 1: B[1], 2: B[2]}
>>> [OSH.lift(b) for b in OSH.basis()]
[OS{ }, OS{4} + OS{5} + OS{6}, OS{1} + OS{2} + OS{3}]

```

We implement an S_4 action on the vertices:

```

sage: # needs sage.graphs sage.groups
sage: M = matroids.CompleteGraphic(4)
sage: G = SymmetricGroup(4)
sage: edge_map = {i: M.groundset_to_edges([i])[0][:2]
.....:             for i in M.groundset()}
sage: inv_map = {v: k for k, v in edge_map.items()}
sage: def vert_action(g, x):
.....:     a, b = edge_map[x]
.....:     return inv_map[tuple(sorted([g(a+1)-1, g(b+1)-1]))]
sage: OSG = M.orlik_solomon_algebra(QQ, invariant=(G, vert_action))
sage: B = OSG.basis()
sage: [OSG.lift(b) for b in B]
[OS{ }, OS{0} + OS{1} + OS{2} + OS{3} + OS{4} + OS{5}]

```

```

>>> from sage.all import *
>>> # needs sage.graphs sage.groups
>>> M = matroids.CompleteGraphic(Integer(4))
>>> G = SymmetricGroup(Integer(4))
>>> edge_map = {i: M.groundset_to_edges([i])[Integer(0)][Integer(2)]
...             for i in M.groundset()}
>>> inv_map = {v: k for k, v in edge_map.items()}
>>> def vert_action(g, x):
...     a, b = edge_map[x]
...     return inv_map[tuple(sorted([g(a+Integer(1))-Integer(1), g(b+Integer(1))-
↪ Integer(1)]))]
>>> OSG = M.orlik_solomon_algebra(QQ, invariant=(G, vert_action))
>>> B = OSG.basis()
>>> [OSG.lift(b) for b in B]
[OS{ }, OS{0} + OS{1} + OS{2} + OS{3} + OS{4} + OS{5}]

```

We use this to describe the Young subgroup $S_2 \times S_2$ action:

```

sage: # needs sage.graphs sage.groups
sage: H = G.young_subgroup([2,2])
sage: OSH = M.orlik_solomon_algebra(QQ, invariant=(H, vert_action))
sage: B = OSH.basis()
sage: [OSH.lift(b) for b in B]
[OS{ }, OS{5}, OS{1} + OS{2} + OS{3} + OS{4}, OS{0},
-1/2*OS{1, 2} + OS{1, 5} - 1/2*OS{3, 4} + OS{3, 5},
OS{0, 5}, OS{0, 1} + OS{0, 2} + OS{0, 3} + OS{0, 4},
-1/2*OS{0, 1, 2} + OS{0, 1, 5} - 1/2*OS{0, 3, 4} + OS{0, 3, 5}]

```

```

>>> from sage.all import *
>>> # needs sage.graphs sage.groups
>>> H = G.young_subgroup([Integer(2), Integer(2)])
>>> OSH = M.orlik_solomon_algebra(QQ, invariant=(H, vert_action))
>>> B = OSH.basis()
>>> [OSH.lift(b) for b in B]
[OS{ }, OS{5}, OS{1} + OS{2} + OS{3} + OS{4}, OS{0},
-1/2*OS{1, 2} + OS{1, 5} - 1/2*OS{3, 4} + OS{3, 5},
OS{0, 5}, OS{0, 1} + OS{0, 2} + OS{0, 3} + OS{0, 4},
-1/2*OS{0, 1, 2} + OS{0, 1, 5} - 1/2*OS{0, 3, 4} + OS{0, 3, 5}]

```

We demonstrate the algebra structure:

```

sage: matrix([[b*bp for b in B] for bp in B]) #_
↪needs sage.graphs sage.groups
[ B[0]    B[1]    B[2]    B[3]    B[4]    B[5]    B[6]    B[7]]
[ B[1]      0  2*B[4]    B[5]      0      0  2*B[7]      0]
[ B[2] -2*B[4]      0    B[6]      0 -2*B[7]      0      0]
[ B[3]  -B[5]  -B[6]      0    B[7]      0      0      0]
[ B[4]      0      0    B[7]      0      0      0      0]
[ B[5]      0 -2*B[7]      0      0      0      0      0]
[ B[6]  2*B[7]      0      0      0      0      0      0]
[ B[7]      0      0      0      0      0      0      0]

```



```

>>> from sage.all import *
>>> matrix([[b*bp for b in B] for bp in B])
↳needs sage.graphs sage.groups
[ B[0] B[1] B[2] B[3] B[4] B[5] B[6] B[7]]
[ B[1] 0 2*B[4] B[5] 0 0 2*B[7] 0]
[ B[2] -2*B[4] 0 B[6] 0 -2*B[7] 0 0]
[ B[3] -B[5] -B[6] 0 B[7] 0 0 0]
[ B[4] 0 0 B[7] 0 0 0 0]
[ B[5] 0 -2*B[7] 0 0 0 0 0]
[ B[6] 2*B[7] 0 0 0 0 0 0]
[ B[7] 0 0 0 0 0 0 0]

```

Note

The algebra structure only exists when the action on the groundset yields an equivariant matroid, in the sense that $g \cdot I \in \mathcal{I}$ for every $g \in G$ and for every $I \in \mathcal{I}$.

construction()

Return the functorial construction of `self`.

This implementation of the method only returns `None`.

5.16 Partition/Diagram Algebras

```

class sage.combinat.partition_algebra.PartitionAlgebraElement_ak
    Bases: PartitionAlgebraElement_generic
class sage.combinat.partition_algebra.PartitionAlgebraElement_bk
    Bases: PartitionAlgebraElement_generic
class sage.combinat.partition_algebra.PartitionAlgebraElement_generic
    Bases: IndexedFreeModuleElement
class sage.combinat.partition_algebra.PartitionAlgebraElement_pk
    Bases: PartitionAlgebraElement_generic
class sage.combinat.partition_algebra.PartitionAlgebraElement_prk
    Bases: PartitionAlgebraElement_generic
class sage.combinat.partition_algebra.PartitionAlgebraElement_rk
    Bases: PartitionAlgebraElement_generic
class sage.combinat.partition_algebra.PartitionAlgebraElement_sk
    Bases: PartitionAlgebraElement_generic
class sage.combinat.partition_algebra.PartitionAlgebraElement_tk
    Bases: PartitionAlgebraElement_generic
class sage.combinat.partition_algebra.PartitionAlgebra_ak(R, k, n, name=None)
    Bases: PartitionAlgebra_generic
EXAMPLES:

```

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_ak(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.partition_algebra import *
>>> p = PartitionAlgebra_ak(QQ, Integer(3), Integer(1))
>>> p == loads(dumps(p))
True
```

class sage.combinat.partition_algebra.PartitionAlgebra_bk(*R, k, n, name=None*)

Bases: *PartitionAlgebra_generic*

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_bk(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.partition_algebra import *
>>> p = PartitionAlgebra_bk(QQ, Integer(3), Integer(1))
>>> p == loads(dumps(p))
True
```

class sage.combinat.partition_algebra.PartitionAlgebra_generic(*R, cclass, n, k, name=None, prefix=None*)

Bases: *CombinatorialFreeModule*

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: s = PartitionAlgebra_sk(QQ, 3, 1)
sage: TestSuite(s).run()
sage: s == loads(dumps(s))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.partition_algebra import *
>>> s = PartitionAlgebra_sk(QQ, Integer(3), Integer(1))
>>> TestSuite(s).run()
>>> s == loads(dumps(s))
True
```

one_basis()

Return the basis index for the unit of the algebra.

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: s = PartitionAlgebra_sk(ZZ, 3, 1)
```

(continues on next page)

(continued from previous page)

```
sage: len(s.one().support()) # indirect doctest
1
```

```
>>> from sage.all import *
>>> from sage.combinat.partition_algebra import *
>>> s = PartitionAlgebra_sk(ZZ, Integer(3), Integer(1))
>>> len(s.one().support()) # indirect doctest
1
```

product_on_basis (*left, right*)

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: s = PartitionAlgebra_sk(QQ, 3, 1)
sage: t12 = s(Set([Set([1,-2]),Set([2,-1]),Set([3,-3])]))
sage: t12^2 == s(1) #indirect doctest
True
```

```
>>> from sage.all import *
>>> from sage.combinat.partition_algebra import *
>>> s = PartitionAlgebra_sk(QQ, Integer(3), Integer(1))
>>> t12 = s(Set([Set([Integer(1),-Integer(2)]),Set([Integer(2),-Integer(1)]),
↳Set([Integer(3),-Integer(3)])]))
>>> t12**Integer(2) == s(Integer(1)) #indirect doctest
True
```

class sage.combinat.partition_algebra.**PartitionAlgebra_pk** (*R, k, n, name=None*)Bases: *PartitionAlgebra_generic*

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_pk(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.partition_algebra import *
>>> p = PartitionAlgebra_pk(QQ, Integer(3), Integer(1))
>>> p == loads(dumps(p))
True
```

class sage.combinat.partition_algebra.**PartitionAlgebra_prk** (*R, k, n, name=None*)Bases: *PartitionAlgebra_generic*

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_prk(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.partition_algebra import *
>>> p = PartitionAlgebra_prk(QQ, Integer(3), Integer(1))
>>> p == loads(dumps(p))
True
```

class sage.combinat.partition_algebra.PartitionAlgebra_rk($R, k, n, name=None$)

Bases: *PartitionAlgebra_generic*

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_rk(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.partition_algebra import *
>>> p = PartitionAlgebra_rk(QQ, Integer(3), Integer(1))
>>> p == loads(dumps(p))
True
```

class sage.combinat.partition_algebra.PartitionAlgebra_sk($R, k, n, name=None$)

Bases: *PartitionAlgebra_generic*

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_sk(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.partition_algebra import *
>>> p = PartitionAlgebra_sk(QQ, Integer(3), Integer(1))
>>> p == loads(dumps(p))
True
```

class sage.combinat.partition_algebra.PartitionAlgebra_tk($R, k, n, name=None$)

Bases: *PartitionAlgebra_generic*

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_tk(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

```
>>> from sage.all import *
>>> from sage.combinat.partition_algebra import *
>>> p = PartitionAlgebra_tk(QQ, Integer(3), Integer(1))
>>> p == loads(dumps(p))
True
```

`sage.combinat.partition_algebra.SetPartitionsAk(k)`

Return the combinatorial class of set partitions of type A_k .

EXAMPLES:

```
sage: A3 = SetPartitionsAk(3); A3
Set partitions of {1, ..., 3, -1, ..., -3}

sage: A3.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: A3.last() #random
{{-1}, {-2}, {3}, {1}, {-3}, {2}}
sage: A3.random_element() #random
{{1, 3, -3, -1}, {2, -2}}

sage: A3.cardinality()
203

sage: A2p5 = SetPartitionsAk(2.5); A2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block
sage: A2p5.cardinality()
52

sage: A2p5.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: A2p5.last() #random
{{-1}, {-2}, {2}, {3, -3}, {1}}
sage: A2p5.random_element() #random
{{-1}, {-2}, {3, -3}, {1, 2}}
```

```
>>> from sage.all import *
>>> A3 = SetPartitionsAk(Integer(3)); A3
Set partitions of {1, ..., 3, -1, ..., -3}

>>> A3.first() #random
{{1, 2, 3, -1, -3, -2}}
>>> A3.last() #random
{{-1}, {-2}, {3}, {1}, {-3}, {2}}
>>> A3.random_element() #random
{{1, 3, -3, -1}, {2, -2}}

>>> A3.cardinality()
203

>>> A2p5 = SetPartitionsAk(RealNumber('2.5')); A2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block
>>> A2p5.cardinality()
52

>>> A2p5.first() #random
{{1, 2, 3, -1, -3, -2}}
>>> A2p5.last() #random
{{-1}, {-2}, {2}, {3, -3}, {1}}
```

(continues on next page)

(continued from previous page)

```
>>> A2p5.random_element() #random
{{-1}, {-2}, {3, -3}, {1, 2}}
```

```
class sage.combinat.partition_algebra.SetPartitionsAk_k(k)
```

```
    Bases: SetPartitions_set
```

```
    Element
```

```
        alias of SetPartitionsXkElement
```

```
class sage.combinat.partition_algebra.SetPartitionsAkhalf_k(k)
```

```
    Bases: SetPartitions_set
```

```
    Element
```

```
        alias of SetPartitionsXkElement
```

```
sage.combinat.partition_algebra.SetPartitionsBk(k)
```

Return the combinatorial class of set partitions of type B_k .

These are the set partitions where every block has size 2.

EXAMPLES:

```
sage: B3 = SetPartitionsBk(3); B3
Set partitions of {1, ..., 3, -1, ..., -3} with block size 2

sage: B3.first() #random
{{2, -2}, {1, -3}, {3, -1}}
sage: B3.last() #random
{{1, 2}, {3, -2}, {-3, -1}}
sage: B3.random_element() #random
{{2, -1}, {1, -3}, {3, -2}}

sage: B3.cardinality()
15

sage: B2p5 = SetPartitionsBk(2.5); B2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and
↪with block size 2

sage: B2p5.first() #random
{{2, -1}, {3, -3}, {1, -2}}
sage: B2p5.last() #random
{{1, 2}, {3, -3}, {-1, -2}}
sage: B2p5.random_element() #random
{{2, -2}, {3, -3}, {1, -1}}

sage: B2p5.cardinality()
3
```

```
>>> from sage.all import *
>>> B3 = SetPartitionsBk(Integer(3)); B3
Set partitions of {1, ..., 3, -1, ..., -3} with block size 2

>>> B3.first() #random
```

(continues on next page)

(continued from previous page)

```

{{2, -2}, {1, -3}, {3, -1}}
>>> B3.last() #random
{{1, 2}, {3, -2}, {-3, -1}}
>>> B3.random_element() #random
{{2, -1}, {1, -3}, {3, -2}}

>>> B3.cardinality()
15

>>> B2p5 = SetPartitionsBk(RealNumber('2.5')); B2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and
↳with block size 2

>>> B2p5.first() #random
{{2, -1}, {3, -3}, {1, -2}}
>>> B2p5.last() #random
{{1, 2}, {3, -3}, {-1, -2}}
>>> B2p5.random_element() #random
{{2, -2}, {3, -3}, {1, -1}}

>>> B2p5.cardinality()
3

```

class sage.combinat.partition_algebra.**SetPartitionsBk_k**(*k*)

Bases: *SetPartitionsAk_k*

cardinality()

Return the number of set partitions in B_k where k is an integer.

This is given by $(2k)!! = (2k-1)*(2k-3)*\dots*5*3*1$.

EXAMPLES:

```

sage: SetPartitionsBk(3).cardinality()
15
sage: SetPartitionsBk(2).cardinality()
3
sage: SetPartitionsBk(1).cardinality()
1
sage: SetPartitionsBk(4).cardinality()
105
sage: SetPartitionsBk(5).cardinality()
945

```

```

>>> from sage.all import *
>>> SetPartitionsBk(Integer(3)).cardinality()
15
>>> SetPartitionsBk(Integer(2)).cardinality()
3
>>> SetPartitionsBk(Integer(1)).cardinality()
1
>>> SetPartitionsBk(Integer(4)).cardinality()
105

```

(continues on next page)

(continued from previous page)

```
>>> SetPartitionsBk(Integer(5)).cardinality()
945
```

```
class sage.combinat.partition_algebra.SetPartitionsBkhalf_k(k)
```

```
    Bases: SetPartitionsAkhalf_k
```

```
    cardinality()
```

```
sage.combinat.partition_algebra.SetPartitionsIk(k)
```

Return the combinatorial class of set partitions of type I_k .

These are set partitions with a propagating number of less than k . Note that the identity set partition $\{\{1, -1\}, \dots, \{k, -k\}\}$ is not in I_k .

EXAMPLES:

```
sage: I3 = SetPartitionsIk(3); I3
Set partitions of {1, ..., 3, -1, ..., -3} with propagating number < 3
sage: I3.cardinality()
197

sage: I3.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: I3.last() #random
{{-1}, {-2}, {3}, {1}, {-3}, {2}}
sage: I3.random_element() #random
{{-1}, {-3, -2}, {2, 3}, {1}}

sage: I2p5 = SetPartitionsIk(2.5); I2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and
↳ propagating number < 3
sage: I2p5.cardinality()
50

sage: I2p5.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: I2p5.last() #random
{{-1}, {-2}, {2}, {3, -3}, {1}}
sage: I2p5.random_element() #random
{{-1}, {-2}, {1, 3, -3}, {2}}
```

```
>>> from sage.all import *
>>> I3 = SetPartitionsIk(Integer(3)); I3
Set partitions of {1, ..., 3, -1, ..., -3} with propagating number < 3
>>> I3.cardinality()
197

>>> I3.first() #random
{{1, 2, 3, -1, -3, -2}}
>>> I3.last() #random
{{-1}, {-2}, {3}, {1}, {-3}, {2}}
>>> I3.random_element() #random
{{-1}, {-3, -2}, {2, 3}, {1}}
```

(continues on next page)

(continued from previous page)

```

>>> I2p5 = SetPartitionsIk(RealNumber('2.5')); I2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and
↳ propagating number < 3
>>> I2p5.cardinality()
50

>>> I2p5.first() #random
{{1, 2, 3, -1, -3, -2}}
>>> I2p5.last() #random
{{-1}, {-2}, {2}, {3, -3}, {1}}
>>> I2p5.random_element() #random
{{-1}, {-2}, {1, 3, -3}, {2}}

```

```
class sage.combinat.partition_algebra.SetPartitionsIk_k(k)
```

Bases: *SetPartitionsAk_k*

cardinality()

```
class sage.combinat.partition_algebra.SetPartitionsIkhalf_k(k)
```

Bases: *SetPartitionsAkhalf_k*

cardinality()

```
sage.combinat.partition_algebra.SetPartitionsPRk(k)
```

Return the combinatorial class of set partitions of type PR_k .

EXAMPLES:

```

sage: SetPartitionsPRk(3)
Set partitions of {1, ..., 3, -1, ..., -3} with at most 1 positive
and negative entry in each block and that are planar

```

```

>>> from sage.all import *
>>> SetPartitionsPRk(Integer(3))
Set partitions of {1, ..., 3, -1, ..., -3} with at most 1 positive
and negative entry in each block and that are planar

```

```
class sage.combinat.partition_algebra.SetPartitionsPRk_k(k)
```

Bases: *SetPartitionsRk_k*

cardinality()

```
class sage.combinat.partition_algebra.SetPartitionsPRkhalf_k(k)
```

Bases: *SetPartitionsRkhalf_k*

cardinality()

```
sage.combinat.partition_algebra.SetPartitionsPk(k)
```

Return the combinatorial class of set partitions of type P_k .

These are the planar set partitions.

EXAMPLES:

```

sage: P3 = SetPartitionsPk(3); P3
Set partitions of {1, ..., 3, -1, ..., -3} that are planar
sage: P3.cardinality()
132

sage: P3.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: P3.last() #random
{{-1}, {-2}, {3}, {1}, {-3}, {2}}
sage: P3.random_element() #random
{{1, 2, -1}, {-3}, {3, -2}}

sage: P2p5 = SetPartitionsPk(2.5); P2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and
↳that are planar
sage: P2p5.cardinality()
42

sage: P2p5.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: P2p5.last() #random
{{-1}, {-2}, {2}, {3, -3}, {1}}
sage: P2p5.random_element() #random
{{1, 2, 3, -3}, {-1, -2}}

```

```

>>> from sage.all import *
>>> P3 = SetPartitionsPk(Integer(3)); P3
Set partitions of {1, ..., 3, -1, ..., -3} that are planar
>>> P3.cardinality()
132

>>> P3.first() #random
{{1, 2, 3, -1, -3, -2}}
>>> P3.last() #random
{{-1}, {-2}, {3}, {1}, {-3}, {2}}
>>> P3.random_element() #random
{{1, 2, -1}, {-3}, {3, -2}}

>>> P2p5 = SetPartitionsPk(RealNumber('2.5')); P2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and
↳that are planar
>>> P2p5.cardinality()
42

>>> P2p5.first() #random
{{1, 2, 3, -1, -3, -2}}
>>> P2p5.last() #random
{{-1}, {-2}, {2}, {3, -3}, {1}}
>>> P2p5.random_element() #random
{{1, 2, 3, -3}, {-1, -2}}

```

```
class sage.combinat.partition_algebra.SetPartitionsPk_k(k)
```

```
Bases: SetPartitionsAk_k
```

cardinality()

class sage.combinat.partition_algebra.**SetPartitionsPkhalf_k**(k)

Bases: *SetPartitionsAkhalf_k*

cardinality()

sage.combinat.partition_algebra.**SetPartitionsRk**(k)

Return the combinatorial class of set partitions of type R_k .

EXAMPLES:

```
sage: SetPartitionsRk(3)
Set partitions of {1, ..., 3, -1, ..., -3} with at most 1 positive
and negative entry in each block
```

```
>>> from sage.all import *
>>> SetPartitionsRk(Integer(3))
Set partitions of {1, ..., 3, -1, ..., -3} with at most 1 positive
and negative entry in each block
```

class sage.combinat.partition_algebra.**SetPartitionsRk_k**(k)

Bases: *SetPartitionsAk_k*

cardinality()

class sage.combinat.partition_algebra.**SetPartitionsRkhalf_k**(k)

Bases: *SetPartitionsAkhalf_k*

cardinality()

sage.combinat.partition_algebra.**SetPartitionsSk**(k)

Return the combinatorial class of set partitions of type S_k .

There is a bijection between these set partitions and the permutations of $1, \dots, k$.

EXAMPLES:

```
sage: S3 = SetPartitionsSk(3); S3
Set partitions of {1, ..., 3, -1, ..., -3} with propagating number 3
sage: S3.cardinality()
6

sage: S3.list() #random
[{{2, -2}, {3, -3}, {1, -1}},
 {{1, -1}, {2, -3}, {3, -2}},
 {{2, -1}, {3, -3}, {1, -2}},
 {{1, -2}, {2, -3}, {3, -1}},
 {{1, -3}, {2, -1}, {3, -2}},
 {{1, -3}, {2, -2}, {3, -1}}]
sage: S3.first() #random
{{2, -2}, {3, -3}, {1, -1}}
sage: S3.last() #random
{{1, -3}, {2, -2}, {3, -1}}
sage: S3.random_element() #random
{{1, -3}, {2, -1}, {3, -2}}
```

(continues on next page)

(continued from previous page)

```

sage: S3p5 = SetPartitionsSk(3.5); S3p5
Set partitions of {1, ..., 4, -1, ..., -4} with 4 and -4 in the same block and
↳propagating number 4
sage: S3p5.cardinality()
6

sage: S3p5.list() #random
[{{2, -2}, {3, -3}, {1, -1}, {4, -4}},
 {{2, -3}, {1, -1}, {4, -4}, {3, -2}},
 {{2, -1}, {3, -3}, {1, -2}, {4, -4}},
 {{2, -3}, {1, -2}, {4, -4}, {3, -1}},
 {{1, -3}, {2, -1}, {4, -4}, {3, -2}},
 {{1, -3}, {2, -2}, {4, -4}, {3, -1}}]
sage: S3p5.first() #random
{{2, -2}, {3, -3}, {1, -1}, {4, -4}}
sage: S3p5.last() #random
{{1, -3}, {2, -2}, {4, -4}, {3, -1}}
sage: S3p5.random_element() #random
{{1, -3}, {2, -2}, {4, -4}, {3, -1}}

```

```

>>> from sage.all import *
>>> S3 = SetPartitionsSk(Integer(3)); S3
Set partitions of {1, ..., 3, -1, ..., -3} with propagating number 3
>>> S3.cardinality()
6

>>> S3.list() #random
[{{2, -2}, {3, -3}, {1, -1}},
 {{1, -1}, {2, -3}, {3, -2}},
 {{2, -1}, {3, -3}, {1, -2}},
 {{1, -2}, {2, -3}, {3, -1}},
 {{1, -3}, {2, -1}, {3, -2}},
 {{1, -3}, {2, -2}, {3, -1}}]
>>> S3.first() #random
{{2, -2}, {3, -3}, {1, -1}}
>>> S3.last() #random
{{1, -3}, {2, -2}, {3, -1}}
>>> S3.random_element() #random
{{1, -3}, {2, -1}, {3, -2}}

>>> S3p5 = SetPartitionsSk(RealNumber('3.5')); S3p5
Set partitions of {1, ..., 4, -1, ..., -4} with 4 and -4 in the same block and
↳propagating number 4
>>> S3p5.cardinality()
6

>>> S3p5.list() #random
[{{2, -2}, {3, -3}, {1, -1}, {4, -4}},
 {{2, -3}, {1, -1}, {4, -4}, {3, -2}},
 {{2, -1}, {3, -3}, {1, -2}, {4, -4}},
 {{2, -3}, {1, -2}, {4, -4}, {3, -1}},

```

(continues on next page)

(continued from previous page)

```

    {{1, -3}, {2, -1}, {4, -4}, {3, -2}},
    {{1, -3}, {2, -2}, {4, -4}, {3, -1}}}
>>> S3p5.first() #random
{{2, -2}, {3, -3}, {1, -1}, {4, -4}}
>>> S3p5.last() #random
{{1, -3}, {2, -2}, {4, -4}, {3, -1}}
>>> S3p5.random_element() #random
{{1, -3}, {2, -2}, {4, -4}, {3, -1}}

```

class sage.combinat.partition_algebra.**SetPartitionsSk_k**(k)

Bases: *SetPartitionsAk_k*

cardinality()

Return $k!$.

class sage.combinat.partition_algebra.**SetPartitionsSkhalf_k**(k)

Bases: *SetPartitionsAkhalf_k*

cardinality()

sage.combinat.partition_algebra.**SetPartitionsTk**(k)

Return the combinatorial class of set partitions of type T_k .

These are planar set partitions where every block is of size 2.

EXAMPLES:

```

sage: T3 = SetPartitionsTk(3); T3
Set partitions of {1, ..., 3, -1, ..., -3} with block size 2 and that are planar
sage: T3.cardinality()
5

sage: T3.first() #random
{{1, -3}, {2, 3}, {-1, -2}}
sage: T3.last() #random
{{1, 2}, {3, -1}, {-3, -2}}
sage: T3.random_element() #random
{{1, -3}, {2, 3}, {-1, -2}}

sage: T2p5 = SetPartitionsTk(2.5); T2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and
↪with block size 2 and that are planar
sage: T2p5.cardinality()
2

sage: T2p5.first() #random
{{2, -2}, {3, -3}, {1, -1}}
sage: T2p5.last() #random
{{1, 2}, {3, -3}, {-1, -2}}

```

```

>>> from sage.all import *
>>> T3 = SetPartitionsTk(Integer(3)); T3
Set partitions of {1, ..., 3, -1, ..., -3} with block size 2 and that are planar
>>> T3.cardinality()

```

(continues on next page)

(continued from previous page)

```

5

>>> T3.first() #random
{{1, -3}, {2, 3}, {-1, -2}}
>>> T3.last() #random
{{1, 2}, {3, -1}, {-3, -2}}
>>> T3.random_element() #random
{{1, -3}, {2, 3}, {-1, -2}}

>>> T2p5 = SetPartitionsTk(RealNumber('2.5')); T2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and
↳with block size 2 and that are planar
>>> T2p5.cardinality()
2

>>> T2p5.first() #random
{{2, -2}, {3, -3}, {1, -1}}
>>> T2p5.last() #random
{{1, 2}, {3, -3}, {-1, -2}}

```

```
class sage.combinat.partition_algebra.SetPartitionsTk_k(k)
```

Bases: `SetPartitionsBk_k`

`cardinality()`

```
class sage.combinat.partition_algebra.SetPartitionsTkhalf_k(k)
```

Bases: `SetPartitionsBkhalf_k`

`cardinality()`

```
class sage.combinat.partition_algebra.SetPartitionsXkElement(parent, s, check=True)
```

Bases: `SetPartition`

An element for the classes of `SetPartitionXk` where X is some letter.

`check()`

Check to make sure this is a set partition.

EXAMPLES:

```

sage: A2p5 = SetPartitionsAk(2.5)
sage: x = A2p5.first(); x
{{-3, -2, -1, 1, 2, 3}}
sage: x.check()
sage: y = A2p5.next(x); y
{{-3, 3}, {-2, -1, 1, 2}}
sage: y.check()

```

```

>>> from sage.all import *
>>> A2p5 = SetPartitionsAk(RealNumber('2.5'))
>>> x = A2p5.first(); x
{{-3, -2, -1, 1, 2, 3}}
>>> x.check()
>>> y = A2p5.next(x); y

```

(continues on next page)

(continued from previous page)

```
{-3, 3}, {-2, -1, 1, 2}}
>>> y.check()
```

`sage.combinat.partition_algebra.identity(k)`

Return the identity set partition $1, -1, \dots, k, -k$.

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: pa.identity(2)
{{2, -2}, {1, -1}}
```

```
>>> from sage.all import *
>>> import sage.combinat.partition_algebra as pa
>>> pa.identity(Integer(2))
{{2, -2}, {1, -1}}
```

`sage.combinat.partition_algebra.is_planar(sp)`

Return True if the diagram corresponding to the set partition is planar; otherwise, it returns False.

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: pa.is_planar( pa.to_set_partition([[1,-2],[2,-1]]))
False
sage: pa.is_planar( pa.to_set_partition([[1,-1],[2,-2]]))
True
```

```
>>> from sage.all import *
>>> import sage.combinat.partition_algebra as pa
>>> pa.is_planar( pa.to_set_partition([[Integer(1),-Integer(2)],[Integer(2),-
↪Integer(1)]]))
False
>>> pa.is_planar( pa.to_set_partition([[Integer(1),-Integer(1)],[Integer(2),-
↪Integer(2)]]))
True
```

`sage.combinat.partition_algebra.pair_to_graph(sp1, sp2)`

Return a graph consisting of the disjoint union of the graphs of set partitions `sp1` and `sp2` along with edges joining the bottom row (negative numbers) of `sp1` to the top row (positive numbers) of `sp2`.

The vertices of the graph `sp1` appear in the result as pairs $(k, 1)$, whereas the vertices of the graph `sp2` appear as pairs $(k, 2)$.

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: sp1 = pa.to_set_partition([[1,-2],[2,-1]])
sage: sp2 = pa.to_set_partition([[1,-2],[2,-1]])
sage: g = pa.pair_to_graph( sp1, sp2 ); g
Graph on 8 vertices
```

```
>>> from sage.all import *
>>> import sage.combinat.partition_algebra as pa
>>> sp1 = pa.to_set_partition([[Integer(1), -Integer(2)], [Integer(2), -Integer(1)]])
>>> sp2 = pa.to_set_partition([[Integer(1), -Integer(2)], [Integer(2), -Integer(1)]])
>>> g = pa.pair_to_graph( sp1, sp2 ); g
Graph on 8 vertices
```

```
sage: g.vertices(sort=False) #random
[(1, 2), (-1, 1), (-2, 2), (-1, 2), (-2, 1), (2, 1), (2, 2), (1, 1)]
sage: g.edges(sort=False) #random
[((1, 2), (-1, 1), None),
 ((1, 2), (-2, 2), None),
 ((-1, 1), (2, 1), None),
 ((-1, 2), (2, 2), None),
 ((-2, 1), (1, 1), None),
 ((-2, 1), (2, 2), None)]
```

```
>>> from sage.all import *
>>> g.vertices(sort=False) #random
[(1, 2), (-1, 1), (-2, 2), (-1, 2), (-2, 1), (2, 1), (2, 2), (1, 1)]
>>> g.edges(sort=False) #random
[((1, 2), (-1, 1), None),
 ((1, 2), (-2, 2), None),
 ((-1, 1), (2, 1), None),
 ((-1, 2), (2, 2), None),
 ((-2, 1), (1, 1), None),
 ((-2, 1), (2, 2), None)]
```

Another example which used to be wrong until [Issue #15958](#):

```
sage: sp3 = pa.to_set_partition([[1, -1], [2], [-2]])
sage: sp4 = pa.to_set_partition([[1], [-1], [2], [-2]])
sage: g = pa.pair_to_graph( sp3, sp4 ); g
Graph on 8 vertices

sage: g.vertices(sort=True)
[(-2, 1), (-2, 2), (-1, 1), (-1, 2), (1, 1), (1, 2), (2, 1), (2, 2)]
sage: g.edges(sort=True)
[((-2, 1), (2, 2), None), ((-1, 1), (1, 1), None),
 ((-1, 1), (1, 2), None)]
```

```
>>> from sage.all import *
>>> sp3 = pa.to_set_partition([[Integer(1), -Integer(1)], [Integer(2)], [-
↪ Integer(2)]])
>>> sp4 = pa.to_set_partition([[Integer(1)], [-Integer(1)], [Integer(2)], [-
↪ Integer(2)]])
>>> g = pa.pair_to_graph( sp3, sp4 ); g
Graph on 8 vertices

>>> g.vertices(sort=True)
[(-2, 1), (-2, 2), (-1, 1), (-1, 2), (1, 1), (1, 2), (2, 1), (2, 2)]
>>> g.edges(sort=True)
```

(continues on next page)

(continued from previous page)

```
[((-2, 1), (2, 2), None), ((-1, 1), (1, 1), None),
 ((-1, 1), (1, 2), None)]
```

`sage.combinat.partition_algebra.propagating_number(sp)`

Return the propagating number of the set partition `sp`.

The propagating number is the number of blocks with both a positive and negative number.

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: sp1 = pa.to_set_partition([[1,-2],[2,-1]])
sage: sp2 = pa.to_set_partition([[1,2],[-2,-1]])
sage: pa.propagating_number(sp1)
2
sage: pa.propagating_number(sp2)
0
```

```
>>> from sage.all import *
>>> import sage.combinat.partition_algebra as pa
>>> sp1 = pa.to_set_partition([[Integer(1),-Integer(2)],[Integer(2),-Integer(1)]])
>>> sp2 = pa.to_set_partition([[Integer(1),Integer(2)],[Integer(2),-Integer(1)]])
>>> pa.propagating_number(sp1)
2
>>> pa.propagating_number(sp2)
0
```

`sage.combinat.partition_algebra.set_partition_composition(sp1, sp2)`

Return a tuple consisting of the composition of the set partitions `sp1` and `sp2` and the number of components removed from the middle rows of the graph.

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: sp1 = pa.to_set_partition([[1,-2],[2,-1]])
sage: sp2 = pa.to_set_partition([[1,-2],[2,-1]])
sage: pa.set_partition_composition(sp1, sp2) == (pa.identity(2), 0)
True
```

```
>>> from sage.all import *
>>> import sage.combinat.partition_algebra as pa
>>> sp1 = pa.to_set_partition([[Integer(1),-Integer(2)],[Integer(2),-Integer(1)]])
>>> sp2 = pa.to_set_partition([[Integer(1),Integer(2)],[Integer(2),-Integer(1)]])
>>> pa.set_partition_composition(sp1, sp2) == (pa.identity(Integer(2)),
↪Integer(0))
True
```

`sage.combinat.partition_algebra.to_graph(sp)`

Return a graph representing the set partition `sp`.

EXAMPLES:

```

sage: import sage.combinat.partition_algebra as pa
sage: g = pa.to_graph( pa.to_set_partition([[1,-2],[2,-1]])); g
Graph on 4 vertices

sage: g.vertices(sort=False) #random
[1, 2, -2, -1]
sage: g.edges(sort=False) #random
[(1, -2, None), (2, -1, None)]

```

```

>>> from sage.all import *
>>> import sage.combinat.partition_algebra as pa
>>> g = pa.to_graph( pa.to_set_partition([[Integer(1),-Integer(2)],[Integer(2),-
↳Integer(1)]])); g
Graph on 4 vertices

>>> g.vertices(sort=False) #random
[1, 2, -2, -1]
>>> g.edges(sort=False) #random
[(1, -2, None), (2, -1, None)]

```

`sage.combinat.partition_algebra.to_set_partition(l, k=None)`

Convert a list of a list of numbers to a set partitions.

Each list of numbers in the outer list specifies the numbers contained in one of the blocks in the set partition.

If k is specified, then the set partition will be a set partition of $1, \dots, k, -1, \dots, -k$. Otherwise, k will default to the minimum number needed to contain all of the specified numbers.

EXAMPLES:

```

sage: import sage.combinat.partition_algebra as pa
sage: pa.to_set_partition([[1,-1],[2,-2]]) == pa.identity(2)
True

```

```

>>> from sage.all import *
>>> import sage.combinat.partition_algebra as pa
>>> pa.to_set_partition([[Integer(1),-Integer(1)],[Integer(2),-Integer(2)]])) ==
↳pa.identity(Integer(2))
True

```

5.17 Quantum Clifford Algebras

AUTHORS:

- Travis Scrimshaw (2021-05): initial version

class `sage.algebras.quantum_clifford.QuantumCliffordAlgebra` ($n, k, q, F, \text{psi}, \text{indices}$)

Bases: `CombinatorialFreeModule`

The quantum Clifford algebra.

The *quantum Clifford algebra*, or q -Clifford algebra, of rank n and twist k is the unital associative algebra $\text{Cl}_q(n, k)$

over a field F with generators $\psi_a, \psi_a^*, \omega_a$ for $a = 1, \dots, n$ that satisfy the following relations:

$$\begin{aligned} \omega_a \omega_b &= \omega_b \omega_a, & \omega_a^{4k} &= (1 + q^{-2k}) \omega_a^{2k} - q^{-2k}, \\ \omega_a \psi_b &= q^{\delta_{ab}} \psi_b \omega_a, & \omega_a \psi_b^* &= \psi_b^* \omega_a, \\ \psi_a \psi_b + \psi_b \psi_a &= 0, & \psi_a^* \psi_b^* + \psi_b^* \psi_a^* &= 0, \\ \psi_a \psi_a^* + q^k \psi_a^* \psi_a &= \omega_a^{-k}, & \psi_a^* \psi_a + q^{-k} \psi_a^* \psi_a &= \omega_a^k, \\ \psi_a \psi_b^* + \psi_b^* \psi_a &= 0 & & \text{if } a \neq b. \end{aligned}$$

When $k = 2$, we recover the original definition given by Hayashi in [Hayashi1990]. The $k = 1$ version was used in [Kwon2014].

INPUT:

- n – positive integer; the rank
- k – positive integer (default: 1); the twist
- q – (optional) the parameter q
- F – (default: $\mathbf{Q}(q)$) the base field that contains q

EXAMPLES:

We construct the rank 3 and twist 1 q -Clifford algebra:

```
sage: Cl = algebras.QuantumClifford(3)
sage: Cl
Quantum Clifford algebra of rank 3 and twist 1 with q=q over
Fraction Field of Univariate Polynomial Ring in q over Integer Ring
sage: q = Cl.q()
```

```
>>> from sage.all import *
>>> Cl = algebras.QuantumClifford(Integer(3))
>>> Cl
Quantum Clifford algebra of rank 3 and twist 1 with q=q over
Fraction Field of Univariate Polynomial Ring in q over Integer Ring
>>> q = Cl.q()
```

Some sample computations:

```
sage: p0, p1, p2, d0, d1, d2, w0, w1, w2 = Cl.gens()
sage: p0 * p1
psi0*psi1
sage: p1 * p0
-psi0*psi1
sage: p0 * w0 * p1 * d0 * w2
(1/(q^3-q))*psi1*w2 + (-q/(q^2-1))*psi1*w0^2*w2
sage: w0^4
-1/q^2 + ((q^2+1)/q^2)*w0^2
```

```
>>> from sage.all import *
>>> p0, p1, p2, d0, d1, d2, w0, w1, w2 = Cl.gens()
>>> p0 * p1
psi0*psi1
>>> p1 * p0
-psi0*psi1
```

(continues on next page)

(continued from previous page)

```
>>> p0 * w0 * p1 * d0 * w2
(1/(q^3-q))*psi1*w2 + (-q/(q^2-1))*psi1*w0^2*w2
>>> w0**Integer(4)
-1/q^2 + ((q^2+1)/q^2)*w0^2
```

We construct the homomorphism from $U_q(\mathfrak{sl}_3)$ to $Cl(3, 1)$ given in (3.17) of [Hayashi1990]:

```
sage: e1 = p0*d1; e2 = p1*d2
sage: f1 = p1*d0; f2 = p2*d1
sage: k1 = w0~w1; k2 = w1~w2
sage: k1i = w1~w0; k2i = w2~w1
sage: (e1, e2, f1, f2, k1, k2, k1i, k2i)
(psi0*psid1, psi1*psid2,
 -psid0*psi1, -psid1*psi2,
 (q^2+1)*w0*w1 - q^2*w0*w1^3, (q^2+1)*w1*w2 - q^2*w1*w2^3,
 (q^2+1)*w0*w1 - q^2*w0^3*w1, (q^2+1)*w1*w2 - q^2*w1^3*w2)
```

```
>>> from sage.all import *
>>> e1 = p0*d1; e2 = p1*d2
>>> f1 = p1*d0; f2 = p2*d1
>>> k1 = w0~w1; k2 = w1~w2
>>> k1i = w1~w0; k2i = w2~w1
>>> (e1, e2, f1, f2, k1, k2, k1i, k2i)
(psi0*psid1, psi1*psid2,
 -psid0*psi1, -psid1*psi2,
 (q^2+1)*w0*w1 - q^2*w0*w1^3, (q^2+1)*w1*w2 - q^2*w1*w2^3,
 (q^2+1)*w0*w1 - q^2*w0^3*w1, (q^2+1)*w1*w2 - q^2*w1^3*w2)
```

We check that k_i and k_i^{-1} are inverses:

```
sage: k1 * k1i
1
sage: k2 * k2i
1
```

```
>>> from sage.all import *
>>> k1 * k1i
1
>>> k2 * k2i
1
```

The relations between e_i , f_i , and k_i :

```
sage: k1 * f1 == q^-2 * f1 * k1
True
sage: k2 * f1 == q^1 * f1 * k2
True
sage: k2 * e1 == q^-1 * e1 * k2
True
sage: k1 * e1 == q^2 * e1 * k1
True
sage: e1 * f1 - f1 * e1 == (k1 - k1i)/(q-q^-1)
```

(continues on next page)

(continued from previous page)

```
True
sage: e2 * f1 - f1 * e2
0
```

```
>>> from sage.all import *
>>> k1 * f1 == q**Integer(2) * f1 * k1
True
>>> k2 * f1 == q**Integer(1) * f1 * k2
True
>>> k2 * e1 == q**Integer(1) * e1 * k2
True
>>> k1 * e1 == q**Integer(2) * e1 * k1
True
>>> e1 * f1 - f1 * e1 == (k1 - k1i)/(q-q**Integer(1))
True
>>> e2 * f1 - f1 * e2
0
```

The q -Serre relations:

```
sage: e1 * e1 * e2 - (q^1 + q^-1) * e1 * e2 * e1 + e2 * e1 * e1
0
sage: f1 * f1 * f2 - (q^1 + q^-1) * f1 * f2 * f1 + f2 * f1 * f1
0
```

```
>>> from sage.all import *
>>> e1 * e1 * e2 - (q**Integer(1) + q**Integer(-1)) * e1 * e2 * e1 + e2 * e1 * e1
0
>>> f1 * f1 * f2 - (q**Integer(1) + q**Integer(-1)) * f1 * f2 * f1 + f2 * f1 * f1
0
```

This also can be constructed at the special point when $q^{2k} = 1$, but the basis used is different:

```
sage: Cl = algebras.QuantumClifford(1, 1, -1)
sage: Cl.inject_variables()
Defining psi0, psid0, w0
sage: psi0 * psid0
psi0*psid0
sage: psid0 * psi0
-w0 + psi0*psid0
sage: w0^2
1
```

```
>>> from sage.all import *
>>> Cl = algebras.QuantumClifford(Integer(1), Integer(1), -Integer(1))
>>> Cl.inject_variables()
Defining psi0, psid0, w0
>>> psi0 * psid0
psi0*psid0
>>> psid0 * psi0
-w0 + psi0*psid0
```

(continues on next page)

(continued from previous page)

```
>>> w0**Integer(2)
1
```

algebra_generators()

Return the algebra generators of self.

EXAMPLES:

```
sage: C1 = algebras.QuantumClifford(3)
sage: C1.algebra_generators()
Finite family {'psi0': psi0, 'psi1': psi1, 'psi2': psi2,
               'psid0': psid0, 'psid1': psid1, 'psid2': psid2,
               'w0': w0, 'w1': w1, 'w2': w2}
```

```
>>> from sage.all import *
>>> C1 = algebras.QuantumClifford(Integer(3))
>>> C1.algebra_generators()
Finite family {'psi0': psi0, 'psi1': psi1, 'psi2': psi2,
               'psid0': psid0, 'psid1': psid1, 'psid2': psid2,
               'w0': w0, 'w1': w1, 'w2': w2}
```

dimension()

Return the dimension of self.

EXAMPLES:

```
sage: C1 = algebras.QuantumClifford(3)
sage: C1.dimension()
512

sage: C1 = algebras.QuantumClifford(4, 2) # long time
sage: C1.dimension() # long time
65536
```

```
>>> from sage.all import *
>>> C1 = algebras.QuantumClifford(Integer(3))
>>> C1.dimension()
512

>>> C1 = algebras.QuantumClifford(Integer(4), Integer(2)) # long time
>>> C1.dimension() # long time
65536
```

gens()

Return the generators of self.

EXAMPLES:

```
sage: C1 = algebras.QuantumClifford(3)
sage: C1.gens()
(psi0, psi1, psi2, psid0, psid1, psid2, w0, w1, w2)
```

```
>>> from sage.all import *
>>> Cl = algebras.QuantumClifford(Integer(3))
>>> Cl.gens()
(psi0, psi1, psi2, psid0, psid1, psid2, w0, w1, w2)
```

one_basis()

Return the index of the basis element of 1.

EXAMPLES:

```
sage: Cl = algebras.QuantumClifford(3)
sage: Cl.one_basis()
((0, 0, 0), (0, 0, 0))
```

```
>>> from sage.all import *
>>> Cl = algebras.QuantumClifford(Integer(3))
>>> Cl.one_basis()
((0, 0, 0), (0, 0, 0))
```

q()

Return the q of self.

EXAMPLES:

```
sage: Cl = algebras.QuantumClifford(3)
sage: Cl.q()
q

sage: Cl = algebras.QuantumClifford(3, q=QQ(-5))
sage: Cl.q()
-5
```

```
>>> from sage.all import *
>>> Cl = algebras.QuantumClifford(Integer(3))
>>> Cl.q()
q

>>> Cl = algebras.QuantumClifford(Integer(3), q=QQ(-Integer(5)))
>>> Cl.q()
-5
```

rank()

Return the rank k of self.

EXAMPLES:

```
sage: Cl = algebras.QuantumClifford(3, 2)
sage: Cl.rank()
3
```

```
>>> from sage.all import *
>>> Cl = algebras.QuantumClifford(Integer(3), Integer(2))
>>> Cl.rank()
3
```

twist()

Return the twist k of `self`.

EXAMPLES:

```
sage: Cl = algebras.QuantumClifford(3, 2)
sage: Cl.twist()
2
```

```
>>> from sage.all import *
>>> Cl = algebras.QuantumClifford(Integer(3), Integer(2))
>>> Cl.twist()
2
```

class sage.algebras.quantum_clifford.**QuantumCliffordAlgebraGeneric**(n, k, q, F)

Bases: *QuantumCliffordAlgebra*

The quantum Clifford algebra when $q^{2k} \neq 1$.

The *quantum Clifford algebra*, or q -Clifford algebra, of rank n and twist k is the unital associative algebra $\text{Cl}_q(n, k)$ over a field F with generators $\psi_a, \psi_a^*, \omega_a$ for $a = 1, \dots, n$ that satisfy the following relations:

$$\begin{aligned} \omega_a \omega_b &= \omega_b \omega_a, & \omega_a^{4k} &= (1 + q^{-2k}) \omega_a^{2k} - q^{-2k}, \\ \omega_a \psi_b &= q^{\delta_{ab}} \psi_b \omega_a, & \omega_a \psi_b^* &= \psi_b^* \omega_a, \\ \psi_a \psi_b + \psi_b \psi_a &= 0, & \psi_a^* \psi_b^* + \psi_b^* \psi_a^* &= 0, \\ \psi_a \psi_a^* &= \frac{q^k \omega_a^{3k} - q^{-k} \omega_a^k}{q^k - q^{-k}}, & \psi_a^* \psi_a &= \frac{q^{2k} (\omega_a - \omega_a^{3k})}{q^k - q^{-k}}, \\ \psi_a \psi_b^* + \psi_b^* \psi_a &= 0 & \text{if } a \neq b, \end{aligned}$$

where $q \in F$ such that $q^{2k} \neq 1$.

When $k = 2$, we recover the original definition given by Hayashi in [Hayashi1990]. The $k = 1$ version was used in [Kwon2014].

class **Element**

Bases: *IndexedFreeModuleElement*

inverse()

Return the inverse if `self` is a basis element.

EXAMPLES:

```
sage: Cl = algebras.QuantumClifford(2)
sage: Cl.inject_variables()
Defining psi0, psi1, psid0, psid1, w0, w1
sage: w0^-1
(q^2+1)*w0 - q^2*w0^3
sage: w0^-1 * w0
1
sage: w0^-2
(q^2+1) - q^2*w0^2
sage: w0^-2 * w0^2
1
sage: w0^-2 * w0 == w0^-1
True
```

(continues on next page)

(continued from previous page)

```

sage: w = w0 * w1
sage: w^-1
(q^4+2*q^2+1)*w0*w1 + (-q^4-q^2)*w0*w1^3
+ (-q^4-q^2)*w0^3*w1 + q^4*w0^3*w1^3
sage: w^-1 * w
1
sage: w * w^-1
1

sage: (2*w0)^-1
((q^2+1)/2)*w0 - q^2/2*w0^3

sage: (w0 + w1)^-1
Traceback (most recent call last):
...
ValueError: cannot invert self (= w1 + w0)
sage: (psi0 * w0)^-1
Traceback (most recent call last):
...
ValueError: cannot invert self (= psi0*w0)

sage: Cl = algebras.QuantumClifford(1, 2)
sage: Cl.inject_variables()
Defining psi0, psid0, w0
sage: (psi0 + psid0).inverse()
psid0*w0^2 + q^2*psi0*w0^2

sage: Cl = algebras.QuantumClifford(2, 2)
sage: Cl.inject_variables()
Defining psi0, psi1, psid0, psid1, w0, w1
sage: w0^-1
(q^4+1)*w0^3 - q^4*w0^7
sage: w0 * w0^-1
1

```

```

>>> from sage.all import *
>>> Cl = algebras.QuantumClifford(Integer(2))
>>> Cl.inject_variables()
Defining psi0, psi1, psid0, psid1, w0, w1
>>> w0**-Integer(1)
(q^2+1)*w0 - q^2*w0^3
>>> w0**-Integer(1) * w0
1
>>> w0**-Integer(2)
(q^2+1) - q^2*w0^2
>>> w0**-Integer(2) * w0**Integer(2)
1
>>> w0**-Integer(2) * w0 == w0**-Integer(1)
True
>>> w = w0 * w1
>>> w**-Integer(1)
(q^4+2*q^2+1)*w0*w1 + (-q^4-q^2)*w0*w1^3

```

(continues on next page)

(continued from previous page)

```

+ (-q^4-q^2)*w0^3*w1 + q^4*w0^3*w1^3
>>> w**Integer(1) * w
1
>>> w * w**Integer(1)
1

>>> (Integer(2)*w0)**Integer(1)
((q^2+1)/2)*w0 - q^2/2*w0^3

>>> (w0 + w1)**Integer(1)
Traceback (most recent call last):
...
ValueError: cannot invert self (= w1 + w0)
>>> (psi0 * w0)**Integer(1)
Traceback (most recent call last):
...
ValueError: cannot invert self (= psi0*w0)

>>> C1 = algebras.QuantumClifford(Integer(1), Integer(2))
>>> C1.inject_variables()
Defining psi0, psid0, w0
>>> (psi0 + psid0).inverse()
psid0*w0^2 + q^2*psi0*w0^2

>>> C1 = algebras.QuantumClifford(Integer(2), Integer(2))
>>> C1.inject_variables()
Defining psi0, psi1, psid0, psid1, w0, w1
>>> w0**Integer(1)
(q^4+1)*w0^3 - q^4*w0^7
>>> w0 * w0**Integer(1)
1

```

product_on_basis(*m1*, *m2*)

Return the product of the basis elements indexed by *m1* and *m2*.

EXAMPLES:

```

sage: C1 = algebras.QuantumClifford(3)
sage: C1.inject_variables()
Defining psi0, psi1, psi2, psid0, psid1, psid2, w0, w1, w2
sage: psi0^2 # indirect doctest
0
sage: psid0^2
0
sage: w0 * psi0
q*psi0*w0
sage: w0 * psid0
1/q*psid0*w0
sage: w2 * w0
w0*w2
sage: w0^4
-1/q^2 + ((q^2+1)/q^2)*w0^2

```

```

>>> from sage.all import *
>>> Cl = algebras.QuantumClifford(Integer(3))
>>> Cl.inject_variables()
Defining psi0, psi1, psi2, psid0, psid1, psid2, w0, w1, w2
>>> psi0**Integer(2) # indirect doctest
0
>>> psid0**Integer(2)
0
>>> w0 * psi0
q*psi0*w0
>>> w0 * psid0
1/q*psid0*w0
>>> w2 * w0
w0*w2
>>> w0**Integer(4)
-1/q^2 + ((q^2+1)/q^2)*w0^2

```

class sage.algebras.quantum_clifford.**QuantumCliffordAlgebraRootUnity**(n, k, q, F)

Bases: [QuantumCliffordAlgebra](#)

The quantum Clifford algebra when $q^{2k} = 1$.

The *quantum Clifford algebra*, or q -Clifford algebra, of rank n and twist k is the unital associative algebra $Cl_q(n, k)$ over a field F with generators $\psi_a, \psi_a^*, \omega_a$ for $a = 1, \dots, n$ that satisfy the following relations:

$$\begin{aligned}
 \omega_a \omega_b &= \omega_b \omega_a, & \omega_a^{2k} &= 1, \\
 \omega_a \psi_b &= q^{\delta_{ab}} \psi_b \omega_a, & \omega_a \psi_b^* &= \psi_b^* \omega_a, \\
 \psi_a \psi_b + \psi_b \psi_a &= 0, & \psi_a^* \psi_b^* + \psi_b^* \psi_a^* &= 0, \\
 \psi_a \psi_a^* + q^k \psi_a^* \psi_a &= \omega_a^k & \psi_a \psi_b^* + \psi_b^* \psi_a &= 0 \quad (a \neq b),
 \end{aligned}$$

where $q \in F$ such that $q^{2k} = 1$. This has further relations of

$$\begin{aligned}
 \psi_a^* \psi_a \psi_a^* &= \psi_a^* \omega_a^k, \\
 \psi_a \psi_a^* \psi_a &= q^k \psi_a \omega_a^k, \\
 (\psi_a \psi_a^*)^2 &= \psi_a \psi_a^* \omega_a^k.
 \end{aligned}$$

class **Element**

Bases: [IndexedFreeModuleElement](#)

inverse()

Return the inverse if self is a basis element.

EXAMPLES:

```

sage: Cl = algebras.QuantumClifford(3, 3, -1)
sage: Cl.inject_variables()
Defining psi0, psi1, psi2, psid0, psid1, psid2, w0, w1, w2
sage: w0^-1
w0^5
sage: w0^-1 * w0
1
sage: w0^-2

```

(continues on next page)

(continued from previous page)

```

w0^4
sage: w0^-2 * w0^2
1
sage: w0^-2 * w0 == w0^-1
True
sage: w = w0 * w1^3
sage: w^-1
w0^5*w1^3
sage: w^-1 * w
1
sage: w * w^-1
1

sage: (2*w0)^-1
1/2*w0^5

sage: Cl = algebras.QuantumClifford(3, 1, -1)
sage: Cl.inject_variables()
Defining psi0, psi1, psi2, psid0, psid1, psid2, w0, w1, w2

sage: (w0 + w1)^-1
Traceback (most recent call last):
...
ValueError: cannot invert self (= w1 + w0)
sage: (psi0 * w0)^-1
Traceback (most recent call last):
...
ValueError: cannot invert self (= psi0*w0)

sage: z = CyclotomicField(6).gen()
sage: Cl = algebras.QuantumClifford(1, 3, z)
sage: Cl.inject_variables()
Defining psi0, psid0, w0
sage: (psi0 + psid0).inverse()
psid0*w0^3 - psi0*w0^3

sage: Cl = algebras.QuantumClifford(2, 2, -1)
sage: Cl.inject_variables()
Defining psi0, psi1, psid0, psid1, w0, w1
sage: w0^-1
w0^3
sage: w0 * w0^-1
1

```

```

>>> from sage.all import *
>>> Cl = algebras.QuantumClifford(Integer(3), Integer(3), -Integer(1))
>>> Cl.inject_variables()
Defining psi0, psi1, psi2, psid0, psid1, psid2, w0, w1, w2
>>> w0**Integer(1)
w0^5
>>> w0**Integer(1) * w0
1

```

(continues on next page)

(continued from previous page)

```

>>> w0**Integer(2)
w0^4
>>> w0**Integer(2) * w0**Integer(2)
1
>>> w0**Integer(2) * w0 == w0**Integer(1)
True
>>> w = w0 * w1**Integer(3)
>>> w**Integer(1)
w0^5*w1^3
>>> w**Integer(1) * w
1
>>> w * w**Integer(1)
1

>>> (Integer(2)*w0)**Integer(1)
1/2*w0^5

>>> Cl = algebras.QuantumClifford(Integer(3), Integer(1), -Integer(1))
>>> Cl.inject_variables()
Defining psi0, psi1, psi2, psid0, psid1, psid2, w0, w1, w2

>>> (w0 + w1)**Integer(1)
Traceback (most recent call last):
...
ValueError: cannot invert self (= w1 + w0)
>>> (psi0 * w0)**Integer(1)
Traceback (most recent call last):
...
ValueError: cannot invert self (= psi0*w0)

>>> z = CyclotomicField(Integer(6)).gen()
>>> Cl = algebras.QuantumClifford(Integer(1), Integer(3), z)
>>> Cl.inject_variables()
Defining psi0, psid0, w0
>>> (psi0 + psid0).inverse()
psid0*w0^3 - psi0*w0^3

>>> Cl = algebras.QuantumClifford(Integer(2), Integer(2), -Integer(1))
>>> Cl.inject_variables()
Defining psi0, psi1, psid0, psid1, w0, w1
>>> w0**Integer(1)
w0^3
>>> w0 * w0**Integer(1)
1

```

product_on_basis(*m1*, *m2*)

Return the product of the basis elements indexed by *m1* and *m2*.

EXAMPLES:

```

sage: z = CyclotomicField(3).gen()
sage: Cl = algebras.QuantumClifford(3, 3, z)

```

(continues on next page)

(continued from previous page)

```

sage: Cl.inject_variables()
Defining psi0, psi1, psi2, psid0, psid1, psid2, w0, w1, w2
sage: psi0^2 # indirect doctest
0
sage: psid0^2
0
sage: w0 * psi0
-(-zeta3)*psi0*w0
sage: w0 * psid0
-(zeta3+1)*psid0*w0
sage: psi0 * psid0
psi0*psid0
sage: psid0 * psi0
w0^3 - psi0*psid0
sage: w2 * w0
w0*w2
sage: w0^6
1
sage: psi0 * psi1
psi0*psi1
sage: psi1 * psi0
-psi0*psi1
sage: psi1 * (psi0 * psi2)
-psi0*psi1*psi2

sage: z = CyclotomicField(6).gen()
sage: Cl = algebras.QuantumClifford(3, 3, z)
sage: Cl.inject_variables()
Defining psi0, psi1, psi2, psid0, psid1, psid2, w0, w1, w2

sage: psid1 * (psi1 * psid1)
psid1*w1^3
sage: (psi1*psid1) * (psi1 * psid1)
psi1*psid1*w1^3
sage: (psi1 * psid1) * psi1
-psi1*w1^3

```

```

>>> from sage.all import *
>>> z = CyclotomicField(Integer(3)).gen()
>>> Cl = algebras.QuantumClifford(Integer(3), Integer(3), z)
>>> Cl.inject_variables()
Defining psi0, psi1, psi2, psid0, psid1, psid2, w0, w1, w2
>>> psi0**Integer(2) # indirect doctest
0
>>> psid0**Integer(2)
0
>>> w0 * psi0
-(-zeta3)*psi0*w0
>>> w0 * psid0
-(zeta3+1)*psid0*w0
>>> psi0 * psid0
psi0*psid0

```

(continues on next page)

(continued from previous page)

```

>>> psid0 * psi0
w0^3 - psi0*psid0
>>> w2 * w0
w0*w2
>>> w0**Integer(6)
1
>>> psi0 * psi1
psi0*psi1
>>> psi1 * psi0
-psi0*psi1
>>> psi1 * (psi0 * psi2)
-psi0*psi1*psi2

>>> z = CyclotomicField(Integer(6)).gen()
>>> C1 = algebras.QuantumClifford(Integer(3), Integer(3), z)
>>> C1.inject_variables()
Defining psi0, psi1, psi2, psid0, psid1, psid2, w0, w1, w2

>>> psid1 * (psi1 * psid1)
psid1*w1^3
>>> (psi1* psid1) * (psi1 * psid1)
psi1*psid1*w1^3
>>> (psi1 * psid1) * psi1
-psid1*w1^3

```

5.18 Quantum Groups Using GAP's QuaGroup Package

AUTHORS:

- Travis Scrimshaw (03-2017): initial version

See the [documentation for GAP's QuaGroup package](#), originally authored by Willem Adriaan de Graaf.

class sage.algebras.quantum_groups.quantum_group_gap.**CrystalGraphVertex** (*V*, *s*)
 Bases: *SageObject*

Helper class used as the vertices of a crystal graph.

class sage.algebras.quantum_groups.quantum_group_gap.**HighestWeightModule** (*Q*, *weight*)
 Bases: *QuantumGroupModule*

A highest weight module of a quantum group.

Element

alias of *QuaGroupRepresentationElement*

an_element ()

Return the highest weight vector of *self*.

EXAMPLES:

```

sage: Q = QuantumGroup(['A', 2])
sage: V = Q.highest_weight_module([1, 1])
sage: V.highest_weight_vector()
1*v0

```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> V = Q.highest_weight_module([Integer(1), Integer(1)])
>>> V.highest_weight_vector()
1*v0
```

highest_weight_vector()

Return the highest weight vector of `self`.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 2])
sage: V = Q.highest_weight_module([1, 1])
sage: V.highest_weight_vector()
1*v0
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> V = Q.highest_weight_module([Integer(1), Integer(1)])
>>> V.highest_weight_vector()
1*v0
```

tensor(*V, **options)

Return the tensor product of `self` with `V`.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 2])
sage: V = Q.highest_weight_module([1, 1])
sage: Vp = Q.highest_weight_module([1, 0])
sage: Vp.tensor(V)
Highest weight module of weight Lambda[1] of Quantum Group of type ['A', 2]
↳with q=q
# Highest weight module of weight Lambda[1] + Lambda[2] of Quantum Group of
↳type ['A', 2] with q=q
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> V = Q.highest_weight_module([Integer(1), Integer(1)])
>>> Vp = Q.highest_weight_module([Integer(1), Integer(0)])
>>> Vp.tensor(V)
Highest weight module of weight Lambda[1] of Quantum Group of type ['A', 2]
↳with q=q
# Highest weight module of weight Lambda[1] + Lambda[2] of Quantum Group of
↳type ['A', 2] with q=q
```

class `sage.algebras.quantum_groups.quantum_group_gap.HighestWeightSubmodule` (*ambient, gen, weight*)

Bases: `QuantumGroupModule`

Initialize `self`.

EXAMPLES:


```
sage: Q = QuantumGroup(['A', 2])
sage: V = Q.highest_weight_module([1, 0])
sage: T = tensor([V, V])
sage: S = T.highest_weight_decomposition()[0]
sage: TestSuite(S).run()
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> V = Q.highest_weight_module([Integer(1), Integer(0)])
>>> T = tensor([V, V])
>>> S = T.highest_weight_decomposition()[Integer(0)]
>>> TestSuite(S).run()
```

Element

alias of *QuaGroupRepresentationElement*

ambient()

Return the ambient module of self.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 2])
sage: V = Q.highest_weight_module([1, 0])
sage: T = tensor([V, V])
sage: S = T.highest_weight_decomposition()[0]
sage: S.ambient() is T
True
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> V = Q.highest_weight_module([Integer(1), Integer(0)])
>>> T = tensor([V, V])
>>> S = T.highest_weight_decomposition()[Integer(0)]
>>> S.ambient() is T
True
```

an_element()

Return the highest weight vector of self.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 2])
sage: V = Q.highest_weight_module([1, 0])
sage: T = tensor([V, V])
sage: S = T.highest_weight_decomposition()[1]
sage: u = S.highest_weight_vector(); u
(1)*e.1
sage: u.lift()
-q^-1*(1*v0<x>F[a1]*v0) + 1*(F[a1]*v0<x>1*v0)
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> V = Q.highest_weight_module([Integer(1), Integer(0)])
```

(continues on next page)

(continued from previous page)

```

>>> T = tensor([V,V])
>>> S = T.highest_weight_decomposition()[Integer(1)]
>>> u = S.highest_weight_vector(); u
(1)*e.1
>>> u.lift()
-q^-1*(1*v0<x>F[a1]*v0) + 1*(F[a1]*v0<x>1*v0)

```

crystal_graph(*use_ambient=True*)

Return the crystal graph of *self*.

INPUT:

- *use_ambient* – boolean (default: *True*); if *True*, the vertices are given in terms of the ambient module

EXAMPLES:

```

sage: Q = QuantumGroup(['A',2])
sage: V = Q.highest_weight_module([1,0])
sage: T = tensor([V,V])
sage: S = T.highest_weight_decomposition()[1]
sage: G = S.crystal_graph()
sage: sorted(G.vertices(sort=False), key=str)
[<-q^-1*(1*v0<x>F[a1+a2]*v0) + 1*(F[a1+a2]*v0<x>1*v0)>,
 <-q^-1*(1*v0<x>F[a1]*v0) + 1*(F[a1]*v0<x>1*v0)>,
 <-q^-1*(F[a1]*v0<x>F[a1+a2]*v0) + 1*(F[a1+a2]*v0<x>F[a1]*v0)>]
sage: sorted(S.crystal_graph(False).vertices(sort=False), key=str)
[<(1)*e.1>, <(1)*e.2>, <(1)*e.3>]

```

```

>>> from sage.all import *
>>> Q = QuantumGroup(['A',Integer(2)])
>>> V = Q.highest_weight_module([Integer(1),Integer(0)])
>>> T = tensor([V,V])
>>> S = T.highest_weight_decomposition()[Integer(1)]
>>> G = S.crystal_graph()
>>> sorted(G.vertices(sort=False), key=str)
[<-q^-1*(1*v0<x>F[a1+a2]*v0) + 1*(F[a1+a2]*v0<x>1*v0)>,
 <-q^-1*(1*v0<x>F[a1]*v0) + 1*(F[a1]*v0<x>1*v0)>,
 <-q^-1*(F[a1]*v0<x>F[a1+a2]*v0) + 1*(F[a1+a2]*v0<x>F[a1]*v0)>]
>>> sorted(S.crystal_graph(False).vertices(sort=False), key=str)
[<(1)*e.1>, <(1)*e.2>, <(1)*e.3>]

```

highest_weight_vector()

Return the highest weight vector of *self*.

EXAMPLES:

```

sage: Q = QuantumGroup(['A',2])
sage: V = Q.highest_weight_module([1,0])
sage: T = tensor([V,V])
sage: S = T.highest_weight_decomposition()[1]
sage: u = S.highest_weight_vector(); u
(1)*e.1
sage: u.lift()
-q^-1*(1*v0<x>F[a1]*v0) + 1*(F[a1]*v0<x>1*v0)

```

```

>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> V = Q.highest_weight_module([Integer(1), Integer(0)])
>>> T = tensor([V, V])
>>> S = T.highest_weight_decomposition()[Integer(1)]
>>> u = S.highest_weight_vector(); u
(1)*e.1
>>> u.lift()
-q^-1*(1*v0<x>F[a1]*v0) + 1*(F[a1]*v0<x>1*v0)

```

lift()

The lift morphism from `self` to the ambient space.

EXAMPLES:

```

sage: Q = QuantumGroup(['A', 2])
sage: V = Q.highest_weight_module([1, 0])
sage: T = tensor([V, V])
sage: S = T.highest_weight_decomposition()[0]
sage: S.lift
Generic morphism:
  From: Highest weight submodule with weight 2*Lambda[1] generated by 1*(1*v0
↪<x>1*v0)
  To:   Highest weight module ... # Highest weight module ...
sage: x = sum(S.basis())
sage: x.lift()
1*(1*v0<x>1*v0) + 1*(1*v0<x>F[a1]*v0) + 1*(1*v0<x>F[a1+a2]*v0)
+ q^-1*(F[a1]*v0<x>1*v0) + 1*(F[a1]*v0<x>F[a1]*v0)
+ 1*(F[a1]*v0<x>F[a1+a2]*v0) + q^-1*(F[a1+a2]*v0<x>1*v0)
+ q^-1*(F[a1+a2]*v0<x>F[a1]*v0) + 1*(F[a1+a2]*v0<x>F[a1+a2]*v0)

```

```

>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> V = Q.highest_weight_module([Integer(1), Integer(0)])
>>> T = tensor([V, V])
>>> S = T.highest_weight_decomposition()[Integer(0)]
>>> S.lift
Generic morphism:
  From: Highest weight submodule with weight 2*Lambda[1] generated by 1*(1*v0
↪<x>1*v0)
  To:   Highest weight module ... # Highest weight module ...
>>> x = sum(S.basis())
>>> x.lift()
1*(1*v0<x>1*v0) + 1*(1*v0<x>F[a1]*v0) + 1*(1*v0<x>F[a1+a2]*v0)
+ q^-1*(F[a1]*v0<x>1*v0) + 1*(F[a1]*v0<x>F[a1]*v0)
+ 1*(F[a1]*v0<x>F[a1+a2]*v0) + q^-1*(F[a1+a2]*v0<x>1*v0)
+ q^-1*(F[a1+a2]*v0<x>F[a1]*v0) + 1*(F[a1+a2]*v0<x>F[a1+a2]*v0)

```

retract(elt)

The retract map from the ambient space to `self`.

EXAMPLES:

```

sage: Q = QuantumGroup(['A', 2])
sage: V = Q.highest_weight_module([1, 0])
sage: T = tensor([V, V])
sage: all(S.retract(S.lift(x)) == x
.....:     for S in T.highest_weight_decomposition()
.....:     for x in S.basis())
True

```

```

>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> V = Q.highest_weight_module([Integer(1), Integer(0)])
>>> T = tensor([V, V])
>>> all(S.retract(S.lift(x)) == x
...     for S in T.highest_weight_decomposition()
...     for x in S.basis())
True

```

class sage.algebras.quantum_groups.quantum_group_gap.**LowerHalfQuantumGroup**(*Q*)

Bases: *Parent, UniqueRepresentation*

The lower half of the quantum group.

class **Element** (*parent, libgap_elt*)

Bases: *QuaGroupModuleElement*

An element of the lower half of the quantum group.

bar()

Return the bar involution on *self*.

EXAMPLES:

```

sage: Q = QuantumGroup(['A', 2])
sage: F1, F2 = Q.F_simple()
sage: B = Q.lower_half()
sage: x = B(Q.an_element()); x
1 + (q)*F[a1]
sage: x.bar()
1 + (q^-1)*F[a1]
sage: (F1*x).bar() == F1 * x.bar()
True
sage: (F2*x).bar() == F2 * x.bar()
True

sage: Q = QuantumGroup(['G', 2])
sage: F1, F2 = Q.F_simple()
sage: q = Q.q()
sage: B = Q.lower_half()
sage: x = B(q^-2*F1*F2^2*F1)
sage: x
(q + q^-5)*F[a1]*F[a1+a2]*F[a2]
+ (q^8 + q^6 + q^2 + 1)*F[a1]^2*F[a2]^2
sage: x.bar()
(q^5 + q^-1)*F[a1]*F[a1+a2]*F[a2]
+ (q^12 + q^10 + q^6 + q^4)*F[a1]^2*F[a2]^2

```

```

>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> F1, F2 = Q.F_simple()
>>> B = Q.lower_half()
>>> x = B(Q.an_element()); x
1 + (q)*F[a1]
>>> x.bar()
1 + (q^-1)*F[a1]
>>> (F1*x).bar() == F1 * x.bar()
True
>>> (F2*x).bar() == F2 * x.bar()
True

>>> Q = QuantumGroup(['G', Integer(2)])
>>> F1, F2 = Q.F_simple()
>>> q = Q.q()
>>> B = Q.lower_half()
>>> x = B(q**Integer(2)*F1*F2**Integer(2)*F1)
>>> x
(q + q^-5)*F[a1]*F[a1+a2]*F[a2]
+ (q^8 + q^6 + q^2 + 1)*F[a1]^2)*F[a2]^2)
>>> x.bar()
(q^5 + q^-1)*F[a1]*F[a1+a2]*F[a2]
+ (q^12 + q^10 + q^6 + q^4)*F[a1]^2)*F[a2]^2)

```

braid_group_action(*braid*)

Return the action of the braid group element *braid* projected into *self*.

INPUT:

- *braid* – a reduced word of a braid group element

EXAMPLES:

```

sage: Q = QuantumGroup(['A', 2])
sage: L = Q.lower_half()
sage: v = L.highest_weight_vector().f_tilde([1, 2, 2, 1]); v
F[a1]*F[a1+a2]*F[a2]
sage: v.braid_group_action([1])
(-q^3-q)*F[a2]^2)
sage: v.braid_group_action([]) == v
True

```

```

>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> L = Q.lower_half()
>>> v = L.highest_weight_vector().f_tilde([Integer(1), Integer(2),
↪ Integer(2), Integer(1)]); v
F[a1]*F[a1+a2]*F[a2]
>>> v.braid_group_action([Integer(1)])
(-q^3-q)*F[a2]^2)
>>> v.braid_group_action([]) == v
True

```

monomial_coefficients(*copy=True*)

Return the dictionary of *self* whose keys are the basis indices and the values are coefficients.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 2])
sage: B = Q.lower_half()
sage: x = B.retract(Q.an_element()); x
1 + (q)*F[a1]
sage: sorted(x.monomial_coefficients().items(), key=str)
[((0, 0, 0), 1), ((1, 0, 0), q)]
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> B = Q.lower_half()
>>> x = B.retract(Q.an_element()); x
1 + (q)*F[a1]
>>> sorted(x.monomial_coefficients().items(), key=str)
[((0, 0, 0), 1), ((1, 0, 0), q)]
```

tau()Return the action of the τ anti-automorphism on self.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 2])
sage: F1, F2 = Q.F_simple()
sage: B = Q.lower_half()
sage: x = B(Q.an_element()); x
1 + (q)*F[a1]
sage: x.tau()
1 + (q)*F[a1]
sage: (F1*x).tau() == x.tau() * F1.tau()
True
sage: (F2*x).tau() == x.tau() * F2.tau()
True

sage: Q = QuantumGroup(['G', 2])
sage: F1, F2 = Q.F_simple()
sage: q = Q.q()
sage: B = Q.lower_half()
sage: x = B(q^-2*F1*F2^2*F1)
sage: x
(q + q^-5)*F[a1]*F[a1+a2]*F[a2]
+ (q^8 + q^6 + q^2 + 1)*F[a1]^2*F[a2]^2
sage: x.tau()
(q + q^-5)*F[a1]*F[a1+a2]*F[a2]
+ (q^8 + q^6 + q^2 + 1)*F[a1]^2*F[a2]^2
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> F1, F2 = Q.F_simple()
>>> B = Q.lower_half()
>>> x = B(Q.an_element()); x
1 + (q)*F[a1]
>>> x.tau()
```

(continues on next page)

(continued from previous page)

```

1 + (q)*F[a1]
>>> (F1*x).tau() == x.tau() * F1.tau()
True
>>> (F2*x).tau() == x.tau() * F2.tau()
True

>>> Q = QuantumGroup(['G', Integer(2)])
>>> F1, F2 = Q.F_simple()
>>> q = Q.q()
>>> B = Q.lower_half()
>>> x = B(q**Integer(2)*F1*F2**Integer(2)*F1)
>>> x
(q + q^-5)*F[a1]*F[a1+a2]*F[a2]
+ (q^8 + q^6 + q^2 + 1)*F[a1]^2)*F[a2]^2)
>>> x.tau()
(q + q^-5)*F[a1]*F[a1+a2]*F[a2]
+ (q^8 + q^6 + q^2 + 1)*F[a1]^2)*F[a2]^2)

```

algebra_generators()Return the algebra generators of `self`.

EXAMPLES:

```

sage: Q = QuantumGroup(['A', 2])
sage: B = Q.lower_half()
sage: B.algebra_generators()
Finite family {1: F[a1], 2: F[a2]}

```

```

>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> B = Q.lower_half()
>>> B.algebra_generators()
Finite family {1: F[a1], 2: F[a2]}

```

ambient()Return the ambient quantum group of `self`.

EXAMPLES:

```

sage: Q = QuantumGroup(['A', 2])
sage: B = Q.lower_half()
sage: B.ambient() is Q
True

```

```

>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> B = Q.lower_half()
>>> B.ambient() is Q
True

```

an_element()Return the highest weight vector of `self`.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 2])
sage: B = Q.lower_half()
sage: B.highest_weight_vector()
1
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> B = Q.lower_half()
>>> B.highest_weight_vector()
1
```

basis()

Return the basis of `self`.

This returns the PBW basis of `self`, which is given by monomials in $\{F_\alpha\}$, where α runs over all positive roots.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 2])
sage: B = Q.lower_half()
sage: basis = B.basis(); basis
Lazy family (monomial(i))_{i in The Cartesian product of
  (Non negative integers, Non negative integers, Non negative integers)}
sage: basis[1, 2, 1]
F[a1]*F[a1+a2]^(2)*F[a2]
sage: basis[1, 2, 4]
F[a1]*F[a1+a2]^(2)*F[a2]^(4)
sage: basis[1, 0, 4]
F[a1]*F[a2]^(4)
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> B = Q.lower_half()
>>> basis = B.basis(); basis
Lazy family (monomial(i))_{i in The Cartesian product of
  (Non negative integers, Non negative integers, Non negative integers)}
>>> basis[Integer(1), Integer(2), Integer(1)]
F[a1]*F[a1+a2]^(2)*F[a2]
>>> basis[Integer(1), Integer(2), Integer(4)]
F[a1]*F[a1+a2]^(2)*F[a2]^(4)
>>> basis[Integer(1), Integer(0), Integer(4)]
F[a1]*F[a2]^(4)
```

canonical_basis_elements()

Construct the monomial elements of `self` indexed by `k`.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 2])
sage: B = Q.lower_half()
sage: C = B.canonical_basis_elements(); C
Lazy family (Canonical basis(i))_{i in The Cartesian product of
  (Non negative integers, Non negative integers)}
```

(continues on next page)

(continued from previous page)

```
sage: C[2,1]
[F[a1]^(2)*F[a2], F[a1]*F[a1+a2] + (q^2)*F[a1]^(2)*F[a2]]
sage: C[1,2]
[F[a1]*F[a2]^(2), (q^2)*F[a1]*F[a2]^(2) + F[a1+a2]*F[a2]]
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> B = Q.lower_half()
>>> C = B.canonical_basis_elements(); C
Lazy family (Canonical basis(i))_{i in The Cartesian product of
  (Non negative integers, Non negative integers)}
>>> C[Integer(2), Integer(1)]
[F[a1]^(2)*F[a2], F[a1]*F[a1+a2] + (q^2)*F[a1]^(2)*F[a2]]
>>> C[Integer(1), Integer(2)]
[F[a1]*F[a2]^(2), (q^2)*F[a1]*F[a2]^(2) + F[a1+a2]*F[a2]]
```

gens()

Return the algebra generators of self.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 2])
sage: B = Q.lower_half()
sage: B.algebra_generators()
Finite family {1: F[a1], 2: F[a2]}
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> B = Q.lower_half()
>>> B.algebra_generators()
Finite family {1: F[a1], 2: F[a2]}
```

highest_weight_vector()

Return the highest weight vector of self.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 2])
sage: B = Q.lower_half()
sage: B.highest_weight_vector()
1
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> B = Q.lower_half()
>>> B.highest_weight_vector()
1
```

lift(elt)

Lift elt to the ambient quantum group of self.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 2])
sage: B = Q.lower_half()
sage: x = B.lift(B.an_element()); x
1
sage: x.parent() is Q
True
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> B = Q.lower_half()
>>> x = B.lift(B.an_element()); x
1
>>> x.parent() is Q
True
```

one()

Return the highest weight vector of self.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 2])
sage: B = Q.lower_half()
sage: B.highest_weight_vector()
1
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> B = Q.lower_half()
>>> B.highest_weight_vector()
1
```

retract(elt)

Retract elt from the ambient quantum group to self.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 2])
sage: B = Q.lower_half()
sage: x = Q.an_element(); x
1 + (q)*F[a1] + E[a1] + (q^2-1-q^-2 + q^-4)*[ K1 ; 2 ]
+ K1 + (-q^-1 + q^-3)*K1[ K1 ; 1 ]
sage: B.retract(x)
1 + (q)*F[a1]
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> B = Q.lower_half()
>>> x = Q.an_element(); x
1 + (q)*F[a1] + E[a1] + (q^2-1-q^-2 + q^-4)*[ K1 ; 2 ]
+ K1 + (-q^-1 + q^-3)*K1[ K1 ; 1 ]
>>> B.retract(x)
1 + (q)*F[a1]
```

zero()Return the zero element of `self`.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 2])
sage: B = Q.lower_half()
sage: B.zero()
0
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> B = Q.lower_half()
>>> B.zero()
0
```

class `sage.algebras.quantum_groups.quantum_group_gap.QaGroupModuleElement` (*parent*,
libgap_elt)

Bases: `Element`Base class for elements created using `QaGroup`.**e_tilde(i)**Return the action of the Kashiwara operator $\tilde{e}_i$ on `self`.

INPUT:

- `i` – an element of the index set or a list to perform a string of operators

EXAMPLES:

```
sage: Q = QuantumGroup(['B', 2])
sage: x = Q.one().f_tilde([1, 2, 1, 1, 2, 2])
sage: x.e_tilde([2, 2, 1, 2])
F[a1]^ (2)
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['B', Integer(2)])
>>> x = Q.one().f_tilde([Integer(1), Integer(2), Integer(1), Integer(1),
↳ Integer(2), Integer(2)])
>>> x.e_tilde([Integer(2), Integer(2), Integer(1), Integer(2)])
F[a1]^ (2)
```

f_tilde(i)Return the action of the Kashiwara operator $\tilde{f}_i$ on `self`.

INPUT:

- `i` – an element of the index set or a list to perform a string of operators

EXAMPLES:

```
sage: Q = QuantumGroup(['B', 2])
sage: Q.one().f_tilde(1)
F[a1]
sage: Q.one().f_tilde(2)
```

(continues on next page)

(continued from previous page)

```
F[a2]
sage: Q.one().f_tilde([1,2,1,1,2])
F[a1]*F[a1+a2]^(2)
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['B', Integer(2)])
>>> Q.one().f_tilde(Integer(1))
F[a1]
>>> Q.one().f_tilde(Integer(2))
F[a2]
>>> Q.one().f_tilde([Integer(1), Integer(2), Integer(1), Integer(1), Integer(2)])
F[a1]*F[a1+a2]^(2)
```

gap()

Return the gap representation of self.

EXAMPLES:

```
sage: Q = QuantumGroup(['B', 3])
sage: x = Q.an_element()
sage: x.gap()
1+(q)*F1+E1+(q^4-1-q^-4+q^-8)*[ K1 ; 2 ]+K1+(-q^-2+q^-6)*K1[ K1 ; 1 ]
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['B', Integer(3)])
>>> x = Q.an_element()
>>> x.gap()
1+(q)*F1+E1+(q^4-1-q^-4+q^-8)*[ K1 ; 2 ]+K1+(-q^-2+q^-6)*K1[ K1 ; 1 ]
```

class sage.algebras.quantum_groups.quantum_group_gap.**QuaGroupRepresentationElement** (*parent*, *lib-gap_elt*)

Bases: *QuaGroupModuleElement*

Element of a quantum group representation.

monomial_coefficients (*copy=True*)

Return the dictionary of self whose keys are the basis indices and the values are coefficients.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 2])
sage: V = Q.highest_weight_module([1, 1])
sage: v = V.highest_weight_vector()
sage: F1, F2 = Q.F_simple()
sage: q = Q.q()
sage: x = v + F1*v + q*F2*F1*v; x
1*v0 + F[a1]*v0 + (q^2)*F[a1]*F[a2]*v0 + (q)*F[a1+a2]*v0
sage: sorted(x.monomial_coefficients().items(), key=str)
[(0, 1), (1, 1), (3, q^2), (4, q)]
```

```

>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> V = Q.highest_weight_module([Integer(1), Integer(1)])
>>> v = V.highest_weight_vector()
>>> F1, F2 = Q.F_simple()
>>> q = Q.q()
>>> x = v + F1*v + q*F2*F1*v; x
1*v0 + F[a1]*v0 + (q^2)*F[a1]*F[a2]*v0 + (q)*F[a1+a2]*v0
>>> sorted(x.monomial_coefficients().items(), key=str)
[(0, 1), (1, 1), (3, q^2), (4, q)]

```

class sage.algebras.quantum_groups.quantum_group_gap.**QuantumGroup**(cartan_type, q)

Bases: UniqueRepresentation, Parent

A Drinfel'd-Jimbo quantum group (implemented using the optional GAP package QuaGroup).

EXAMPLES:

We check the quantum Serre relations. We first we import the q -binomial using the q -int for quantum groups:

```
sage: from sage.algebras.quantum_groups.q_numbers import q_binomial
```

```

>>> from sage.all import *
>>> from sage.algebras.quantum_groups.q_numbers import q_binomial

```

We verify the Serre relations for type A_2 :

```

sage: Q = algebras.QuantumGroup(['A', 2])
sage: F1, F12, F2 = Q.F()
sage: q = Q.q()
sage: F1^2*F2 - q_binomial(2, 1, q) * F1*F2*F1 + F2*F1^2
0

```

```

>>> from sage.all import *
>>> Q = algebras.QuantumGroup(['A', Integer(2)])
>>> F1, F12, F2 = Q.F()
>>> q = Q.q()
>>> F1**Integer(2)*F2 - q_binomial(Integer(2), Integer(1), q) * F1*F2*F1 +
↪ F2*F1**Integer(2)
0

```

We verify the Serre relations for type B_2 :

```

sage: Q = algebras.QuantumGroup(['B', 2])
sage: F1, F12, F122, F2 = Q.F()
sage: F1^2*F2 - q_binomial(2, 1, q^2) * F1*F2*F1 + F2*F1^2
0
sage: (F2^3*F1 - q_binomial(3, 1, q) * F2^2*F1*F2
....: + q_binomial(3, 2, q) * F2*F1*F2^2 - F1*F2^3)
0

```

```

>>> from sage.all import *
>>> Q = algebras.QuantumGroup(['B', Integer(2)])
>>> F1, F12, F122, F2 = Q.F()

```

(continues on next page)

(continued from previous page)

```

>>> F1**Integer(2)*F2 - q_binomial(Integer(2), Integer(1), q**Integer(2)) *
↳ F1**F2**F1 + F2**F1**Integer(2)
0
>>> (F2**Integer(3)*F1 - q_binomial(Integer(3), Integer(1), q) *
↳ F2**Integer(2)*F1**F2
... + q_binomial(Integer(3), Integer(2), q) * F2**F1**F2**Integer(2) -
↳ F1**F2**Integer(3))
0

```

REFERENCES:

- [Wikipedia article Quantum_group](#)

E()

Return the family of generators $\{E_\alpha\}_{\alpha \in \Phi}$, where Φ is the root system of `self`.

EXAMPLES:

```

sage: Q = QuantumGroup(['B', 2])
sage: list(Q.E())
[E[a1], E[a1+a2], E[a1+2*a2], E[a2]]

```

```

>>> from sage.all import *
>>> Q = QuantumGroup(['B', Integer(2)])
>>> list(Q.E())
[E[a1], E[a1+a2], E[a1+2*a2], E[a2]]

```

E_simple()

Return the family of generators $\{E_i := E_{\alpha_i}\}_{i \in I}$.

EXAMPLES:

```

sage: Q = QuantumGroup(['B', 2])
sage: Q.E_simple()
Finite family {1: E[a1], 2: E[a2]}

```

```

>>> from sage.all import *
>>> Q = QuantumGroup(['B', Integer(2)])
>>> Q.E_simple()
Finite family {1: E[a1], 2: E[a2]}

```

class Element (*parent, libgap_elt*)

Bases: *QuaGroupModuleElement*

bar()

Return the bar involution on `self`.

The bar involution is defined by

$$\overline{E_i} = E_i, \quad \overline{F_i} = F_i, \quad \overline{K_i} = K_i^{-1}.$$

EXAMPLES:

```

sage: Q = QuantumGroup(['A', 2])
sage: [gen.bar() for gen in Q.gens()]
[F[a1],
 (q-q^-1)*F[a1]*F[a2] + F[a1+a2],
 F[a2],
 (-q + q^-1)*[ K1 ; 1 ] + K1, K1,
 (-q + q^-1)*[ K2 ; 1 ] + K2, K2,
 E[a1],
 (-q^2 + 1)*E[a1]*E[a2] + (q^2)*E[a1+a2],
 E[a2]]

```

```

>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> [gen.bar() for gen in Q.gens()]
[F[a1],
 (q-q^-1)*F[a1]*F[a2] + F[a1+a2],
 F[a2],
 (-q + q^-1)*[ K1 ; 1 ] + K1, K1,
 (-q + q^-1)*[ K2 ; 1 ] + K2, K2,
 E[a1],
 (-q^2 + 1)*E[a1]*E[a2] + (q^2)*E[a1+a2],
 E[a2]]

```

braid_group_action(*braid*)

Return the action of the braid group element *braid*.

The braid group operator $T_i: U_q(\mathfrak{g}) \rightarrow U_q(\mathfrak{g})$ is defined by

$$\begin{aligned}
 T_i(E_i) &= -F_i K_i, \\
 T_i(E_j) &= \sum_{k=0}^{-a_{ij}} (-1)^k q_i^{-k} E_i^{(-a_{ij}-k)} E_j E_i^{(k)} \text{ if } i \neq j, \\
 T_i(K_j) &= K_j K_i^{a_{ij}}, \\
 T_i(F_i) &= -K_i^{-1} E_i, \\
 T_i(F_j) &= \sum_{k=0}^{-a_{ij}} (-1)^k q_i^{-k} F_i^{(k)} F_j F_i^{(-a_{ij}-k)} \text{ if } i \neq j,
 \end{aligned}$$

where $a_{ij} = \langle \alpha_j, \alpha_i^\vee \rangle$ is the (i, j) -entry of the Cartan matrix associated to $\mathfrak{g}$.

INPUT:

- *braid* – a reduced word of a braid group element

EXAMPLES:

```

sage: Q = QuantumGroup(['A', 2])
sage: F1 = Q.F_simple()[1]
sage: F1.braid_group_action([1])
(q-q^-1)*[ K1 ; 1 ]*E[a1] + (-1)*K1*E[a1]
sage: F1.braid_group_action([1, 2])
F[a2]
sage: F1.braid_group_action([2, 1])
(-q^3 + 3*q-q^-1 + q^-3)*[ K1 ; 1 ]*[ K2 ; 1 ]*E[a1]*E[a2]
+ (q^3-2*q + q^-1)*[ K1 ; 1 ]*[ K2 ; 1 ]*E[a1+a2]

```

(continues on next page)

(continued from previous page)

```

+ (q^2-2 + q^-2)*[ K1 ; 1 ]*K2*E[a1]*E[a2]
+ (-q^2 + 1)*[ K1 ; 1 ]*K2*E[a1+a2]
+ (q^2-2 + q^-2)*K1*[ K2 ; 1 ]*E[a1]*E[a2]
+ (-q^2 + 1)*K1*[ K2 ; 1 ]*E[a1+a2]
+ (-q + q^-1)*K1*K2*E[a1]*E[a2] + (q)*K1*K2*E[a1+a2]
sage: F1.braid_group_action([1,2,1]) == F1.braid_group_action([2,1,2])
True
sage: F1.braid_group_action([]) == F1
True

```

```

>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> F1 = Q.F_simple()[Integer(1)]
>>> F1.braid_group_action([Integer(1)])
(q-q^-1)*[ K1 ; 1 ]*E[a1] + (-1)*K1*E[a1]
>>> F1.braid_group_action([Integer(1), Integer(2)])
F[a2]
>>> F1.braid_group_action([Integer(2), Integer(1)])
(-q^3 + 3*q-3*q^-1 + q^-3)*[ K1 ; 1 ]*[ K2 ; 1 ]*E[a1]*E[a2]
+ (q^3-2*q + q^-1)*[ K1 ; 1 ]*[ K2 ; 1 ]*E[a1+a2]
+ (q^2-2 + q^-2)*[ K1 ; 1 ]*K2*E[a1]*E[a2]
+ (-q^2 + 1)*[ K1 ; 1 ]*K2*E[a1+a2]
+ (q^2-2 + q^-2)*K1*[ K2 ; 1 ]*E[a1]*E[a2]
+ (-q^2 + 1)*K1*[ K2 ; 1 ]*E[a1+a2]
+ (-q + q^-1)*K1*K2*E[a1]*E[a2] + (q)*K1*K2*E[a1+a2]
>>> F1.braid_group_action([Integer(1), Integer(2), Integer(1)]) == F1.braid_
group_action([Integer(2), Integer(1), Integer(2)])
True
>>> F1.braid_group_action([]) == F1
True

```

omega()Return the action of the ω automorphism on self.The ω automorphism is defined by

$$\omega(E_i) = F_i, \quad \omega(F_i) = E_i, \quad \omega(K_i) = K_i^{-1}.$$

EXAMPLES:

```

sage: Q = QuantumGroup(['A', 2])
sage: [gen.omega() for gen in Q.gens()]
[E[a1],
 (-q)*E[a1+a2],
 E[a2],
 (-q + q^-1)*[ K1 ; 1 ] + K1,
 K1,
 (-q + q^-1)*[ K2 ; 1 ] + K2,
 K2,
 F[a1],
 (-q^-1)*F[a1+a2],
 F[a2]]

```



```

>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> [gen.omega() for gen in Q.gens()]
[E[a1],
 (-q)*E[a1+a2],
 E[a2],
 (-q + q^-1)*[ K1 ; 1 ] + K1,
 K1,
 (-q + q^-1)*[ K2 ; 1 ] + K2,
 K2,
 F[a1],
 (-q^-1)*F[a1+a2],
 F[a2]]

```

tau()

Return the action of the τ anti-automorphism on `self`.

The τ anti-automorphism is defined by

$$\tau(E_i) = E_i, \quad \tau(F_i) = F_i, \quad \tau(K_i) = K_i^{-1}.$$

EXAMPLES:

```

sage: Q = QuantumGroup(['A', 2])
sage: [gen.tau() for gen in Q.gens()]
[F[a1],
 (-q^2 + 1)*F[a1]*F[a2] + (-q)*F[a1+a2],
 F[a2],
 (-q + q^-1)*[ K1 ; 1 ] + K1,
 K1,
 (-q + q^-1)*[ K2 ; 1 ] + K2,
 K2,
 E[a1],
 (q-q^-1)*E[a1]*E[a2] + (-q)*E[a1+a2],
 E[a2]]

```

```

>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> [gen.tau() for gen in Q.gens()]
[F[a1],
 (-q^2 + 1)*F[a1]*F[a2] + (-q)*F[a1+a2],
 F[a2],
 (-q + q^-1)*[ K1 ; 1 ] + K1,
 K1,
 (-q + q^-1)*[ K2 ; 1 ] + K2,
 K2,
 E[a1],
 (q-q^-1)*E[a1]*E[a2] + (-q)*E[a1+a2],
 E[a2]]

```

F()

Return the family of generators $\{F_\alpha\}_{\alpha \in \Phi}$, where Φ is the root system of `self`.

EXAMPLES:

```
sage: Q = QuantumGroup(['G', 2])
sage: list(Q.F())
[F[a1], F[3*a1+a2], F[2*a1+a2], F[3*a1+2*a2], F[a1+a2], F[a2]]
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['G', Integer(2)])
>>> list(Q.F())
[F[a1], F[3*a1+a2], F[2*a1+a2], F[3*a1+2*a2], F[a1+a2], F[a2]]
```

F_simple()

Return the family of generators $\{F_i := F_{\alpha_i}\}_{i \in I}$.

EXAMPLES:

```
sage: Q = QuantumGroup(['G', 2])
sage: Q.F_simple()
Finite family {1: F[a1], 2: F[a2]}
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['G', Integer(2)])
>>> Q.F_simple()
Finite family {1: F[a1], 2: F[a2]}
```

K()

Return the family of generators $\{K_i\}_{i \in I}$.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 3])
sage: Q.K()
Finite family {1: K1, 2: K2, 3: K3}
sage: Q.K_inverse()
Finite family {1: (-q + q^-1)*[ K1 ; 1 ] + K1,
                2: (-q + q^-1)*[ K2 ; 1 ] + K2,
                3: (-q + q^-1)*[ K3 ; 1 ] + K3}
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(3)])
>>> Q.K()
Finite family {1: K1, 2: K2, 3: K3}
>>> Q.K_inverse()
Finite family {1: (-q + q^-1)*[ K1 ; 1 ] + K1,
                2: (-q + q^-1)*[ K2 ; 1 ] + K2,
                3: (-q + q^-1)*[ K3 ; 1 ] + K3}
```

K_inverse()

Return the family of generators $\{K_i^{-1}\}_{i \in I}$.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 3])
sage: Q.K_inverse()
Finite family {1: (-q + q^-1)*[ K1 ; 1 ] + K1,
```

(continues on next page)

(continued from previous page)

```

2: (-q + q^-1)*[ K2 ; 1 ] + K2,
3: (-q + q^-1)*[ K3 ; 1 ] + K3}

```

```

>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(3)])
>>> Q.K_inverse()
Finite family {1: (-q + q^-1)*[ K1 ; 1 ] + K1,
                2: (-q + q^-1)*[ K2 ; 1 ] + K2,
                3: (-q + q^-1)*[ K3 ; 1 ] + K3}

```

algebra_generators()

Return the algebra generators of self.

EXAMPLES:

```

sage: Q = QuantumGroup(['A', 2])
sage: list(Q.algebra_generators())
[F[a1], F[a2],
 K1, K2,
 (-q + q^-1)*[ K1 ; 1 ] + K1, (-q + q^-1)*[ K2 ; 1 ] + K2,
 E[a1], E[a2]]

```

```

>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> list(Q.algebra_generators())
[F[a1], F[a2],
 K1, K2,
 (-q + q^-1)*[ K1 ; 1 ] + K1, (-q + q^-1)*[ K2 ; 1 ] + K2,
 E[a1], E[a2]]

```

antipode(elt)

Return the antipode of elt.

The antipode $S: U_q(\mathfrak{g}) \rightarrow U_q(\mathfrak{g})$ is the anti-automorphism defined by

$$S(E_i) = -K_i^{-1}E_i, \quad S(F_i) = -F_iK_i, \quad S(K_i) = K_i^{-1}.$$

EXAMPLES:

```

sage: Q = QuantumGroup(['B', 2])
sage: [Q.antipode(f) for f in Q.F()]
[(-1)*F[a1]*K1,
 (-q^6 + q^2)*F[a1]*F[a2]*K1*K2 + (-q^4)*F[a1+a2]*K1*K2,
 (-q^8 + q^6 + q^4-q^2)*F[a1]*F[a2]^2*K1
 + (-q^9 + 2*q^7-2*q^3 + q)*F[a1]*F[a2]^2*K1*K2[ K2 ; 1 ]
 + (-q^5 + q^3)*F[a1+a2]*F[a2]*K1
 + (-q^6 + 2*q^4-q^2)*F[a1+a2]*F[a2]*K1*K2[ K2 ; 1 ]
 + (-q^4)*F[a1+2*a2]*K1 + (-q^5 + q^3)*F[a1+2*a2]*K1*K2[ K2 ; 1 ],
 (-1)*F[a2]*K2]

```

```

>>> from sage.all import *
>>> Q = QuantumGroup(['B', Integer(2)])

```

(continues on next page)

(continued from previous page)

```
>>> [Q.antipode(f) for f in Q.F()]
[(-1)*F[a1]*K1,
 (-q^6 + q^2)*F[a1]*F[a2]*K1*K2 + (-q^4)*F[a1+a2]*K1*K2,
 (-q^8 + q^6 + q^4-q^2)*F[a1]*F[a2]^(2)*K1
 + (-q^9 + 2*q^7-2*q^3 + q)*F[a1]*F[a2]^(2)*K1*K2[ K2 ; 1 ]
 + (-q^5 + q^3)*F[a1+a2]*F[a2]*K1
 + (-q^6 + 2*q^4-q^2)*F[a1+a2]*F[a2]*K1*K2[ K2 ; 1 ]
 + (-q^4)*F[a1+2*a2]*K1 + (-q^5 + q^3)*F[a1+2*a2]*K1*K2[ K2 ; 1 ],
 (-1)*F[a2]*K2]
```

cartan_type()

Return the Cartan type of `self`.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 2])
sage: Q.cartan_type()
['A', 2]
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> Q.cartan_type()
['A', 2]
```

coproduct (elt, n=1)

Return the coproduct of `elt` (iterated `n` times).

The comultiplication $\Delta: U_q(\mathfrak{g}) \rightarrow U_q(\mathfrak{g}) \otimes U_q(\mathfrak{g})$ is defined by

$$\begin{aligned}\Delta(E_i) &= E_i \otimes 1 + K_i \otimes E_i, \\ \Delta(F_i) &= F_i \otimes K_i^{-1} + 1 \otimes F_i, \\ \Delta(K_i) &= K_i \otimes K_i.\end{aligned}$$

EXAMPLES:

```
sage: Q = QuantumGroup(['B', 2])
sage: [Q.coproduct(e) for e in Q.E()]
[1*(E[a1]<x>1) + 1*(K1<x>E[a1]),
 1*(E[a1+a2]<x>1) + 1*(K1*K2<x>E[a1+a2]) + q^2-q^-2*(K2*E[a1]<x>E[a2]),
 q^4-q^2-1 + q^-2*(E[a1]<x>E[a2]^(2)) + 1*(E[a1+2*a2]<x>1)
 + 1*(K1<x>E[a1+2*a2]) + q-q^-1*(K1*K2[ K2 ; 1 ]<x>E[a1+2*a2])
 + q-q^-1*(K2*E[a1+a2]<x>E[a2]) + q^5-2*q^3
 + 2*q^-1-q^-3*(K2[ K2 ; 1 ]*E[a1]<x>E[a2]^(2)),
 1*(E[a2]<x>1) + 1*(K2<x>E[a2])]
sage: [Q.coproduct(f, 2) for f in Q.F_simple()]
[1*(1<x>1<x>F[a1]) + -q^2 + q^-2*(1<x>F[a1]<x>[ K1 ; 1 ])
 + 1*(1<x>F[a1]<x>K1) + q^4-2 + q^-4*(F[a1]<x>[ K1 ; 1 ]<x>[ K1 ; 1 ])
 + -q^2 + q^-2*(F[a1]<x>[ K1 ; 1 ]<x>K1) + -q^2
 + q^-2*(F[a1]<x>K1<x>[ K1 ; 1 ]) + 1*(F[a1]<x>K1<x>K1),
 1*(1<x>1<x>F[a2]) + -q + q^-1*(1<x>F[a2]<x>[ K2 ; 1 ])
 + 1*(1<x>F[a2]<x>K2) + q^2-2 + q^-2*(F[a2]<x>[ K2 ; 1 ]<x>[ K2 ; 1 ])
 + -q + q^-1*(F[a2]<x>[ K2 ; 1 ]<x>K2) + -q
 + q^-1*(F[a2]<x>K2<x>[ K2 ; 1 ]) + 1*(F[a2]<x>K2<x>K2)]
```

```

>>> from sage.all import *
>>> Q = QuantumGroup(['B', Integer(2)])
>>> [Q.coproduct(e) for e in Q.E()]
[1*(E[a1]<x>1) + 1*(K1<x>E[a1]),
 1*(E[a1+a2]<x>1) + 1*(K1*K2<x>E[a1+a2]) + q^2-q^-2*(K2*E[a1]<x>E[a2]),
 q^4-q^2-1 + q^-2*(E[a1]<x>E[a2]^2) + 1*(E[a1+2*a2]<x>1)
 + 1*(K1<x>E[a1+2*a2]) + q-q^-1*(K1*K2[ K2 ; 1 ]<x>E[a1+2*a2])
 + q-q^-1*(K2*E[a1+a2]<x>E[a2]) + q^5-2*q^3
 + 2*q^-1-q^-3*(K2[ K2 ; 1 ]*E[a1]<x>E[a2]^2),
 1*(E[a2]<x>1) + 1*(K2<x>E[a2])]
>>> [Q.coproduct(f, Integer(2)) for f in Q.F_simple()]
[1*(1<x>1<x>F[a1]) + -q^2 + q^-2*(1<x>F[a1]<x>[ K1 ; 1 ])
 + 1*(1<x>F[a1]<x>K1) + q^4-2 + q^-4*(F[a1]<x>[ K1 ; 1 ]<x>[ K1 ; 1 ])
 + -q^2 + q^-2*(F[a1]<x>[ K1 ; 1 ]<x>K1) + -q^2
 + q^-2*(F[a1]<x>K1<x>[ K1 ; 1 ]) + 1*(F[a1]<x>K1<x>K1),
 1*(1<x>1<x>F[a2]) + -q + q^-1*(1<x>F[a2]<x>[ K2 ; 1 ])
 + 1*(1<x>F[a2]<x>K2) + q^2-2 + q^-2*(F[a2]<x>[ K2 ; 1 ]<x>[ K2 ; 1 ])
 + -q + q^-1*(F[a2]<x>[ K2 ; 1 ]<x>K2) + -q
 + q^-1*(F[a2]<x>K2<x>[ K2 ; 1 ]) + 1*(F[a2]<x>K2<x>K2)]

```

counit (*elt*)

Return the counit of *elt*.

The counit $\varepsilon: U_q(\mathfrak{g}) \rightarrow \mathbf{Q}(q)$ is defined by

$$\varepsilon(E_i) = \varepsilon(F_i) = 0, \quad \varepsilon(K_i) = 1.$$

EXAMPLES:

```

sage: Q = QuantumGroup(['B', 2])
sage: x = Q.an_element()^2
sage: Q.counit(x)
4
sage: Q.counit(Q.one())
1
sage: Q.counit(Q.zero())
0

```

```

>>> from sage.all import *
>>> Q = QuantumGroup(['B', Integer(2)])
>>> x = Q.an_element()**Integer(2)
>>> Q.counit(x)
4
>>> Q.counit(Q.one())
1
>>> Q.counit(Q.zero())
0

```

gap ()

Return the gap representation of *self*.

EXAMPLES:

```
sage: Q = QuantumGroup(['A',2])
sage: Q.gap()
QuantumUEA( <root system of type A2>, Qpar = q )
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A',Integer(2)])
>>> Q.gap()
QuantumUEA( <root system of type A2>, Qpar = q )
```

gens()

Return the generators of self.

EXAMPLES:

```
sage: Q = QuantumGroup(['A',2])
sage: Q.gens()
(F[a1], F[a1+a2], F[a2],
 K1, (-q + q^-1)*[ K1 ; 1 ] + K1,
 K2, (-q + q^-1)*[ K2 ; 1 ] + K2,
 E[a1], E[a1+a2], E[a2])
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A',Integer(2)])
>>> Q.gens()
(F[a1], F[a1+a2], F[a2],
 K1, (-q + q^-1)*[ K1 ; 1 ] + K1,
 K2, (-q + q^-1)*[ K2 ; 1 ] + K2,
 E[a1], E[a1+a2], E[a2])
```

highest_weight_module(weight)

Return the highest weight module of weight weight of self.

EXAMPLES:

```
sage: Q = QuantumGroup(['A',2])
sage: Q.highest_weight_module([1,3])
Highest weight module of weight Lambda[1] + 3*Lambda[2] of
Quantum Group of type ['A', 2] with q=q
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A',Integer(2)])
>>> Q.highest_weight_module([Integer(1),Integer(3)])
Highest weight module of weight Lambda[1] + 3*Lambda[2] of
Quantum Group of type ['A', 2] with q=q
```

lower_half()

Return the lower half of the quantum group self.

EXAMPLES:

```
sage: Q = QuantumGroup(['A',2])
sage: Q.lower_half()
Lower Half of Quantum Group of type ['A', 2] with q=q
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> Q.lower_half()
Lower Half of Quantum Group of type ['A', 2] with q=q
```

one()

Return the multiplicative identity of `self`.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 2])
sage: Q.one()
1
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> Q.one()
1
```

q()

Return the parameter q .

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 3])
sage: Q.q()
q
sage: zeta3 = CyclotomicField(3).gen()
sage: Q = QuantumGroup(['B', 2], q=zeta3)
sage: Q.q()
zeta3
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(3)])
>>> Q.q()
q
>>> zeta3 = CyclotomicField(Integer(3)).gen()
>>> Q = QuantumGroup(['B', Integer(2)], q=zeta3)
>>> Q.q()
zeta3
```

some_elements()

Return some elements of `self`.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 1])
sage: Q.some_elements()
[1 + (q)*F[a1] + E[a1] + (q^2-1-q^-2 + q^-4)*[ K1 ; 2 ]
 + K1 + (-q^-1 + q^-3)*K1[ K1 ; 1 ],
 K1, F[a1], E[a1]]
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(1)])
>>> Q.some_elements()
[1 + (q)*F[a1] + E[a1] + (q^2-1-q^-2 + q^-4)*[ K1 ; 2 ]
 + K1 + (-q^-1 + q^-3)*K1[ K1 ; 1 ],
 K1, F[a1], E[a1]]
```

zero()

Return the multiplicative identity of `self`.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 2])
sage: Q.zero()
0
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> Q.zero()
0
```

```
class sage.algebras.quantum_groups.quantum_group_gap.QuantumGroupHomset(X, Y,
                                                                    category=None,
                                                                    check=True,
                                                                    base=None)
```

Bases: `HomsetWithBase`

The homset whose domain is a quantum group.

```
class sage.algebras.quantum_groups.quantum_group_gap.QuantumGroupModule(Q, category)
```

Bases: `Parent`, `UniqueRepresentation`

Abstract base class for quantum group representations.

R_matrix()

Return the R -matrix of `self`.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 1])
sage: V = Q.highest_weight_module([1])
sage: V.R_matrix()
[      1      0      0      0]
[      0      q -q^2 + 1      0]
[      0      0      q      0]
[      0      0      0      1]
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(1)])
>>> V = Q.highest_weight_module([Integer(1)])
>>> V.R_matrix()
[      1      0      0      0]
[      0      q -q^2 + 1      0]
[      0      0      q      0]
[      0      0      0      1]
```


basis()

Return a basis of `self`.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 2])
sage: V = Q.highest_weight_module([1, 1])
sage: V.basis()
Family (1*v0, F[a1]*v0, F[a2]*v0, F[a1]*F[a2]*v0, F[a1+a2]*v0,
        F[a1]*F[a1+a2]*v0, F[a1+a2]*F[a2]*v0, F[a1+a2]^(2)*v0)
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> V = Q.highest_weight_module([Integer(1), Integer(1)])
>>> V.basis()
Family (1*v0, F[a1]*v0, F[a2]*v0, F[a1]*F[a2]*v0, F[a1+a2]*v0,
        F[a1]*F[a1+a2]*v0, F[a1+a2]*F[a2]*v0, F[a1+a2]^(2)*v0)
```

crystal_basis()

Return the crystal basis of `self`.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 2])
sage: V = Q.highest_weight_module([1, 1])
sage: V.crystal_basis()
Family (1*v0, F[a1]*v0, F[a2]*v0, F[a1]*F[a2]*v0,
        (q)*F[a1]*F[a2]*v0 + F[a1+a2]*v0, F[a1+a2]*F[a2]*v0,
        (-q^-2)*F[a1]*F[a1+a2]*v0, (-q^-1)*F[a1+a2]^(2)*v0)
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> V = Q.highest_weight_module([Integer(1), Integer(1)])
>>> V.crystal_basis()
Family (1*v0, F[a1]*v0, F[a2]*v0, F[a1]*F[a2]*v0,
        (q)*F[a1]*F[a2]*v0 + F[a1+a2]*v0, F[a1+a2]*F[a2]*v0,
        (-q^-2)*F[a1]*F[a1+a2]*v0, (-q^-1)*F[a1+a2]^(2)*v0)
```

crystal_graph()

Return the crystal graph of `self`.

EXAMPLES:

```
sage: Q = QuantumGroup(['A', 2])
sage: V = Q.highest_weight_module([1, 1])
sage: G = V.crystal_graph(); G
Digraph on 8 vertices

sage: B = crystals.Tableaux(['A', 2], shape=[2, 1])
sage: G.is_isomorphic(B.digraph(), edge_labels=True)
True
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
```

(continues on next page)

(continued from previous page)

```

>>> V = Q.highest_weight_module([Integer(1), Integer(1)])
>>> G = V.crystal_graph(); G
Digraph on 8 vertices

>>> B = crystals.Tableaux(['A', Integer(2)], shape=[Integer(2), Integer(1)])
>>> G.is_isomorphic(B.digraph(), edge_labels=True)
True

```

gap()Return the gap representation of *self*.

EXAMPLES:

```

sage: Q = QuantumGroup(['A', 2])
sage: V = Q.highest_weight_module([1, 1])
sage: V.gap()
<8-dimensional left-module over QuantumUEA( <root system of type A2>,
Qpar = q )>

```

```

>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> V = Q.highest_weight_module([Integer(1), Integer(1)])
>>> V.gap()
<8-dimensional left-module over QuantumUEA( <root system of type A2>,
Qpar = q )>

```

zero()Return the zero element of *self*.

EXAMPLES:

```

sage: Q = QuantumGroup(['A', 2])
sage: V = Q.highest_weight_module([1, 1])
sage: V.zero()
0*v0

```

```

>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> V = Q.highest_weight_module([Integer(1), Integer(1)])
>>> V.zero()
0*v0

```

```

class sage.algebras.quantum_groups.quantum_group_gap.QuantumGroupMorphism(parent, im_gens,
                                                                              check=True)

```

Bases: *Morphism*

A morphism whose domain is a quantum group.

im_gens()Return the image of the generators under *self*.

EXAMPLES:

```
sage: Q = QuantumGroup(['A',1])
sage: F, K, Ki, E = Q.gens()
sage: phi = Q.hom([E, Ki, K, F])
sage: phi.im_gens()
(E[a1], (-q + q^-1)*[ K1 ; 1 ] + K1, K1, F[a1])
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A',Integer(1)])
>>> F, K, Ki, E = Q.gens()
>>> phi = Q.hom([E, Ki, K, F])
>>> phi.im_gens()
(E[a1], (-q + q^-1)*[ K1 ; 1 ] + K1, K1, F[a1])
```

class sage.algebras.quantum_groups.quantum_group_gap.**TensorProductOfHighestWeightModules** (*modules, **options)

Bases: *QuantumGroupModule*

Initialize self.

EXAMPLES:

```
sage: Q = QuantumGroup(['A',2])
sage: V = Q.highest_weight_module([1,1])
sage: T = tensor([V,V])
sage: TestSuite(T).run()
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A',Integer(2)])
>>> V = Q.highest_weight_module([Integer(1),Integer(1)])
>>> T = tensor([V,V])
>>> TestSuite(T).run()
```

Element

alias of *QuaGroupRepresentationElement*

highest_weight_decomposition()

Return the highest weight decomposition of self.

EXAMPLES:

```
sage: Q = QuantumGroup(['A',2])
sage: V = Q.highest_weight_module([1,0])
sage: T = tensor([V,V])
sage: T.highest_weight_decomposition()
[Highest weight submodule with weight 2*Lambda[1] generated by 1*(1*v0<x>
↪ 1*v0),
 Highest weight submodule with weight Lambda[2] generated by -q^-1*(1*v0<x>
↪ F[a1]*v0) + 1*(F[a1]*v0<x>1*v0)]
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A',Integer(2)])
```

(continues on next page)

(continued from previous page)

```

>>> V = Q.highest_weight_module([Integer(1), Integer(0)])
>>> T = tensor([V, V])
>>> T.highest_weight_decomposition()
[Highest weight submodule with weight 2*Lambda[1] generated by 1*(1*v0<x>
↪ 1*v0),
 Highest weight submodule with weight Lambda[2] generated by -q^-1*(1*v0<x>
↪ F[a1]*v0) + 1*(F[a1]*v0<x>1*v0)]

```

highest_weight_vectors()

Return the highest weight vectors of `self`.

EXAMPLES:

```

sage: Q = QuantumGroup(['A', 2])
sage: V = Q.highest_weight_module([1, 0])
sage: T = tensor([V, V])
sage: T.highest_weight_vectors()
[1*(1*v0<x>1*v0), -q^-1*(1*v0<x>F[a1]*v0) + 1*(F[a1]*v0<x>1*v0)]

```

```

>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> V = Q.highest_weight_module([Integer(1), Integer(0)])
>>> T = tensor([V, V])
>>> T.highest_weight_vectors()
[1*(1*v0<x>1*v0), -q^-1*(1*v0<x>F[a1]*v0) + 1*(F[a1]*v0<x>1*v0)]

```

some_elements()

Return the highest weight vectors of `self`.

EXAMPLES:

```

sage: Q = QuantumGroup(['A', 2])
sage: V = Q.highest_weight_module([1, 0])
sage: T = tensor([V, V])
sage: T.highest_weight_vectors()
[1*(1*v0<x>1*v0), -q^-1*(1*v0<x>F[a1]*v0) + 1*(F[a1]*v0<x>1*v0)]

```

```

>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> V = Q.highest_weight_module([Integer(1), Integer(0)])
>>> T = tensor([V, V])
>>> T.highest_weight_vectors()
[1*(1*v0<x>1*v0), -q^-1*(1*v0<x>F[a1]*v0) + 1*(F[a1]*v0<x>1*v0)]

```

tensor_factors()

Return the factors of `self`.

EXAMPLES:

```

sage: Q = QuantumGroup(['A', 2])
sage: V = Q.highest_weight_module([1, 0])
sage: T = tensor([V, V])
sage: T.tensor_factors()

```

(continues on next page)

(continued from previous page)

```
(Highest weight module of weight Lambda[1] of Quantum Group of type ['A', 2]
↪with q=q,
Highest weight module of weight Lambda[1] of Quantum Group of type ['A', 2]
↪with q=q)
```

```
>>> from sage.all import *
>>> Q = QuantumGroup(['A', Integer(2)])
>>> V = Q.highest_weight_module([Integer(1), Integer(0)])
>>> T = tensor([V, V])
>>> T.tensor_factors()
(Highest weight module of weight Lambda[1] of Quantum Group of type ['A', 2]
↪with q=q,
Highest weight module of weight Lambda[1] of Quantum Group of type ['A', 2]
↪with q=q)
```

`sage.algebras.quantum_groups.quantum_group_gap.projection_lower_half(Q)`

Return the projection onto the lower half of the quantum group.

EXAMPLES:

```
sage: from sage.algebras.quantum_groups.quantum_group_gap import projection_lower_
↪half
sage: Q = QuantumGroup(['G', 2])
sage: phi = projection_lower_half(Q); phi
Quantum group homomorphism endomorphism of Quantum Group of type ['G', 2] with q=q
Defn: F[a1] |--> F[a1]
      F[a2] |--> F[a2]
      K1 |--> 0
      K2 |--> 0
      (-q + q^-1)*[ K1 ; 1 ] + K1 |--> 0
      (-q^3 + q^-3)*[ K2 ; 1 ] + K2 |--> 0
      E[a1] |--> 0
      E[a2] |--> 0
sage: all(phi(f) == f for f in Q.F())
True
sage: all(phi(e) == Q.zero() for e in Q.E())
True
sage: all(phi(K) == Q.zero() for K in Q.K())
True
```

```
>>> from sage.all import *
>>> from sage.algebras.quantum_groups.quantum_group_gap import projection_lower_
↪half
>>> Q = QuantumGroup(['G', Integer(2)])
>>> phi = projection_lower_half(Q); phi
Quantum group homomorphism endomorphism of Quantum Group of type ['G', 2] with q=q
Defn: F[a1] |--> F[a1]
      F[a2] |--> F[a2]
      K1 |--> 0
      K2 |--> 0
      (-q + q^-1)*[ K1 ; 1 ] + K1 |--> 0
      (-q^3 + q^-3)*[ K2 ; 1 ] + K2 |--> 0
```

(continues on next page)

(continued from previous page)

```

E[a1] |--> 0
E[a2] |--> 0
>>> all(phi(f) == f for f in Q.F())
True
>>> all(phi(e) == Q.zero() for e in Q.E())
True
>>> all(phi(K) == Q.zero() for K in Q.K())
True

```

5.19 Quantum Matrix Coordinate Algebras

AUTHORS:

- Travis Scrimshaw (01-2016): initial version

class sage.algebras.quantum_matrix_coordinate_algebra.**QuantumGL**(*n*, *q*, *bar*, *R*)

Bases: *QuantumMatrixCoordinateAlgebra_abstract*

Quantum coordinate algebra of $GL(n)$.

The quantum coordinate algebra of $GL(n)$, or quantum $GL(n)$ for short and denoted by $\mathcal{O}_q(GL(n))$, is the quantum coordinate algebra of $M_R(n, n)$ with the addition of the additional central group-like element c which satisfies $cd = dc = 1$, where d is the quantum determinant.

Quantum $GL(n)$ is a Hopf algebra where $\varepsilon(c) = 1$ and the antipode S is given by the (quantum) matrix inverse. That is to say, we have $S(c) = c^{-1} = d$ and

$$S(x_{ij}) = c * (-q)^{i-j} * \tilde{t}_{ji},$$

where we have the quantum minor

$$\tilde{t}_{ij} = \sum_{\sigma} (-q)^{\ell(\sigma)} x_{1,\sigma(1)} \cdots x_{i-1,\sigma(i-1)} x_{i+1,\sigma(i+1)} \cdots x_{n,\sigma(n)}$$

with the sum over permutations $\sigma: \{1, \dots, i-1, i+1, \dots, n\} \rightarrow \{1, \dots, j-1, j+1, \dots, n\}$.

See also

QuantumMatrixCoordinateAlgebra

INPUT:

- *n* – the integer n
- *R* – (optional) the ring R if q is not specified (the default is $\mathbf{Z}$); otherwise the ring containing q
- *q* – (optional) the variable q ; the default is $q \in R[q, q^{-1}]$
- *bar* – (optional) the involution on the base ring; the default is $q \mapsto q^{-1}$

EXAMPLES:

We construct $\mathcal{O}_q(GL(3))$ and the variables:

```

sage: O = algebras.QuantumGL(3)
sage: O.inject_variables()
Defining x11, x12, x13, x21, x22, x23, x31, x32, x33, c

```

```
>>> from sage.all import *
>>> O = algebras.QuantumGL(Integer(3))
>>> O.inject_variables()
Defining x11, x12, x13, x21, x22, x23, x31, x32, x33, c
```

We do some basic computations:

```
sage: x33 * x12
x[1,2]*x[3,3] + (q^-1-q)*x[1,3]*x[3,2]
sage: x23 * x12 * x11
(q^-1)*x[1,1]*x[1,2]*x[2,3] + (q^-2-1)*x[1,1]*x[1,3]*x[2,2]
+ (q^-3-q^-1)*x[1,2]*x[1,3]*x[2,1]
sage: c * O.quantum_determinant()
1
```

```
>>> from sage.all import *
>>> x33 * x12
x[1,2]*x[3,3] + (q^-1-q)*x[1,3]*x[3,2]
>>> x23 * x12 * x11
(q^-1)*x[1,1]*x[1,2]*x[2,3] + (q^-2-1)*x[1,1]*x[1,3]*x[2,2]
+ (q^-3-q^-1)*x[1,2]*x[1,3]*x[2,1]
>>> c * O.quantum_determinant()
1
```

We verify the quantum determinant is in the center and is group-like:

```
sage: qdet = O.quantum_determinant()
sage: all(qdet * g == g * qdet for g in O.algebra_generators())
True
sage: qdet.coproduct() == tensor([qdet, qdet])
True
```

```
>>> from sage.all import *
>>> qdet = O.quantum_determinant()
>>> all(qdet * g == g * qdet for g in O.algebra_generators())
True
>>> qdet.coproduct() == tensor([qdet, qdet])
True
```

We check that the inverse of the quantum determinant is also in the center and group-like:

```
sage: all(c * g == g * c for g in O.algebra_generators())
True
sage: c.coproduct() == tensor([c, c])
True
```

```
>>> from sage.all import *
>>> all(c * g == g * c for g in O.algebra_generators())
True
>>> c.coproduct() == tensor([c, c])
True
```

Moreover, the antipode interchanges the quantum determinant and its inverse:

```
sage: c.antipode() == qdet
True
sage: qdet.antipode() == c
True
```

```
>>> from sage.all import *
>>> c.antipode() == qdet
True
>>> qdet.antipode() == c
True
```

REFERENCES:

- [DD1991]
- [Kar1993]

algebra_generators()

Return the algebra generators of `self`.

EXAMPLES:

```
sage: O = algebras.QuantumGL(2)
sage: O.algebra_generators()
Finite family {(1, 1): x[1,1], (1, 2): x[1,2], (2, 1): x[2,1], (2, 2): x[2,2],
↪ 'c': c}
```

```
>>> from sage.all import *
>>> O = algebras.QuantumGL(Integer(2))
>>> O.algebra_generators()
Finite family {(1, 1): x[1,1], (1, 2): x[1,2], (2, 1): x[2,1], (2, 2): x[2,2],
↪ 'c': c}
```

antipode_on_basis(x)

Return the antipode of the basis element indexed by `x`.

EXAMPLES:

```
sage: O = algebras.QuantumGL(3)
sage: x = O.indices().monoid_generators()
sage: O.antipode_on_basis(x[1,2])
-(q^-1)*c*x[1,2]*x[3,3] + c*x[1,3]*x[3,2]
sage: O.antipode_on_basis(x[2,2])
c*x[1,1]*x[3,3] - q*c*x[1,3]*x[3,1]
sage: O.antipode_on_basis(x['c']) == O.quantum_determinant()
True
```

```
>>> from sage.all import *
>>> O = algebras.QuantumGL(Integer(3))
>>> x = O.indices().monoid_generators()
>>> O.antipode_on_basis(x[Integer(1), Integer(2)])
-(q^-1)*c*x[1,2]*x[3,3] + c*x[1,3]*x[3,2]
>>> O.antipode_on_basis(x[Integer(2), Integer(2)])
c*x[1,1]*x[3,3] - q*c*x[1,3]*x[3,1]
```

(continues on next page)

(continued from previous page)

```
>>> O.antipode_on_basis(x['c']) == O.quantum_determinant()
True
```

coproduct_on_basis(*x*)Return the coproduct on the basis element indexed by *x*.

EXAMPLES:

```
sage: O = algebras.QuantumGL(3)
sage: x = O.indices().monoid_generators()
sage: O.coproduct_on_basis(x[1,2])
x[1,1] # x[1,2] + x[1,2] # x[2,2] + x[1,3] # x[3,2]
sage: O.coproduct_on_basis(x[2,2])
x[2,1] # x[1,2] + x[2,2] # x[2,2] + x[2,3] # x[3,2]
sage: O.coproduct_on_basis(x['c'])
c # c
```

```
>>> from sage.all import *
>>> O = algebras.QuantumGL(Integer(3))
>>> x = O.indices().monoid_generators()
>>> O.coproduct_on_basis(x[Integer(1), Integer(2)])
x[1,1] # x[1,2] + x[1,2] # x[2,2] + x[1,3] # x[3,2]
>>> O.coproduct_on_basis(x[Integer(2), Integer(2)])
x[2,1] # x[1,2] + x[2,2] # x[2,2] + x[2,3] # x[3,2]
>>> O.coproduct_on_basis(x['c'])
c # c
```

product_on_basis(*a*, *b*)Return the product of basis elements indexed by *a* and *b*.

EXAMPLES:

```
sage: O = algebras.QuantumGL(2)
sage: I = O.indices().monoid_generators()
sage: O.product_on_basis(I[1,1], I[2,2])
x[1,1]*x[2,2]
sage: O.product_on_basis(I[2,2], I[1,1])
x[1,1]*x[2,2] + (q^-1-q)*x[1,2]*x[2,1]
```

```
>>> from sage.all import *
>>> O = algebras.QuantumGL(Integer(2))
>>> I = O.indices().monoid_generators()
>>> O.product_on_basis(I[Integer(1), Integer(1)], I[Integer(2), Integer(2)])
x[1,1]*x[2,2]
>>> O.product_on_basis(I[Integer(2), Integer(2)], I[Integer(1), Integer(1)])
x[1,1]*x[2,2] + (q^-1-q)*x[1,2]*x[2,1]
```

```
class sage.algebras.quantum_matrix_coordinate_algebra.QuantumMatrixCoordinateAlgebra(m,
                                                                                       n,
                                                                                       q,
                                                                                       bar,
                                                                                       R)
```

Bases: *QuantumMatrixCoordinateAlgebra_abstract*

A quantum matrix coordinate algebra.

Let R be a commutative ring. The quantum matrix coordinate algebra of $M(m, n)$ is the associative algebra over $R[q, q^{-1}]$ generated by x_{ij} , for $i = 1, 2, \dots, m$, $j = 1, 2, \dots, n$, and subject to the following relations:

$$\begin{aligned} x_{it}x_{ij} &= q^{-1}x_{ij}x_{it} && \text{if } j < t, \\ x_{sj}x_{ij} &= q^{-1}x_{ij}x_{sj} && \text{if } i < s, \\ x_{st}x_{ij} &= x_{ij}x_{st} && \text{if } i < s, j > t, \\ x_{st}x_{ij} &= x_{ij}x_{st} + (q^{-1} - q)x_{it}x_{sj} && \text{if } i < s, j < t. \end{aligned}$$

The quantum matrix coordinate algebra is denoted by $\mathcal{O}_q(M(m, n))$. For $m = n$, it is also a bialgebra given by

$$\Delta(x_{ij}) = \sum_{k=1}^n x_{ik} \otimes x_{kj}, \varepsilon(x_{ij}) = \delta_{ij}.$$

Moreover, there is a central group-like element called the *quantum determinant* that is defined by

$$\det_q = \sum_{\sigma \in S_n} (-q)^{\ell(\sigma)} x_{1, \sigma(1)} x_{2, \sigma(2)} \cdots x_{n, \sigma(n)}.$$

The quantum matrix coordinate algebra also has natural inclusions when restricting to submatrices. That is, let $I \subseteq \{1, 2, \dots, m\}$ and $J \subseteq \{1, 2, \dots, n\}$. Then the subalgebra generated by $\{x_{ij} \mid i \in I, j \in J\}$ is naturally isomorphic to $\mathcal{O}_q(M(|I|, |J|))$.

Note

The q considered here is q^2 in some references, e.g., [ZZ2005].

INPUT:

- `m` – the integer m
- `n` – the integer n
- `R` – (optional) the ring R if q is not specified (the default is $\mathbf{Z}$); otherwise the ring containing q
- `q` – (optional) the variable q ; the default is $q \in R[q, q^{-1}]$
- `bar` – (optional) the involution on the base ring; the default is $q \mapsto q^{-1}$

EXAMPLES:

We construct $\mathcal{O}_q(M(2, 3))$ and the variables:

```
sage: O = algebras.QuantumMatrixCoordinate(2, 3)
sage: O.inject_variables()
Defining x11, x12, x13, x21, x22, x23
```

```
>>> from sage.all import *
>>> O = algebras.QuantumMatrixCoordinate(Integer(2), Integer(3))
>>> O.inject_variables()
Defining x11, x12, x13, x21, x22, x23
```

We do some basic computations:

```
sage: x21 * x11
(q^-1)*x[1,1]*x[2,1]
sage: x23 * x12 * x11
(q^-1)*x[1,1]*x[1,2]*x[2,3] + (q^-2-1)*x[1,1]*x[1,3]*x[2,2]
+ (q^-3-q^-1)*x[1,2]*x[1,3]*x[2,1]
```

```
>>> from sage.all import *
>>> x21 * x11
(q^-1)*x[1,1]*x[2,1]
>>> x23 * x12 * x11
(q^-1)*x[1,1]*x[1,2]*x[2,3] + (q^-2-1)*x[1,1]*x[1,3]*x[2,2]
+ (q^-3-q^-1)*x[1,2]*x[1,3]*x[2,1]
```

We construct the maximal quantum minors:

```
sage: q = O.q()
sage: qm12 = x11*x22 - q*x12*x21
sage: qm13 = x11*x23 - q*x13*x21
sage: qm23 = x12*x23 - q*x13*x22
```

```
>>> from sage.all import *
>>> q = O.q()
>>> qm12 = x11*x22 - q*x12*x21
>>> qm13 = x11*x23 - q*x13*x21
>>> qm23 = x12*x23 - q*x13*x22
```

However, unlike for the quantum determinant, they are not central:

```
sage: all(qm12 * g == g * qm12 for g in O.algebra_generators())
False
sage: all(qm13 * g == g * qm13 for g in O.algebra_generators())
False
sage: all(qm23 * g == g * qm23 for g in O.algebra_generators())
False
```

```
>>> from sage.all import *
>>> all(qm12 * g == g * qm12 for g in O.algebra_generators())
False
>>> all(qm13 * g == g * qm13 for g in O.algebra_generators())
False
>>> all(qm23 * g == g * qm23 for g in O.algebra_generators())
False
```

REFERENCES:

- [FRT1990]
- [ZZ2005]

`algebra_generators()`

Return the algebra generators of `self`.

EXAMPLES:

```
sage: O = algebras.QuantumMatrixCoordinate(2)
sage: O.algebra_generators()
Finite family {(1, 1): x[1,1], (1, 2): x[1,2], (2, 1): x[2,1], (2, 2): x[2,2]}
```

```
>>> from sage.all import *
>>> O = algebras.QuantumMatrixCoordinate(Integer(2))
>>> O.algebra_generators()
Finite family {(1, 1): x[1,1], (1, 2): x[1,2], (2, 1): x[2,1], (2, 2): x[2,2]}
```

coproduct_on_basis(x)

Return the coproduct on the basis element indexed by x .

EXAMPLES:

```
sage: O = algebras.QuantumMatrixCoordinate(4)
sage: x24 = O.algebra_generators()[2,4]
sage: O.coproduct_on_basis(x24.leading_support())
x[2,1] # x[1,4] + x[2,2] # x[2,4] + x[2,3] # x[3,4] + x[2,4] # x[4,4]
```

```
>>> from sage.all import *
>>> O = algebras.QuantumMatrixCoordinate(Integer(4))
>>> x24 = O.algebra_generators()[Integer(2), Integer(4)]
>>> O.coproduct_on_basis(x24.leading_support())
x[2,1] # x[1,4] + x[2,2] # x[2,4] + x[2,3] # x[3,4] + x[2,4] # x[4,4]
```

m()

Return the value m .

EXAMPLES:

```
sage: O = algebras.QuantumMatrixCoordinate(4, 6)
sage: O.m()
4
sage: O = algebras.QuantumMatrixCoordinate(4)
sage: O.m()
4
```

```
>>> from sage.all import *
>>> O = algebras.QuantumMatrixCoordinate(Integer(4), Integer(6))
>>> O.m()
4
>>> O = algebras.QuantumMatrixCoordinate(Integer(4))
>>> O.m()
4
```

class sage.algebras.quantum_matrix_coordinate_algebra.QuantumMatrixCoordinateAlgebra_abstract (*gp_indices,*
n,
q,
bar,
R,
category,
indices_key,

Bases: CombinatorialFreeModule

Abstract base class for quantum coordinate algebras of a set of matrices.

class Element

Bases: IndexedFreeModuleElement

An element of a quantum matrix coordinate algebra.

bar()

Return the image of `self` under the bar involution.

The bar involution is the $\mathbf{Q}$ -algebra anti-automorphism defined by $x_{ij} \mapsto x_{ji}$ and $q \mapsto q^{-1}$.

EXAMPLES:

```
sage: O = algebras.QuantumMatrixCoordinate(4)
sage: x = O.an_element()
sage: x.bar()
1 + 2*x[1,1] + (q^-16)*x[1,1]^2*x[1,2]^2*x[1,3]^3 + 3*x[1,2]
sage: x = O.an_element() * O.algebra_generators()[2,4]; x
x[1,1]^2*x[1,2]^2*x[1,3]^3*x[2,4] + 2*x[1,1]*x[2,4]
+ 3*x[1,2]*x[2,4] + x[2,4]
sage: xb = x.bar(); xb
(q^-16)*x[1,1]^2*x[1,2]^2*x[1,3]^3*x[2,4]
+ (q^-21-q^-15)*x[1,1]^2*x[1,2]^2*x[1,3]^2*x[1,4]*x[2,3]
+ (q^-22-q^-18)*x[1,1]^2*x[1,2]*x[1,3]^3*x[1,4]*x[2,2]
+ (q^-24-q^-20)*x[1,1]*x[1,2]^2*x[1,3]^3*x[1,4]*x[2,1]
+ 2*x[1,1]*x[2,4] + 3*x[1,2]*x[2,4]
+ (2*q^-1-2*q)*x[1,4]*x[2,1]
+ (3*q^-1-3*q)*x[1,4]*x[2,2] + x[2,4]
sage: xb.bar() == x
True
```

```
>>> from sage.all import *
>>> O = algebras.QuantumMatrixCoordinate(Integer(4))
>>> x = O.an_element()
>>> x.bar()
1 + 2*x[1,1] + (q^-16)*x[1,1]^2*x[1,2]^2*x[1,3]^3 + 3*x[1,2]
>>> x = O.an_element() * O.algebra_generators()[Integer(2), Integer(4)]; x
x[1,1]^2*x[1,2]^2*x[1,3]^3*x[2,4] + 2*x[1,1]*x[2,4]
+ 3*x[1,2]*x[2,4] + x[2,4]
>>> xb = x.bar(); xb
(q^-16)*x[1,1]^2*x[1,2]^2*x[1,3]^3*x[2,4]
```

(continues on next page)

(continued from previous page)

```

+ (q^-21-q^-15)*x[1,1]^2*x[1,2]^2*x[1,3]^2*x[1,4]*x[2,3]
+ (q^-22-q^-18)*x[1,1]^2*x[1,2]*x[1,3]^3*x[1,4]*x[2,2]
+ (q^-24-q^-20)*x[1,1]*x[1,2]^2*x[1,3]^3*x[1,4]*x[2,1]
+ 2*x[1,1]*x[2,4] + 3*x[1,2]*x[2,4]
+ (2*q^-1-2*q)*x[1,4]*x[2,1]
+ (3*q^-1-3*q)*x[1,4]*x[2,2] + x[2,4]
>>> xb.bar() == x
True

```

counit_on_basis(*x*)

Return the counit on the basis element indexed by *x*.

EXAMPLES:

```

sage: O = algebras.QuantumMatrixCoordinate(4)
sage: G = O.algebra_generators()
sage: I = [1,2,3,4]
sage: matrix([[G[i,j].counit() for i in I] for j in I]) # indirect doctest
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]

```

```

>>> from sage.all import *
>>> O = algebras.QuantumMatrixCoordinate(Integer(4))
>>> G = O.algebra_generators()
>>> I = [Integer(1), Integer(2), Integer(3), Integer(4)]
>>> matrix([[G[i,j].counit() for i in I] for j in I]) # indirect doctest
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]

```

gens()

Return the generators of *self* as a tuple.

EXAMPLES:

```

sage: O = algebras.QuantumMatrixCoordinate(3)
sage: O.gens()
(x[1,1], x[1,2], x[1,3],
 x[2,1], x[2,2], x[2,3],
 x[3,1], x[3,2], x[3,3])

```

```

>>> from sage.all import *
>>> O = algebras.QuantumMatrixCoordinate(Integer(3))
>>> O.gens()
(x[1,1], x[1,2], x[1,3],
 x[2,1], x[2,2], x[2,3],
 x[3,1], x[3,2], x[3,3])

```

n()

Return the value *n*.

EXAMPLES:

```
sage: O = algebras.QuantumMatrixCoordinate(4)
sage: O.n()
4
sage: O = algebras.QuantumMatrixCoordinate(4, 6)
sage: O.n()
6
```

```
>>> from sage.all import *
>>> O = algebras.QuantumMatrixCoordinate(Integer(4))
>>> O.n()
4
>>> O = algebras.QuantumMatrixCoordinate(Integer(4), Integer(6))
>>> O.n()
6
```

one_basis()

Return the basis element indexing 1.

EXAMPLES:

```
sage: O = algebras.QuantumMatrixCoordinate(4)
sage: O.one_basis()
1
sage: O.one()
1
```

```
>>> from sage.all import *
>>> O = algebras.QuantumMatrixCoordinate(Integer(4))
>>> O.one_basis()
1
>>> O.one()
1
```

product_on_basis(a, b)

Return the product of basis elements indexed by a and b.

EXAMPLES:

```
sage: O = algebras.QuantumMatrixCoordinate(4)
sage: x = O.algebra_generators()
sage: b = x[1,4] * x[2,1] * x[3,4] # indirect doctest
sage: b * (b * b) == (b * b) * b
True
sage: p = prod(list(O.algebra_generators())[:10])
sage: p * (p * p) == (p * p) * p # long time
True
sage: x = O.an_element()
sage: y = x^2 + x[4,4] * x[3,3] * x[1,2]
sage: z = x[2,2] * x[1,4] * x[3,4] * x[1,1]
sage: x * (y * z) == (x * y) * z
True
```

```

>>> from sage.all import *
>>> O = algebras.QuantumMatrixCoordinate(Integer(4))
>>> x = O.algebra_generators()
>>> b = x[Integer(1), Integer(4)] * x[Integer(2), Integer(1)] * x[Integer(3),
↳ Integer(4)] # indirect doctest
>>> b * (b * b) == (b * b) * b
True
>>> p = prod(list(O.algebra_generators())[:Integer(10)])
>>> p * (p * p) == (p * p) * p # long time
True
>>> x = O.an_element()
>>> y = x**Integer(2) + x[Integer(4), Integer(4)] * x[Integer(3), Integer(3)] *
↳ x[Integer(1), Integer(2)]
>>> z = x[Integer(2), Integer(2)] * x[Integer(1), Integer(4)] * x[Integer(3),
↳ Integer(4)] * x[Integer(1), Integer(1)]
>>> x * (y * z) == (x * y) * z
True

```

q()Return the variable q .

EXAMPLES:

```

sage: O = algebras.QuantumMatrixCoordinate(4)
sage: O.q()
q
sage: O.q().parent()
Univariate Laurent Polynomial Ring in q over Integer Ring
sage: O.q().parent() is O.base_ring()
True

```

```

>>> from sage.all import *
>>> O = algebras.QuantumMatrixCoordinate(Integer(4))
>>> O.q()
q
>>> O.q().parent()
Univariate Laurent Polynomial Ring in q over Integer Ring
>>> O.q().parent() is O.base_ring()
True

```

quantum_determinant()Return the quantum determinant of `self`.

The quantum determinant is defined by

$$\det_q = \sum_{\sigma \in S_n} (-q)^{\ell(\sigma)} x_{1, \sigma(1)} x_{2, \sigma(2)} \cdots x_{n, \sigma(n)}.$$

EXAMPLES:

```

sage: O = algebras.QuantumMatrixCoordinate(2)
sage: O.quantum_determinant()
x[1,1]*x[2,2] - q*x[1,2]*x[2,1]

```



```
>>> from sage.all import *
>>> O = algebras.QuantumMatrixCoordinate(Integer(2))
>>> O.quantum_determinant()
x[1,1]*x[2,2] - q*x[1,2]*x[2,1]
```

We verify that the quantum determinant is central:

```
sage: for n in range(2,5):
.....:     O = algebras.QuantumMatrixCoordinate(n)
.....:     qdet = O.quantum_determinant()
.....:     assert all(g * qdet == qdet * g for g in O.algebra_generators())
```

```
>>> from sage.all import *
>>> for n in range(Integer(2), Integer(5)):
...     O = algebras.QuantumMatrixCoordinate(n)
...     qdet = O.quantum_determinant()
...     assert all(g * qdet == qdet * g for g in O.algebra_generators())
```

We also verify that it is group-like:

```
sage: for n in range(2,4):
.....:     O = algebras.QuantumMatrixCoordinate(n)
.....:     qdet = O.quantum_determinant()
.....:     assert qdet.coproduct() == tensor([qdet, qdet])
```

```
>>> from sage.all import *
>>> for n in range(Integer(2), Integer(4)):
...     O = algebras.QuantumMatrixCoordinate(n)
...     qdet = O.quantum_determinant()
...     assert qdet.coproduct() == tensor([qdet, qdet])
```

5.20 Quantum Oscillator Algebras

AUTHORS:

- Travis Scrimshaw (2023-12): initial version

class sage.algebras.quantum_oscillator.**FockSpaceRepresentation**(*oscillator_algebra*)

Bases: `CombinatorialFreeModule`

The unique Fock space representation of the *QuantumOscillatorAlgebra*.

class `Element`

Bases: `IndexedFreeModuleElement`

some_elements()

Return some elements of self.

EXAMPLES:

```
sage: O = algebras.QuantumOscillator()
sage: F = O.fock_space_representation()
sage: F.some_elements()
(|0>, |1>, |52>, |0> + 2*|1> + 3*|2> + |42>)
```

```
>>> from sage.all import *
>>> O = algebras.QuantumOscillator()
>>> F = O.fock_space_representation()
>>> F.some_elements()
(|0>, |1>, |52>, |0> + 2*|1> + 3*|2> + |42>)
```

vacuum()

Return the vacuum element $|0\rangle$ of self.

EXAMPLES:

```
sage: O = algebras.QuantumOscillator()
sage: F = O.fock_space_representation()
sage: F.vacuum()
|0>
```

```
>>> from sage.all import *
>>> O = algebras.QuantumOscillator()
>>> F = O.fock_space_representation()
>>> F.vacuum()
|0>
```

class sage.algebras.quantum_oscillator.QuantumOscillatorAlgebra(q, R)

Bases: CombinatorialFreeModule

The quantum oscillator algebra.

Let R be a commutative algebra and $q \in R$ be a unit. The *quantum oscillator algebra*, or q -oscillator algebra, is the unital associative R -algebra with generators a^+ , a^- and $k^{\pm 1}$ satisfying the following relations:

$$ka^{\pm} = q^{\pm 1}a^{\pm}k, \quad a^-a^+ = 1 - q^2k^2, \quad a^+a^- = 1 - k^2.$$

INPUT:

- q – (optional) the parameter q
- R – (default: $\mathbf{Q}(q)$) the base ring that contains q

EXAMPLES:

We construct the algebra and perform some basic computations:

```
sage: O = algebras.QuantumOscillator()
sage: ap, am, k, ki = O.algebra_generators()
sage: q = O.q()
sage: k^-3 * ap * ki * am^2 * k - q^3 * ap * k^3
q^5*a-*k^-3 - q^3*a-*k^-1 - q^3*a+k^3
```

```
>>> from sage.all import *
>>> O = algebras.QuantumOscillator()
>>> ap, am, k, ki = O.algebra_generators()
>>> q = O.q()
>>> k**Integer(3) * ap * ki * am**Integer(2) * k - q**Integer(3) * ap *
↪ k**Integer(3)
q^5*a-*k^-3 - q^3*a-*k^-1 - q^3*a+k^3
```

We construct representations of the type A_1 quantum coordinate ring using the quantum oscillator algebra and verify the quantum determinant:

```
sage: pi = matrix([[am, k], [-q*k, ap]]); pi
[ a-      k]
[-q*k      a+]
sage: pi[0,0] * pi[1,1] - q * pi[0,1] * pi[1,0]
1
```

```
>>> from sage.all import *
>>> pi = matrix([[am, k], [-q*k, ap]]); pi
[ a-      k]
[-q*k      a+]
>>> pi[Integer(0),Integer(0)] * pi[Integer(1),Integer(1)] - q * pi[Integer(0),
↪Integer(1)] * pi[Integer(1),Integer(0)]
1
```

Next, we use this to build representations for type A_2 :

```
sage: def quantum_det(M):
.....:     n = M.nrows()
.....:     return sum((-q)**sigma.length()
.....:                * prod(M[i,sigma[i]-1] for i in range(n))
.....:                for sigma in Permutations(n))
sage: def build_repr(wd, gens):
.....:     n = gens[0].nrows()
.....:     ret = gens[wd[0]-1]
.....:     for ind in wd[1:]:
.....:         g = gens[ind-1]
.....:         temp = [[None]*n for _ in range(n)]
.....:         for i in range(n):
.....:             for j in range(n):
.....:                 temp[i][j] = sum(tensor([ret[i,k], g[k,j]]) for k in
↪range(n))
.....:     ret = matrix(temp)
.....:     return ret
sage: pi1 = matrix.block_diagonal(pi, matrix.identity(1)); pi1
[ a-      k|  0]
[-q*k      a+|  0]
[-----+----]
[  0      0|  1]
sage: pi2 = matrix.block_diagonal(matrix.identity(1), pi); pi2
[  1|  0      0]
[----+-----]
[  0| a-      k]
[  0|-q*k      a+]
sage: quantum_det(pi1) == 1
True
sage: quantum_det(pi2) == 1
True
sage: pi12 = build_repr([1,2], [pi1, pi2]); pi12
[ a- # 1      k # a-      k # k]
[-q*k # 1      a+ # a-      a+ # k]
```

(continues on next page)

(continued from previous page)

```

[      0 -q*1 # k      1 # a+]
sage: quantum_det(pi12)
1 # 1
sage: pi121 = build_repr([1,2,1], [pi1, pi2]); pi121
[  a- # 1 # a- - q*k # a- # k      a- # 1 # k + k # a- # a+
↪k # k # 1]
[-q*k # 1 # a- - q*a+ # a- # k      -q*k # 1 # k + a+ # a- # a+
↪a+ # k # 1]
[      q^2*1 # k # k      -q*1 # k # a+      1
↪# a+ # 1]
sage: quantum_det(pi121)
1 # 1 # 1
sage: pi212 = build_repr([2,1,2], [pi1, pi2]); pi212
[      1 # a- # 1      1 # k # a-
↪1 # k # k]
[      -q*a- # k # 1      a- # a+ # a- - q*k # 1 # k      a- # a+ # k + k
↪# 1 # a+]
[      q^2*k # k # 1 -q*k # a+ # a- - q*a+ # 1 # k      -q*k # a+ # k + a+
↪# 1 # a+]
sage: quantum_det(pi212)
1 # 1 # 1

```

```

>>> from sage.all import *
>>> def quantum_det(M):
...     n = M.nrows()
...     return sum((-q)**sigma.length()
...               * prod(M[i,sigma[i]-Integer(1)] for i in range(n))
...               for sigma in Permutations(n))
>>> def build_repr(wd, gens):
...     n = gens[Integer(0)].nrows()
...     ret = gens[wd[Integer(0)]]-Integer(1)]
...     for ind in wd[Integer(1):]:
...         g = gens[ind-Integer(1)]
...         temp = [[None]*n for _ in range(n)]
...         for i in range(n):
...             for j in range(n):
...                 temp[i][j] = sum(tensor([ret[i,k], g[k,j]]) for k in range(n))
...         ret = matrix(temp)
...     return ret
>>> pi1 = matrix.block_diagonal(pi, matrix.identity(Integer(1))); pi1
[  a-   k|   0]
[-q*k   a+|   0]
[-----+----]
[   0   0|   1]
>>> pi2 = matrix.block_diagonal(matrix.identity(Integer(1)), pi); pi2
[   1|   0   0]
[----+-----]
[   0|  a-   k]
[   0|-q*k  a+]
>>> quantum_det(pi1) == Integer(1)
True
>>> quantum_det(pi2) == Integer(1)

```

(continues on next page)

(continued from previous page)

```

True
>>> pi12 = build_repr([Integer(1),Integer(2)], [pi1, pi2]); pi12
[ a- # 1    k # a-    k # k]
[-q*k # 1   a+ # a-    a+ # k]
[          0 -q*1 # k    1 # a+]
>>> quantum_det(pi12)
1 # 1
>>> pi121 = build_repr([Integer(1),Integer(2),Integer(1)], [pi1, pi2]); pi121
[  a- # 1 # a- - q*k # a- # k      a- # 1 # k + k # a- # a+
↪k # k # 1]
[-q*k # 1 # a- - q*a+ # a- # k    -q*k # 1 # k + a+ # a- # a+
↪a+ # k # 1]
[          q^2*1 # k # k      -q*1 # k # a+      1
↪# a+ # 1]
>>> quantum_det(pi121)
1 # 1 # 1
>>> pi212 = build_repr([Integer(2),Integer(1),Integer(2)], [pi1, pi2]); pi212
[          1 # a- # 1      1 # k # a-
↪1 # k # k]
[          -q*a- # k # 1    a- # a+ # a- - q*k # 1 # k      a- # a+ # k + k
↪# 1 # a+]
[          q^2*k # k # 1 -q*k # a+ # a- - q*a+ # 1 # k    -q*k # a+ # k + a+
↪# 1 # a+]
>>> quantum_det(pi212)
1 # 1 # 1

```

REFERENCES:

- [Kuniba2022] Section 3.2

class ElementBases: `IndexedFreeModuleElement`**algebra_generators()**Return the algebra generators of `self`.

EXAMPLES:

```

sage: O = algebras.QuantumOscillator()
sage: O.algebra_generators()
Finite family {'am': a-, 'ap': a+, 'k': k, 'ki': k^-1}

```

```

>>> from sage.all import *
>>> O = algebras.QuantumOscillator()
>>> O.algebra_generators()
Finite family {'am': a-, 'ap': a+, 'k': k, 'ki': k^-1}

```

fock_space_representation()Return the Fock space representation of `self`.**See also**[`FockSpaceRepresentation`](#)

EXAMPLES:

```
sage: O = algebras.QuantumOscillator()
sage: O.fock_space_representation()
Fock space representation of Quantum oscillator algebra with q=q
over Fraction Field of Univariate Polynomial Ring in q over Integer Ring
```

```
>>> from sage.all import *
>>> O = algebras.QuantumOscillator()
>>> O.fock_space_representation()
Fock space representation of Quantum oscillator algebra with q=q
over Fraction Field of Univariate Polynomial Ring in q over Integer Ring
```

gens()

Return the generators of self.

EXAMPLES:

```
sage: O = algebras.QuantumOscillator()
sage: O.gens()
(a+, a-, k, k^-1)
```

```
>>> from sage.all import *
>>> O = algebras.QuantumOscillator()
>>> O.gens()
(a+, a-, k, k^-1)
```

one_basis()

Return the index of the basis element of 1.

EXAMPLES:

```
sage: O = algebras.QuantumOscillator()
sage: O.one_basis()
(0, 0)
```

```
>>> from sage.all import *
>>> O = algebras.QuantumOscillator()
>>> O.one_basis()
(0, 0)
```

product_on_basis(ml, mr)

Return the product of the basis elements indexed by ml and mr.

EXAMPLES:

```
sage: O = algebras.QuantumOscillator()
sage: ap, am, k, ki = O.algebra_generators()
sage: O.product_on_basis((-2, 3), (-4, 5))
1/q^12*a^-6*k^8
sage: O.product_on_basis((2, 3), (4, -5))
q^12*a+^6*k^-2
sage: O.product_on_basis((2, 3), (0, -3))
a+^2
```

(continues on next page)

(continued from previous page)

```

sage: k^5 * ki^10
k^-5
sage: k^10 * ki^5
k^5
sage: ap^3 * k^5
a+^3*k^5
sage: am^3 * k^5
a-^3*k^5
sage: k^5 * ap^3
q^15*a+^3*k^5
sage: k^5 * am^3
1/q^15*a-^3*k^5
sage: ki^5 * ap^3
1/q^15*a+^3*k^-5
sage: ki^5 * am^3
q^15*a-^3*k^-5
sage: ap * am
1 - k^2
sage: am * ap
1 - q^2*k^2

sage: (ap + am + k + ki)^2
a-^2 + (q+1)*a-*k^-1 + ((q+1)/q)*a-*k + k^-2 + 4 - q^2*k^2
+ ((q+1)/q)*a+*k^-1 + (q+1)*a+*k + a+^2

sage: (ap)^3 * (am)^5
a-^2 + ((-q^4-q^2-1)/q^8)*a-^2*k^2 + ((q^4+q^2+1)/q^14)*a-^2*k^4 - 1/q^18*a-^
↪2*k^6
sage: (ap)^5 * (am)^3
a+^2 + ((-q^4-q^2-1)/q^4)*a+^2*k^2 + ((q^4+q^2+1)/q^6)*a+^2*k^4 - 1/q^6*a+^
↪2*k^6
sage: (am)^3 * (ap)^5
a+^2 + (-q^10-q^8-q^6)*a+^2*k^2 + (q^18+q^16+q^14)*a+^2*k^4 - q^24*a+^2*k^6
sage: (am)^5 * (ap)^3
a-^2 + (-q^6-q^4-q^2)*a-^2*k^2 + (q^10+q^8+q^6)*a-^2*k^4 - q^12*a-^2*k^6

```

```

>>> from sage.all import *
>>> O = algebras.QuantumOscillator()
>>> ap, am, k, ki = O.algebra_generators()
>>> O.product_on_basis((-Integer(2), Integer(3)), (-Integer(4), Integer(5)))
1/q^12*a-^6*k^8
>>> O.product_on_basis((Integer(2), Integer(3)), (Integer(4), -Integer(5)))
q^12*a+^6*k^-2
>>> O.product_on_basis((Integer(2), Integer(3)), (Integer(0), -Integer(3)))
a+^2
>>> k**Integer(5) * ki**Integer(10)
k^-5
>>> k**Integer(10) * ki**Integer(5)
k^5
>>> ap**Integer(3) * k**Integer(5)
a+^3*k^5
>>> am**Integer(3) * k**Integer(5)

```

(continues on next page)

(continued from previous page)

```

a^-3*k^5
>>> k**Integer(5) * ap**Integer(3)
q^15*a+^3*k^5
>>> k**Integer(5) * am**Integer(3)
1/q^15*a-^3*k^5
>>> ki**Integer(5) * ap**Integer(3)
1/q^15*a+^3*k^-5
>>> ki**Integer(5) * am**Integer(3)
q^15*a-^3*k^-5
>>> ap * am
1 - k^2
>>> am * ap
1 - q^2*k^2

>>> (ap + am + k + ki)**Integer(2)
a^-2 + (q+1)*a-*k^-1 + ((q+1)/q)*a-*k + k^-2 + 4 - q^2*k^2
+ ((q+1)/q)*a+*k^-1 + (q+1)*a+*k + a+^2

>>> (ap)**Integer(3) * (am)**Integer(5)
a^-2 + ((-q^4-q^2-1)/q^8)*a-^2*k^2 + ((q^4+q^2+1)/q^14)*a-^2*k^4 - 1/q^18*a-^
↪2*k^6
>>> (ap)**Integer(5) * (am)**Integer(3)
a+^2 + ((-q^4-q^2-1)/q^4)*a+^2*k^2 + ((q^4+q^2+1)/q^6)*a+^2*k^4 - 1/q^6*a+^
↪2*k^6
>>> (am)**Integer(3) * (ap)**Integer(5)
a+^2 + (-q^10-q^8-q^6)*a+^2*k^2 + (q^18+q^16+q^14)*a+^2*k^4 - q^24*a+^2*k^6
>>> (am)**Integer(5) * (ap)**Integer(3)
a-^2 + (-q^6-q^4-q^2)*a-^2*k^2 + (q^10+q^8+q^6)*a-^2*k^4 - q^12*a-^2*k^6

```

q()Return the q of self.

EXAMPLES:

```

sage: O = algebras.QuantumOscillator()
sage: O.q()
q
sage: O = algebras.QuantumOscillator(q=QQ(-5))
sage: O.q()
-5

```

```

>>> from sage.all import *
>>> O = algebras.QuantumOscillator()
>>> O.q()
q
>>> O = algebras.QuantumOscillator(q=QQ(-Integer(5)))
>>> O.q()
-5

```

some_elements()

Return some elements of self.

EXAMPLES:


```
sage: O = algebras.QuantumOscillator()
sage: O.some_elements()
(a+, a-, k, k^-1, 1, a+^3, a-^4, k^2, k^-5, a+k,
a-^4*k^-3, 1 + 3*k + 2*a+ + a+k)
```

```
>>> from sage.all import *
>>> O = algebras.QuantumOscillator()
>>> O.some_elements()
(a+, a-, k, k^-1, 1, a+^3, a-^4, k^2, k^-5, a+k,
a-^4*k^-3, 1 + 3*k + 2*a+ + a+k)
```

5.21 Quaternion Algebras

AUTHORS:

- Jon Bobber (2009): rewrite
- William Stein (2009): rewrite
- Julian Rueth (2014-03-02): use UniqueFactory for caching
- Peter Bruin (2021): do not require the base ring to be a field
- Lorenz Panny (2022): `QuaternionOrder.isomorphism_to()`, `QuaternionFractionalIdeal_rational.minimal_element()`
- Sebastian A. Spindler (2024): extend ramification functionality to number fields, adapt `QuaternionAlgebra_ab.maximal_order()` to allow for extension of an order
- Eloi Torrents (2024): construct quaternion algebras over number fields from ramification

This code is partly based on Sage code by David Kohel from 2005.

```
class sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebraFactory
```

Bases: `UniqueFactory`

Construct a quaternion algebra.

INPUT:

There are four input formats:

- `QuaternionAlgebra(a, b)`, where a and b can be coerced to units in a common field K of characteristic different from 2.
- `QuaternionAlgebra(K, a, b)`, where K is a ring in which 2 is a unit and a and b are units of K .
- `QuaternionAlgebra(D)`, where $D \geq 1$ is a squarefree integer. This constructs a quaternion algebra of discriminant D over $K = \mathbf{Q}$. Suitable nonzero rational numbers a, b as above are deduced from D .
- `QuaternionAlgebra(K, primes, inv_archimedean)`, where K is a number field or $\mathbf{Q}$, primes is a list of prime ideals of K and inv_archimedean is a list of local invariants (0 or $\frac{1}{2}$) specifying the ramification at the (infinite) real places of K . This constructs a quaternion algebra ramified exactly at the places given by primes and those (algebraic) real embeddings of K indexed in $K.\text{embeddings}(\text{AA})$ by 1 with $\text{inv\_archimedean}[i] = 1/2$.

OUTPUT:

The quaternion algebra $(a, b)_K$ over K generated by i, j subject to $i^2 = a$, $j^2 = b$, and $ji = -ij$.

EXAMPLES:

`QuaternionAlgebra(a, b)` – return the quaternion algebra $(a, b)_K$, where the base ring K is a suitably chosen field containing a and b :

```
sage: QuaternionAlgebra(-2,-3)
Quaternion Algebra (-2, -3) with base ring Rational Field
sage: QuaternionAlgebra(GF(5)(2), GF(5)(3))
Quaternion Algebra (2, 3) with base ring Finite Field of size 5
sage: QuaternionAlgebra(2, GF(5)(3))
Quaternion Algebra (2, 3) with base ring Finite Field of size 5
sage: QuaternionAlgebra(QQ[sqrt(2)](-1), -5) #_
↳needs sage.symbolic
Quaternion Algebra (-1, -5) with base ring Number Field in sqrt2
with defining polynomial x^2 - 2 with sqrt2 = 1.414213562373095?
sage: QuaternionAlgebra(sqrt(-1), sqrt(-3)) #_
↳needs sage.symbolic
Quaternion Algebra (I, sqrt(-3)) with base ring Symbolic Ring
sage: QuaternionAlgebra(1r,1)
Quaternion Algebra (1, 1) with base ring Rational Field
sage: A.<t> = ZZ[]
sage: QuaternionAlgebra(-1, t)
Quaternion Algebra (-1, t) with base ring
Fraction Field of Univariate Polynomial Ring in t over Integer Ring
```

```
>>> from sage.all import *
>>> QuaternionAlgebra(-Integer(2),-Integer(3))
Quaternion Algebra (-2, -3) with base ring Rational Field
>>> QuaternionAlgebra(GF(Integer(5))(Integer(2)), GF(Integer(5))(Integer(3)))
Quaternion Algebra (2, 3) with base ring Finite Field of size 5
>>> QuaternionAlgebra(Integer(2), GF(Integer(5))(Integer(3)))
Quaternion Algebra (2, 3) with base ring Finite Field of size 5
>>> QuaternionAlgebra(QQ[sqrt(Integer(2))](Integer(1)), -Integer(5)) _
↳ # needs sage.symbolic
Quaternion Algebra (-1, -5) with base ring Number Field in sqrt2
with defining polynomial x^2 - 2 with sqrt2 = 1.414213562373095?
>>> QuaternionAlgebra(sqrt(-Integer(1)), sqrt(-Integer(3))) _
↳ # needs sage.symbolic
Quaternion Algebra (I, sqrt(-3)) with base ring Symbolic Ring
>>> QuaternionAlgebra(1,Integer(1))
Quaternion Algebra (1, 1) with base ring Rational Field
>>> A = ZZ['t']; (t,) = A._first_ngens(1)
>>> QuaternionAlgebra(-Integer(1), t)
Quaternion Algebra (-1, t) with base ring
Fraction Field of Univariate Polynomial Ring in t over Integer Ring
```

Python ints and floats may be passed to the `QuaternionAlgebra(a, b)` constructor, as may all pairs of nonzero elements of a domain not of characteristic 2.

The following tests address the issues raised in [Issue #10601](#):

```
sage: QuaternionAlgebra(1r,1)
Quaternion Algebra (1, 1) with base ring Rational Field
sage: QuaternionAlgebra(1,1.0r)
Quaternion Algebra (1.000000000000000, 1.000000000000000) with base ring
Real Field with 53 bits of precision
```

(continues on next page)

(continued from previous page)

```

sage: QuaternionAlgebra(0,0)
Traceback (most recent call last):
...
ValueError: defining elements of quaternion algebra (0, 0)
are not invertible in Rational Field
sage: QuaternionAlgebra(GF(2)(1),1)
Traceback (most recent call last):
...
ValueError: 2 is not invertible in Finite Field of size 2
sage: a = PermutationGroupElement([1,2,3])
sage: QuaternionAlgebra(a, a)
Traceback (most recent call last):
...
ValueError: a and b must be elements of a ring with characteristic not 2

```

```

>>> from sage.all import *
>>> QuaternionAlgebra(1,Integer(1))
Quaternion Algebra (1, 1) with base ring Rational Field
>>> QuaternionAlgebra(Integer(1),1.0)
Quaternion Algebra (1.000000000000000, 1.000000000000000) with base ring
Real Field with 53 bits of precision
>>> QuaternionAlgebra(Integer(0),Integer(0))
Traceback (most recent call last):
...
ValueError: defining elements of quaternion algebra (0, 0)
are not invertible in Rational Field
>>> QuaternionAlgebra(GF(Integer(2))(Integer(1)),Integer(1))
Traceback (most recent call last):
...
ValueError: 2 is not invertible in Finite Field of size 2
>>> a = PermutationGroupElement([Integer(1),Integer(2),Integer(3)])
>>> QuaternionAlgebra(a, a)
Traceback (most recent call last):
...
ValueError: a and b must be elements of a ring with characteristic not 2

```

`QuaternionAlgebra(K, a, b)` – return the quaternion algebra defined by (a, b) over the ring K :

```

sage: QuaternionAlgebra(QQ, -7, -21)
Quaternion Algebra (-7, -21) with base ring Rational Field
sage: QuaternionAlgebra(QQ[sqrt(2)], -2,-3) #_
↪needs sage.symbolic
Quaternion Algebra (-2, -3) with base ring Number Field in sqrt2
with defining polynomial x^2 - 2 with sqrt2 = 1.414213562373095?

```

```

>>> from sage.all import *
>>> QuaternionAlgebra(QQ, -Integer(7), -Integer(21))
Quaternion Algebra (-7, -21) with base ring Rational Field
>>> QuaternionAlgebra(QQ[sqrt(Integer(2))], -Integer(2),-Integer(3)) #_
↪ # needs sage.symbolic
Quaternion Algebra (-2, -3) with base ring Number Field in sqrt2
with defining polynomial x^2 - 2 with sqrt2 = 1.414213562373095?

```

`QuaternionAlgebra(D)` – D is a squarefree integer; return a rational quaternion algebra of discriminant D :

```
sage: QuaternionAlgebra(1)
Quaternion Algebra (-1, 1) with base ring Rational Field
sage: QuaternionAlgebra(2)
Quaternion Algebra (-1, -1) with base ring Rational Field
sage: QuaternionAlgebra(7)
Quaternion Algebra (-1, -7) with base ring Rational Field
sage: QuaternionAlgebra(2*3*5*7)
Quaternion Algebra (-22, 210) with base ring Rational Field
```

```
>>> from sage.all import *
>>> QuaternionAlgebra(Integer(1))
Quaternion Algebra (-1, 1) with base ring Rational Field
>>> QuaternionAlgebra(Integer(2))
Quaternion Algebra (-1, -1) with base ring Rational Field
>>> QuaternionAlgebra(Integer(7))
Quaternion Algebra (-1, -7) with base ring Rational Field
>>> QuaternionAlgebra(Integer(2)*Integer(3)*Integer(5)*Integer(7))
Quaternion Algebra (-22, 210) with base ring Rational Field
```

`QuaternionAlgebra(K, primes, inv_archimedean)` – return the quaternion algebra over K with the ramification specified by `primes` and `inv_archimedean`:

```
sage: QuaternionAlgebra(QQ, [(2), (3)], [0])
Quaternion Algebra (-1, 3) with base ring Rational Field
sage: QuaternionAlgebra(QQ, [(2), (3)], [1/2])
Traceback (most recent call last):
...
ValueError: quaternion algebra over the rationals must have an even number of
↳ramified places

sage: x = polygen(ZZ, 'x')
sage: K.<w> = NumberField(x^2-x-1)
sage: P = K.prime_above(2)
sage: Q = K.prime_above(3)
sage: A = QuaternionAlgebra(K, [P,Q], [0,0])
sage: A.discriminant()
Fractional ideal (6)
sage: A = QuaternionAlgebra(K, [P,Q], [1/2,0])
Traceback (most recent call last):
...
ValueError: quaternion algebra over a number field must have an even number of
↳ramified places
```

```
>>> from sage.all import *
>>> QuaternionAlgebra(QQ, [(Integer(2)), (Integer(3))], [Integer(0)])
Quaternion Algebra (-1, 3) with base ring Rational Field
>>> QuaternionAlgebra(QQ, [(Integer(2)), (Integer(3))], [Integer(1)/Integer(2)])
Traceback (most recent call last):
...
ValueError: quaternion algebra over the rationals must have an even number of
↳ramified places
```

(continues on next page)

(continued from previous page)

```

>>> x = polygen(ZZ, 'x')
>>> K = NumberField(x**Integer(2)-x-Integer(1), names=('w',)); (w,) = K._first_
    ↪ngens(1)
>>> P = K.prime_above(Integer(2))
>>> Q = K.prime_above(Integer(3))
>>> A = QuaternionAlgebra(K, [P,Q], [Integer(0),Integer(0)])
>>> A.discriminant()
Fractional ideal (6)
>>> A = QuaternionAlgebra(K, [P,Q], [Integer(1)/Integer(2),Integer(0)])
Traceback (most recent call last):
...
ValueError: quaternion algebra over a number field must have an even number of_
    ↪ramified places

```

The construction via ramification fails if the base field is not a number field:

```

sage: QuaternionAlgebra(RR, [], [])
Traceback (most recent call last):
...
ValueError: quaternion algebra construction via ramification only works over a_
    ↪number field

```

```

>>> from sage.all import *
>>> QuaternionAlgebra(RR, [], [])
Traceback (most recent call last):
...
ValueError: quaternion algebra construction via ramification only works over a_
    ↪number field

```

The list of local invariants must specify the ramification data at all real places of the number field K :

```

sage: QuaternionAlgebra(QuadraticField(5), [], [0])
Traceback (most recent call last):
...
ValueError: must specify ramification at all real places of the number field

```

```

>>> from sage.all import *
>>> QuaternionAlgebra(QuadraticField(Integer(5)), [], [Integer(0)])
Traceback (most recent call last):
...
ValueError: must specify ramification at all real places of the number field

```

The list of local invariants specifying the ramification at the real places may only contain 0 and $\frac{1}{2}$:

```

sage: QuaternionAlgebra(QuadraticField(5), [], [0,1])
Traceback (most recent call last):
...
ValueError: list of local invariants specifying ramification should contain only_
    ↪0 and 1/2

```

```
>>> from sage.all import *
>>> QuaternionAlgebra(QuadraticField(Integer(5)), [], [Integer(0), Integer(1)])
Traceback (most recent call last):
...
ValueError: list of local invariants specifying ramification should contain only_
↳0 and 1/2
```

Similarly, the list of finite ramified places must consist of primes or prime ideals of the number field K :

```
sage: QuaternionAlgebra(QuadraticField(5), ['water', 2], [])
Traceback (most recent call last):
...
ValueError: quaternion algebra constructor requires a list of primes specifying_
↳the ramification
```

```
>>> from sage.all import *
>>> QuaternionAlgebra(QuadraticField(Integer(5)), ['water', Integer(2)], [])
Traceback (most recent call last):
...
ValueError: quaternion algebra constructor requires a list of primes specifying_
↳the ramification
```

If the coefficients a and b in the definition of the quaternion algebra are not integral, then a slower generic type is used for arithmetic:

```
sage: type(QuaternionAlgebra(-1, -3).0)
<... 'sage.algebras.quatalg.quaternion_algebra_element.QuaternionAlgebraElement_
↳rational_field'>
sage: type(QuaternionAlgebra(-1, -3/2).0)
<... 'sage.algebras.quatalg.quaternion_algebra_element.QuaternionAlgebraElement_
↳generic'>
```

```
>>> from sage.all import *
>>> type(QuaternionAlgebra(-Integer(1), -Integer(3)).gen(0))
<... 'sage.algebras.quatalg.quaternion_algebra_element.QuaternionAlgebraElement_
↳rational_field'>
>>> type(QuaternionAlgebra(-Integer(1), -Integer(3)/Integer(2)).gen(0))
<... 'sage.algebras.quatalg.quaternion_algebra_element.QuaternionAlgebraElement_
↳generic'>
```

Make sure caching is sane:

```
sage: A = QuaternionAlgebra(2, 3); A
Quaternion Algebra (2, 3) with base ring Rational Field
sage: B = QuaternionAlgebra(GF(5)(2), GF(5)(3)); B
Quaternion Algebra (2, 3) with base ring Finite Field of size 5
sage: A is QuaternionAlgebra(2, 3)
True
sage: B is QuaternionAlgebra(GF(5)(2), GF(5)(3))
True
sage: Q = QuaternionAlgebra(2); Q
Quaternion Algebra (-1, -1) with base ring Rational Field
sage: Q is QuaternionAlgebra(QQ, -1, -1)
```

(continues on next page)

(continued from previous page)

```

True
sage: Q is QuaternionAlgebra(-1,-1)
True
sage: Q.<ii,jj,kk> = QuaternionAlgebra(15); Q.variable_names()
('ii', 'jj', 'kk')
sage: QuaternionAlgebra(15).variable_names()
('i', 'j', 'k')

```

```

>>> from sage.all import *
>>> A = QuaternionAlgebra(Integer(2),Integer(3)); A
Quaternion Algebra (2, 3) with base ring Rational Field
>>> B = QuaternionAlgebra(GF(Integer(5))(Integer(2)),GF(Integer(5))(Integer(3)));
↪B
Quaternion Algebra (2, 3) with base ring Finite Field of size 5
>>> A is QuaternionAlgebra(Integer(2),Integer(3))
True
>>> B is QuaternionAlgebra(GF(Integer(5))(Integer(2)),GF(Integer(5))(Integer(3)))
True
>>> Q = QuaternionAlgebra(Integer(2)); Q
Quaternion Algebra (-1, -1) with base ring Rational Field
>>> Q is QuaternionAlgebra(QQ,-Integer(1),-Integer(1))
True
>>> Q is QuaternionAlgebra(-Integer(1),-Integer(1))
True
>>> Q = QuaternionAlgebra(Integer(15), names=('ii', 'jj', 'kk',)); (ii, jj, kk,)
↪= Q._first_ngens(3); Q.variable_names()
('ii', 'jj', 'kk')
>>> QuaternionAlgebra(Integer(15)).variable_names()
('i', 'j', 'k')

```

create_key (*arg0*, *arg1=None*, *arg2=None*, *names='i,j,k'*)

Create a key that uniquely determines a quaternion algebra.

create_object (*version*, *key*, ***extra_args*)

Create the object from the key (extra arguments are ignored). This is only called if the object was not found in the cache.

class sage.algebras.quatalg.quaternion_algebra.**QuaternionAlgebra_ab** (*base_ring*, *a*, *b*,
names='i,j,k')

Bases: *QuaternionAlgebra_abstract*

A quaternion algebra of the form $(a, b)_K$.

See *QuaternionAlgebra* for many more examples.

INPUT:

- *base_ring* – a commutative ring K in which 2 is invertible
- *a, b* – units of K
- *names* – string (default: 'i, j, k'); names of the generators

OUTPUT:

The quaternion algebra (a, b) over K generated by i and j subject to $i^2 = a$, $j^2 = b$, and $ji = -ij$.

EXAMPLES:

```
sage: QuaternionAlgebra(QQ, -7, -21) # indirect doctest
Quaternion Algebra (-7, -21) with base ring Rational Field
```

```
>>> from sage.all import *
>>> QuaternionAlgebra(QQ, -Integer(7), -Integer(21)) # indirect doctest
Quaternion Algebra (-7, -21) with base ring Rational Field
```

discriminant()

Return the discriminant of this quaternion algebra.

The discriminant of a quaternion algebra over a number field is the product of the finite places at which the algebra ramifies.

OUTPUT:

The discriminant of this quaternion algebra, given as

- an element of $\mathbf{Z}$ if the algebra is defined over $\mathbf{Q}$,
- an integral fractional ideal of the base number field, otherwise.

EXAMPLES:

```
sage: QuaternionAlgebra(210, -22).discriminant()
210
sage: QuaternionAlgebra(19).discriminant()
19
sage: QuaternionAlgebra(-1, -1).discriminant()
2
```

```
>>> from sage.all import *
>>> QuaternionAlgebra(Integer(210), -Integer(22)).discriminant()
210
>>> QuaternionAlgebra(Integer(19)).discriminant()
19
>>> QuaternionAlgebra(-Integer(1), -Integer(1)).discriminant()
2
```

Some examples over number fields:

```
sage: K = QuadraticField(3)
sage: L = QuadraticField(-15)
sage: QuaternionAlgebra(K, -1, -1).discriminant()
Fractional ideal (1)
sage: QuaternionAlgebra(L, -1, -1).discriminant()
Fractional ideal (2)
```

```
>>> from sage.all import *
>>> K = QuadraticField(Integer(3))
>>> L = QuadraticField(-Integer(15))
>>> QuaternionAlgebra(K, -Integer(1), -Integer(1)).discriminant()
Fractional ideal (1)
>>> QuaternionAlgebra(L, -Integer(1), -Integer(1)).discriminant()
Fractional ideal (2)
```


We can also use number field elements as invariants:

```
sage: x = polygen(ZZ, 'x')
sage: F.<a> = NumberField(x^2 - x - 1)
sage: QuaternionAlgebra(F, 2*a, F(-1)).discriminant()
Fractional ideal (2)
```

```
>>> from sage.all import *
>>> x = polygen(ZZ, 'x')
>>> F = NumberField(x**Integer(2) - x - Integer(1), names=('a',)); (a,) = F._
↳first_ngens(1)
>>> QuaternionAlgebra(F, Integer(2)*a, F(-Integer(1))).discriminant()
Fractional ideal (2)
```

The method does not make sense over an arbitrary base ring:

```
sage: QuaternionAlgebra(RR(2.), 1).discriminant()
Traceback (most recent call last):
...
ValueError: base field must be rational numbers or a number field
```

```
>>> from sage.all import *
>>> QuaternionAlgebra(RR(RealNumber('2.')), Integer(1)).discriminant()
Traceback (most recent call last):
...
ValueError: base field must be rational numbers or a number field
```

gen (*i*=0)

Return the *i*-th generator of self.

INPUT:

- *i* – integer (default: 0)

EXAMPLES:

```
sage: Q.<ii,jj,kk> = QuaternionAlgebra(QQ,-1,-2); Q
Quaternion Algebra (-1, -2) with base ring Rational Field
sage: Q.gen(0)
ii
sage: Q.gen(1)
jj
sage: Q.gen(2)
kk
sage: Q.gens()
(ii, jj, kk)
```

```
>>> from sage.all import *
>>> Q = QuaternionAlgebra(QQ,-Integer(1),-Integer(2), names=('ii', 'jj', 'kk',
↳)); (ii, jj, kk,) = Q._first_ngens(3); Q
Quaternion Algebra (-1, -2) with base ring Rational Field
>>> Q.gen(Integer(0))
ii
>>> Q.gen(Integer(1))
```

(continues on next page)

(continued from previous page)

```

jj
>>> Q.gen(Integer(2))
kk
>>> Q.gens()
(ii, jj, kk)

```

gens()Return the generators of `self`.

EXAMPLES:

```

sage: Q.<ii,jj,kk> = QuaternionAlgebra(QQ,-1,-2); Q
Quaternion Algebra (-1, -2) with base ring Rational Field
sage: Q.gens()
(ii, jj, kk)

```

```

>>> from sage.all import *
>>> Q = QuaternionAlgebra(QQ,-Integer(1),-Integer(2), names=('ii', 'jj', 'kk',
↳)); (ii, jj, kk,) = Q._first_ngens(3); Q
Quaternion Algebra (-1, -2) with base ring Rational Field
>>> Q.gens()
(ii, jj, kk)

```

ideal (*gens*, *left_order=None*, *right_order=None*, *check=True*, ***kwds*)Return the quaternion ideal with given gens over $\mathbb{Z}$.

Neither a left or right order structure need be specified.

INPUT:

- *gens* – list of elements of this quaternion order
- *check* – boolean (default: `True`)
- *left_order* – a quaternion order or `None`
- *right_order* – a quaternion order or `None`

EXAMPLES:

```

sage: R = QuaternionAlgebra(-11,-1)
sage: R.ideal([2*a for a in R.basis()])
Fractional ideal (2, 2*i, 2*j, 2*k)

```

```

>>> from sage.all import *
>>> R = QuaternionAlgebra(-Integer(11),-Integer(1))
>>> R.ideal([Integer(2)*a for a in R.basis()])
Fractional ideal (2, 2*i, 2*j, 2*k)

```

inner_product_matrix()Return the inner product matrix associated to `self`, i.e. the Gram matrix of the reduced norm as a quadratic form on `self`. The standard basis $1, i, j, k$ is orthogonal, so this matrix is just the diagonal matrix with diagonal entries $1, a, b, ab$.

EXAMPLES:

```

sage: Q.<i,j,k> = QuaternionAlgebra(-5,-19)
sage: Q.inner_product_matrix()
[ 2  0  0  0]
[ 0 10  0  0]
[ 0  0 38  0]
[ 0  0  0 190]

sage: R.<a,b> = QQ[]; Q.<i,j,k> = QuaternionAlgebra(Frac(R),a,b)
sage: Q.inner_product_matrix()
[ 2  0  0  0]
[ 0 -2*a 0  0]
[ 0  0 -2*b 0]
[ 0  0  0 2*a*b]

```

```

>>> from sage.all import *
>>> Q = QuaternionAlgebra(-Integer(5),-Integer(19), names=('i', 'j', 'k'));
↳ (i, j, k) = Q._first_ngens(3)
>>> Q.inner_product_matrix()
[ 2  0  0  0]
[ 0 10  0  0]
[ 0  0 38  0]
[ 0  0  0 190]

>>> R = QQ['a, b']; (a, b) = R._first_ngens(2); Q =
↳ QuaternionAlgebra(Frac(R),a,b, names=('i', 'j', 'k')); (i, j, k) = Q._
↳ first_ngens(3)
>>> Q.inner_product_matrix()
[ 2  0  0  0]
[ 0 -2*a 0  0]
[ 0  0 -2*b 0]
[ 0  0  0 2*a*b]

```

invariants()

Return the structural invariants a, b of this quaternion algebra: `self` is generated by i, j subject to $i^2 = a$, $j^2 = b$ and $ji = -ij$.

EXAMPLES:

```

sage: Q.<i,j,k> = QuaternionAlgebra(15)
sage: Q.invariants()
(-3, 5)
sage: i^2
-3
sage: j^2
5

```

```

>>> from sage.all import *
>>> Q = QuaternionAlgebra(Integer(15), names=('i', 'j', 'k')); (i, j, k) =
↳ Q._first_ngens(3)
>>> Q.invariants()
(-3, 5)
>>> i**Integer(2)
-3

```

(continues on next page)

(continued from previous page)

```
>>> j**Integer(2)
5
```

is_definite()

Check whether this quaternion algebra is definite.

A quaternion algebra over $\mathbf{Q}$ is definite if it ramifies at the unique real place of $\mathbf{Q}$, which happens if and only if both of its invariants are negative (see Exercise 2.4(c) in [Voi2021]).

EXAMPLES:

```
sage: QuaternionAlgebra(QQ, -5, -2).is_definite()
True
sage: QuaternionAlgebra(1).is_definite()
False
```

```
>>> from sage.all import *
>>> QuaternionAlgebra(QQ, -Integer(5), -Integer(2)).is_definite()
True
>>> QuaternionAlgebra(Integer(1)).is_definite()
False
```

The method does not make sense over an arbitrary base ring:

```
sage: QuaternionAlgebra(RR(2.), 1).is_definite()
Traceback (most recent call last):
...
ValueError: base field must be rational numbers
```

```
>>> from sage.all import *
>>> QuaternionAlgebra(RR(RealNumber('2.')), Integer(1)).is_definite()
Traceback (most recent call last):
...
ValueError: base field must be rational numbers
```

is_isomorphic(A)

Check whether this quaternion algebra is isomorphic to A .

Currently only implemented for quaternion algebras defined over a number field; based on Main Theorem 14.6.1 in [Voi2021], noting that $\mathbf{Q}$ has a unique infinite place.

INPUT:

- A – a quaternion algebra defined over a number field

EXAMPLES:

```
sage: B = QuaternionAlgebra(-46, -87)
sage: A = QuaternionAlgebra(-58, -69)
sage: A == B
False
sage: B.is_isomorphic(A)
True
```

```
>>> from sage.all import *
>>> B = QuaternionAlgebra(-Integer(46), -Integer(87))
>>> A = QuaternionAlgebra(-Integer(58), -Integer(69))
>>> A == B
False
>>> B.is_isomorphic(A)
True
```

Checking ramification at both finite and infinite places, the method correctly distinguishes isomorphism classes of quaternion algebras that the discriminant can not distinguish:

```
sage: K = QuadraticField(3)
sage: A = QuaternionAlgebra(K, -1, -1)
sage: B = QuaternionAlgebra(K, 1, -1)
sage: A.discriminant() == B.discriminant()
True
sage: B.is_isomorphic(A)
False
```

```
>>> from sage.all import *
>>> K = QuadraticField(Integer(3))
>>> A = QuaternionAlgebra(K, -Integer(1), -Integer(1))
>>> B = QuaternionAlgebra(K, Integer(1), -Integer(1))
>>> A.discriminant() == B.discriminant()
True
>>> B.is_isomorphic(A)
False
```

`is_totally_definite()`

Check whether this quaternion algebra is totally definite.

A quaternion algebra defined over a number field is totally definite if it ramifies at all Archimedean places of its base field. In particular, the base number field has to be totally real (see 14.5.8 in [Voi2021]).

EXAMPLES:

```
sage: QuaternionAlgebra(QQ, -5, -2).is_totally_definite()
True

sage: K = QuadraticField(3)
sage: QuaternionAlgebra(K, -1, -1).is_totally_definite()
True
```

```
>>> from sage.all import *
>>> QuaternionAlgebra(QQ, -Integer(5), -Integer(2)).is_totally_definite()
True

>>> K = QuadraticField(Integer(3))
>>> QuaternionAlgebra(K, -Integer(1), -Integer(1)).is_totally_definite()
True
```

We can also use number field elements as invariants:

```
sage: x = polygen(ZZ, 'x')
sage: F.<a> = NumberField(x^2 - x - 1)
sage: QuaternionAlgebra(F, 2*a, F(-1)).is_totally_definite()
False
```

```
>>> from sage.all import *
>>> x = polygen(ZZ, 'x')
>>> F = NumberField(x**Integer(2) - x - Integer(1), names=('a',)); (a,) = F._
↳first_ngens(1)
>>> QuaternionAlgebra(F, Integer(2)*a, F(-Integer(1))).is_totally_definite()
False
```

The method does not make sense over an arbitrary base ring:

```
sage: QuaternionAlgebra(RR(2.), 1).is_totally_definite()
Traceback (most recent call last):
...
ValueError: base field must be rational numbers or a number field
```

```
>>> from sage.all import *
>>> QuaternionAlgebra(RR(RealNumber('2.')), Integer(1)).is_totally_definite()
Traceback (most recent call last):
...
ValueError: base field must be rational numbers or a number field
```

maximal_order (*take_shortcuts=True, order_basis=None*)

Return a maximal order in this quaternion algebra.

If *order_basis* is specified, the resulting maximal order will contain the order of the quaternion algebra given by this basis. The algorithm used is from [Voi2012].

INPUT:

- *take_shortcuts* – boolean (default: `True`); if the discriminant is prime and the invariants of the algebra are of a nice form, use Proposition 5.2 of [Piz1980].
- *order_basis* – (default: `None`) a basis of an order of this quaternion algebra

OUTPUT: a maximal order in this quaternion algebra

EXAMPLES:

```
sage: QuaternionAlgebra(-1,-7).maximal_order()
Order of Quaternion Algebra (-1, -7) with base ring Rational Field
with basis (1/2 + 1/2*j, 1/2*i + 1/2*k, j, k)

sage: QuaternionAlgebra(-1,-1).maximal_order().basis()
(1/2 + 1/2*i + 1/2*j + 1/2*k, i, j, k)

sage: QuaternionAlgebra(-1,-11).maximal_order().basis()
(1/2 + 1/2*j, 1/2*i + 1/2*k, j, k)

sage: QuaternionAlgebra(-1,-3).maximal_order().basis()
(1/2 + 1/2*j, 1/2*i + 1/2*k, j, k)
```

(continues on next page)

(continued from previous page)

```
sage: QuaternionAlgebra(-3,-1).maximal_order().basis()
(1/2 + 1/2*i, 1/2*j - 1/2*k, i, -k)
```

```
sage: QuaternionAlgebra(-2,-5).maximal_order().basis()
(1/2 + 1/2*j + 1/2*k, 1/4*i + 1/2*j + 1/4*k, j, k)
```

```
sage: QuaternionAlgebra(-5,-2).maximal_order().basis()
(1/2 + 1/2*i - 1/2*k, 1/2*i + 1/4*j - 1/4*k, i, -k)
```

```
sage: QuaternionAlgebra(-17,-3).maximal_order().basis()
(1/2 + 1/2*j, 1/2*i + 1/2*k, -1/3*j - 1/3*k, k)
```

```
sage: QuaternionAlgebra(-3,-17).maximal_order().basis()
(1/2 + 1/2*i, 1/2*j - 1/2*k, -1/3*i + 1/3*k, -k)
```

```
sage: QuaternionAlgebra(-17*9,-3).maximal_order().basis()
(1, 1/3*i, 1/6*i + 1/2*j, 1/2 + 1/3*j + 1/18*k)
```

```
sage: QuaternionAlgebra(-2, -389).maximal_order().basis()
(1/2 + 1/2*j + 1/2*k, 1/4*i + 1/2*j + 1/4*k, j, k)
```

```
>>> from sage.all import *
>>> QuaternionAlgebra(-Integer(1),-Integer(7)).maximal_order()
Order of Quaternion Algebra (-1, -7) with base ring Rational Field
with basis (1/2 + 1/2*j, 1/2*i + 1/2*k, j, k)

>>> QuaternionAlgebra(-Integer(1),-Integer(1)).maximal_order().basis()
(1/2 + 1/2*i + 1/2*j + 1/2*k, i, j, k)

>>> QuaternionAlgebra(-Integer(1),-Integer(11)).maximal_order().basis()
(1/2 + 1/2*j, 1/2*i + 1/2*k, j, k)

>>> QuaternionAlgebra(-Integer(1),-Integer(3)).maximal_order().basis()
(1/2 + 1/2*j, 1/2*i + 1/2*k, j, k)

>>> QuaternionAlgebra(-Integer(3),-Integer(1)).maximal_order().basis()
(1/2 + 1/2*i, 1/2*j - 1/2*k, i, -k)

>>> QuaternionAlgebra(-Integer(2),-Integer(5)).maximal_order().basis()
(1/2 + 1/2*j + 1/2*k, 1/4*i + 1/2*j + 1/4*k, j, k)

>>> QuaternionAlgebra(-Integer(5),-Integer(2)).maximal_order().basis()
(1/2 + 1/2*i - 1/2*k, 1/2*i + 1/4*j - 1/4*k, i, -k)

>>> QuaternionAlgebra(-Integer(17),-Integer(3)).maximal_order().basis()
(1/2 + 1/2*j, 1/2*i + 1/2*k, -1/3*j - 1/3*k, k)

>>> QuaternionAlgebra(-Integer(3),-Integer(17)).maximal_order().basis()
(1/2 + 1/2*i, 1/2*j - 1/2*k, -1/3*i + 1/3*k, -k)

>>> QuaternionAlgebra(-Integer(17)*Integer(9),-Integer(3)).maximal_order().
↪basis()
```

(continues on next page)

(continued from previous page)

```
(1, 1/3*i, 1/6*i + 1/2*j, 1/2 + 1/3*j + 1/18*k)

>>> QuaternionAlgebra(-Integer(2), -Integer(389)).maximal_order().basis()
(1/2 + 1/2*j + 1/2*k, 1/4*i + 1/2*j + 1/4*k, j, k)
```

If you want bases containing 1, switch off `take_shortcuts`:

```
sage: QuaternionAlgebra(-3,-89).maximal_order(take_shortcuts=False)
Order of Quaternion Algebra (-3, -89) with base ring Rational Field
with basis (1, 1/2 + 1/2*i, j, 1/2 + 1/6*i + 1/2*j + 1/6*k)

sage: QuaternionAlgebra(1,1).maximal_order(take_shortcuts=False)      # Matrix_
↪ring
Order of Quaternion Algebra (1, 1) with base ring Rational Field
with basis (1, 1/2 + 1/2*i, j, 1/2*j + 1/2*k)

sage: QuaternionAlgebra(-22,210).maximal_order(take_shortcuts=False)
Order of Quaternion Algebra (-22, 210) with base ring Rational Field
with basis (1, i, 1/2*i + 1/2*j, 1/2 + 17/22*i + 1/44*k)

sage: for d in ( m for m in range(1, 750) if is_squarefree(m) ):      #_
↪long time (3s)
....:     A = QuaternionAlgebra(d)
....:     assert A.maximal_order(take_shortcuts=False).is_maximal()
```

```
>>> from sage.all import *
>>> QuaternionAlgebra(-Integer(3),-Integer(89)).maximal_order(take_
↪shortcuts=False)
Order of Quaternion Algebra (-3, -89) with base ring Rational Field
with basis (1, 1/2 + 1/2*i, j, 1/2 + 1/6*i + 1/2*j + 1/6*k)

>>> QuaternionAlgebra(Integer(1),Integer(1)).maximal_order(take_
↪shortcuts=False)      # Matrix ring
Order of Quaternion Algebra (1, 1) with base ring Rational Field
with basis (1, 1/2 + 1/2*i, j, 1/2*j + 1/2*k)

>>> QuaternionAlgebra(-Integer(22),Integer(210)).maximal_order(take_
↪shortcuts=False)
Order of Quaternion Algebra (-22, 210) with base ring Rational Field
with basis (1, i, 1/2*i + 1/2*j, 1/2 + 17/22*i + 1/44*k)

>>> for d in ( m for m in range(Integer(1), Integer(750)) if is_squarefree(m) )_
↪):      # long time (3s)
...     A = QuaternionAlgebra(d)
...     assert A.maximal_order(take_shortcuts=False).is_maximal()
```

Specifying an order basis gives an extension of orders:

```
sage: A.<i,j,k> = QuaternionAlgebra(-292, -732)
sage: alpha = A.random_element()
sage: while alpha.is_zero():
....:     alpha = A.random_element()
```

(continues on next page)

(continued from previous page)

```

sage: conj = [alpha * b * alpha.inverse() for b in [k,i,j]]
sage: order_basis = tuple(conj) + (A.one(),)
sage: O = A.quaternion_order(basis=order_basis)
sage: R = A.maximal_order(order_basis=order_basis)
sage: O <= R and R.is_maximal()
True

```

```

>>> from sage.all import *
>>> A = QuaternionAlgebra(-Integer(292), -Integer(732), names=('i', 'j', 'k',
↳ )); (i, j, k,) = A._first_ngens(3)
>>> alpha = A.random_element()
>>> while alpha.is_zero():
...     alpha = A.random_element()
>>> conj = [alpha * b * alpha.inverse() for b in [k,i,j]]
>>> order_basis = tuple(conj) + (A.one(),)
>>> O = A.quaternion_order(basis=order_basis)
>>> R = A.maximal_order(order_basis=order_basis)
>>> O <= R and R.is_maximal()
True

```

We do not support number fields other than the rationals yet:

```

sage: K = QuadraticField(5)
sage: QuaternionAlgebra(K,-1,-1).maximal_order()
Traceback (most recent call last):
...
NotImplementedError: maximal order only implemented
for rational quaternion algebras

```

```

>>> from sage.all import *
>>> K = QuadraticField(Integer(5))
>>> QuaternionAlgebra(K,-Integer(1),-Integer(1)).maximal_order()
Traceback (most recent call last):
...
NotImplementedError: maximal order only implemented
for rational quaternion algebras

```

`modp_splitting_data(p)`

Return mod p splitting data for this quaternion algebra at the unramified prime p .

This is 2×2 matrices I, J, K over the finite field $\mathbf{F}_p$ such that if the quaternion algebra has generators i, j, k , then $I^2 = i^2, J^2 = j^2, IJ = K$ and $IJ = -JI$.

Note

Currently only implemented when p is odd and the base ring is $\mathbf{Q}$.

INPUT:

- p – unramified odd prime

OUTPUT: 2-tuple of matrices over finite field

EXAMPLES:

```

sage: Q = QuaternionAlgebra(-15, -19)
sage: Q.modp_splitting_data(7)
(
[0 6]  [6 1]  [6 6]
[1 0], [1 1], [6 1]
)
sage: Q.modp_splitting_data(next_prime(10^5))
(
[ 0 99988]  [97311 4]  [99999 59623]
[ 1 0], [13334 2692], [97311 4]
)
sage: I, J, K = Q.modp_splitting_data(23)
sage: I
[0 8]
[1 0]
sage: I^2
[8 0]
[0 8]
sage: J
[19 2]
[17 4]
sage: J^2
[4 0]
[0 4]
sage: I*J == -J*I
True
sage: I*J == K
True

```

```

>>> from sage.all import *
>>> Q = QuaternionAlgebra(-Integer(15), -Integer(19))
>>> Q.modp_splitting_data(Integer(7))
(
[0 6]  [6 1]  [6 6]
[1 0], [1 1], [6 1]
)
>>> Q.modp_splitting_data(next_prime(Integer(10)**Integer(5)))
(
[ 0 99988]  [97311 4]  [99999 59623]
[ 1 0], [13334 2692], [97311 4]
)
>>> I, J, K = Q.modp_splitting_data(Integer(23))
>>> I
[0 8]
[1 0]
>>> I**Integer(2)
[8 0]
[0 8]
>>> J
[19 2]
[17 4]

```

(continues on next page)

(continued from previous page)

```
>>> J**Integer(2)
[4 0]
[0 4]
>>> I*j == -j*I
True
>>> I*j == K
True
```

The following is a good test because of the asserts in the code:

```
sage: v = [Q.modp_splitting_data(p) for p in primes(20,1000)]
```

```
>>> from sage.all import *
>>> v = [Q.modp_splitting_data(p) for p in primes(Integer(20),Integer(1000))]
```

Proper error handling:

```
sage: Q.modp_splitting_data(5)
Traceback (most recent call last):
...
NotImplementedError: algorithm for computing local splittings
not implemented in general (currently require the first invariant
to be coprime to p)

sage: Q.modp_splitting_data(2)
Traceback (most recent call last):
...
NotImplementedError: p must be odd
```

```
>>> from sage.all import *
>>> Q.modp_splitting_data(Integer(5))
Traceback (most recent call last):
...
NotImplementedError: algorithm for computing local splittings
not implemented in general (currently require the first invariant
to be coprime to p)

>>> Q.modp_splitting_data(Integer(2))
Traceback (most recent call last):
...
NotImplementedError: p must be odd
```

`modp_splitting_map(p)`

Return Python map from the (p -integral) quaternion algebra to the set of 2×2 matrices over $\mathbf{F}_p$.

INPUT:

- p – prime number

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(-1, -7)
sage: f = Q.modp_splitting_map(13)
```

(continues on next page)

(continued from previous page)

```

sage: a = 2+i-j+3*k; b = 7+2*i-4*j+k
sage: f(a*b)
[12  3]
[10  5]
sage: f(a)*f(b)
[12  3]
[10  5]

```

```

>>> from sage.all import *
>>> Q = QuaternionAlgebra(-Integer(1), -Integer(7), names=('i', 'j', 'k'));
↳ (i, j, k) = Q._first_ngens(3)
>>> f = Q.modp_splitting_map(Integer(13))
>>> a = Integer(2)+i-j+Integer(3)*k; b = Integer(7)+Integer(2)*i-
↳ Integer(4)*j+k
>>> f(a*b)
[12  3]
[10  5]
>>> f(a)*f(b)
[12  3]
[10  5]

```

order_with_level (*level*)

Return an order in this quaternion algebra with given level.

INPUT:

- *level* – positive integer

Currently this is only implemented when the base field is the rational numbers and the level is divisible by at most one power of a prime that ramifies in this quaternion algebra.

EXAMPLES:

```

sage: A.<i,j,k> = QuaternionAlgebra(5)
sage: level = 2 * 5 * 17
sage: O = A.order_with_level(level); O
Order of Quaternion Algebra (-2, -5) with base ring Rational Field with basis
↳ (1/2 + 1/2*j + 7/2*k, 1/2*i + 19/2*k, j + 7*k, 17*k)

```

```

>>> from sage.all import *
>>> A = QuaternionAlgebra(Integer(5), names=('i', 'j', 'k')); (i, j, k) = A.
↳ _first_ngens(3)
>>> level = Integer(2) * Integer(5) * Integer(17)
>>> O = A.order_with_level(level); O
Order of Quaternion Algebra (-2, -5) with base ring Rational Field with basis
↳ (1/2 + 1/2*j + 7/2*k, 1/2*i + 19/2*k, j + 7*k, 17*k)

```

Check that the order has the right index in the maximal order:

```

sage: L = O.free_module()
sage: N = A.maximal_order().free_module()
sage: L.index_in(N) == level / 5
True

```

```
>>> from sage.all import *
>>> L = O.free_module()
>>> N = A.maximal_order().free_module()
>>> L.index_in(N) == level / Integer(5)
True
```

quaternion_order (*basis*, *check=True*)

Return the order of this quaternion order with given basis.

INPUT:

- *basis* – list of 4 elements of self
- *check* – boolean (default: True)

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(-11,-1)
sage: Q.quaternion_order([1,i,j,k])
Order of Quaternion Algebra (-11, -1) with base ring Rational Field
with basis (1, i, j, k)
```

```
>>> from sage.all import *
>>> Q = QuaternionAlgebra(-Integer(11),-Integer(1), names=('i', 'j', 'k'));
↪(i, j, k) = Q._first_ngens(3)
>>> Q.quaternion_order([Integer(1),i,j,k])
Order of Quaternion Algebra (-11, -1) with base ring Rational Field
with basis (1, i, j, k)
```

We test out *check=False*:

```
sage: Q.quaternion_order([1,i,j,k], check=False)
Order of Quaternion Algebra (-11, -1) with base ring Rational Field
with basis (1, i, j, k)
sage: Q.quaternion_order([i,j,k], check=False)
Order of Quaternion Algebra (-11, -1) with base ring Rational Field
with basis (i, j, k)
```

```
>>> from sage.all import *
>>> Q.quaternion_order([Integer(1),i,j,k], check=False)
Order of Quaternion Algebra (-11, -1) with base ring Rational Field
with basis (1, i, j, k)
>>> Q.quaternion_order([i,j,k], check=False)
Order of Quaternion Algebra (-11, -1) with base ring Rational Field
with basis (i, j, k)
```

ramified_places (*inf=True*)

Return the places of the base number field at which this quaternion algebra ramifies.

INPUT:

- *inf* – bool (default: True)

OUTPUT:

The non-Archimedean (AKA finite) places at which this quaternion algebra ramifies, given as

- elements of $\mathbf{Z}$ (sorted small to large) if the base field is $\mathbf{Q}$,
- integral fractional ideals of the base number field, otherwise.

Additionally, if `inf` is set to `True`, then the Archimedean (AKA infinite) places at which the quaternion algebra ramifies are also returned, given as

- the embeddings of $\mathbf{Q}$ into $\mathbf{R}$ if the base field is $\mathbf{Q}$, or
- the embeddings of the base number field into the Algebraic Real Field.

Note

Any Archimedean place at which a quaternion algebra ramifies has to be real (see 14.5.8 in [Voi2021]).

EXAMPLES:

```
sage: QuaternionAlgebra(210, -22).ramified_places()
([2, 3, 5, 7], [])
```

```
>>> from sage.all import *
>>> QuaternionAlgebra(Integer(210), -Integer(22)).ramified_places()
([2, 3, 5, 7], [])
```

For a definite quaternion algebra we get ramification at the unique infinite place of $\mathbf{Q}$:

```
sage: QuaternionAlgebra(-1, -1).ramified_places()
([2],
 [Ring morphism:
   From: Rational Field
   To:   Real Field with 53 bits of precision
   Defn: 1 |--> 1.000000000000000])
```

```
>>> from sage.all import *
>>> QuaternionAlgebra(-Integer(1), -Integer(1)).ramified_places()
([2],
 [Ring morphism:
   From: Rational Field
   To:   Real Field with 53 bits of precision
   Defn: 1 |--> 1.000000000000000])
```

Extending the base field can resolve all ramification:

```
sage: F = QuadraticField(-1)
sage: QuaternionAlgebra(F, -1, -1).ramified_places()
([], [])
```

```
>>> from sage.all import *
>>> F = QuadraticField(-Integer(1))
>>> QuaternionAlgebra(F, -Integer(1), -Integer(1)).ramified_places()
([], [])
```

Extending the base field can also resolve all ramification at finite places while still leaving some ramification at infinite places:

```

sage: K = QuadraticField(3)
sage: QuaternionAlgebra(K, -1, -1).ramified_places()
([],
 [Ring morphism:
   From: Number Field in a with defining polynomial x^2 - 3 with a = 1.
   ↪732050807568878?
   To:   Algebraic Real Field
   Defn: a |--> -1.732050807568878?,
 Ring morphism:
   From: Number Field in a with defining polynomial x^2 - 3 with a = 1.
   ↪732050807568878?
   To:   Algebraic Real Field
   Defn: a |--> 1.732050807568878?]])

```

```

>>> from sage.all import *
>>> K = QuadraticField(Integer(3))
>>> QuaternionAlgebra(K, -Integer(1), -Integer(1)).ramified_places()
([],
 [Ring morphism:
   From: Number Field in a with defining polynomial x^2 - 3 with a = 1.
   ↪732050807568878?
   To:   Algebraic Real Field
   Defn: a |--> -1.732050807568878?,
 Ring morphism:
   From: Number Field in a with defining polynomial x^2 - 3 with a = 1.
   ↪732050807568878?
   To:   Algebraic Real Field
   Defn: a |--> 1.732050807568878?]])

```

Extending the base field can also get rid of ramification at infinite places while still leaving some ramification at finite places:

```

sage: L = QuadraticField(-15)
sage: QuaternionAlgebra(L, -1, -1).ramified_places()
([Fractional ideal (2, 1/2*a + 1/2), Fractional ideal (2, 1/2*a - 1/2)], [])

```

```

>>> from sage.all import *
>>> L = QuadraticField(-Integer(15))
>>> QuaternionAlgebra(L, -Integer(1), -Integer(1)).ramified_places()
([Fractional ideal (2, 1/2*a + 1/2), Fractional ideal (2, 1/2*a - 1/2)], [])

```

We can use number field elements as invariants as well:

```

sage: x = polygen(ZZ, 'x')
sage: F.<a> = NumberField(x^2 - x - 1)
sage: QuaternionAlgebra(F, 2*a, F(-1)).ramified_places()
([Fractional ideal (2)],
 [Ring morphism:
   From: Number Field in a with defining polynomial x^2 - x - 1
   To: Algebraic Real Field
   Defn: a |--> -0.618033988749895?]])

```

```
>>> from sage.all import *
>>> x = polygen(ZZ, 'x')
>>> F = NumberField(x**Integer(2) - x - Integer(1), names=('a',)); (a,) = F._
↳first_ngens(1)
>>> QuaternionAlgebra(F, Integer(2)*a, F(-Integer(1))).ramified_places()
([Fractional ideal (2)],
 [Ring morphism:
  From: Number Field in a with defining polynomial x^2 - x - 1
  To: Algebraic Real Field
  Defn: a |--> -0.618033988749895?])
```

The method does not make sense over an arbitrary base ring:

```
sage: QuaternionAlgebra(RR(2.), 1).ramified_places()
Traceback (most recent call last):
...
ValueError: base field must be rational numbers or a number field
```

```
>>> from sage.all import *
>>> QuaternionAlgebra(RR(RealNumber('2.')), Integer(1)).ramified_places()
Traceback (most recent call last):
...
ValueError: base field must be rational numbers or a number field
```

ramified_primes()

Return the (finite) primes of the base number field at which this quaternion algebra ramifies.

OUTPUT:

The list of finite primes at which this quaternion algebra ramifies, given as

- elements of $\mathbf{Z}$ (sorted small to large) if the base field is $\mathbf{Q}$,
- integral fractional ideals of the base number field, otherwise.

EXAMPLES:

```
sage: QuaternionAlgebra(-58, -69).ramified_primes()
[3, 23, 29]
```

```
>>> from sage.all import *
>>> QuaternionAlgebra(-Integer(58), -Integer(69)).ramified_primes()
[3, 23, 29]
```

Under field extensions, the number of ramified primes can increase or decrease:

```
sage: K = QuadraticField(3)
sage: L = QuadraticField(-15)
sage: QuaternionAlgebra(-1, -1).ramified_primes()
[2]
sage: QuaternionAlgebra(K, -1, -1).ramified_primes()
[]
sage: QuaternionAlgebra(L, -1, -1).ramified_primes()
[Fractional ideal (2, 1/2*a + 1/2), Fractional ideal (2, 1/2*a - 1/2)]
```



```

>>> from sage.all import *
>>> K = QuadraticField(Integer(3))
>>> L = QuadraticField(-Integer(15))
>>> QuaternionAlgebra(-Integer(1), -Integer(1)).ramified_primes()
[2]
>>> QuaternionAlgebra(K, -Integer(1), -Integer(1)).ramified_primes()
[]
>>> QuaternionAlgebra(L, -Integer(1), -Integer(1)).ramified_primes()
[Fractional ideal (2, 1/2*a + 1/2), Fractional ideal (2, 1/2*a - 1/2)]

```

We can also use number field elements as invariants:

```

sage: x = polygen(ZZ, 'x')
sage: F.<a> = NumberField(x^2 - x - 1)
sage: QuaternionAlgebra(F, 2*a, F(-1)).ramified_primes()
[Fractional ideal (2)]

```

```

>>> from sage.all import *
>>> x = polygen(ZZ, 'x')
>>> F = NumberField(x**Integer(2) - x - Integer(1), names=('a',)); (a,) = F._
↳first_ngens(1)
>>> QuaternionAlgebra(F, Integer(2)*a, F(-Integer(1))).ramified_primes()
[Fractional ideal (2)]

```

The method does not make sense over an arbitrary base ring:

```

sage: QuaternionAlgebra(RR(2.), 1).ramified_primes()
Traceback (most recent call last):
...
ValueError: base field must be rational numbers or a number field

```

```

>>> from sage.all import *
>>> QuaternionAlgebra(RR(RealNumber('2.')), Integer(1)).ramified_primes()
Traceback (most recent call last):
...
ValueError: base field must be rational numbers or a number field

```

class sage.algebras.quatalg.quaternion_algebra.**QuaternionAlgebra_abstract**

Bases: **Parent**

basis()

Return the fixed basis of self, which is $1, i, j, k$, where i, j, k are the generators of self.

EXAMPLES:

```

sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -5, -2)
sage: Q.basis()
(1, i, j, k)

sage: Q.<xyz,abc,theta> = QuaternionAlgebra(GF(9, 'a'), -5, -2)
sage: Q.basis()
(1, xyz, abc, theta)

```

```

>>> from sage.all import *
>>> Q = QuaternionAlgebra(QQ, -Integer(5), -Integer(2), names=('i', 'j', 'k',));
↪ (i, j, k,) = Q._first_ngens(3)
>>> Q.basis()
(1, i, j, k)

>>> Q = QuaternionAlgebra(GF(Integer(9), 'a'), -Integer(5), -Integer(2), names=(
↪ 'xyz', 'abc', 'theta',)); (xyz, abc, theta,) = Q._first_ngens(3)
>>> Q.basis()
(1, xyz, abc, theta)

```

The basis is cached:

```

sage: Q.basis() is Q.basis()
True

```

```

>>> from sage.all import *
>>> Q.basis() is Q.basis()
True

```

`free_module()`

Return the free module associated to `self` with inner product given by the reduced norm.

EXAMPLES:

```

sage: A.<t> = LaurentPolynomialRing(GF(3))
sage: B = QuaternionAlgebra(A, -1, t)
sage: B.free_module()
Ambient free quadratic module of rank 4 over the principal ideal domain
Univariate Laurent Polynomial Ring in t over Finite Field of size 3
Inner product matrix:
[2 0 0 0]
[0 2 0 0]
[0 0 t 0]
[0 0 0 t]

```

```

>>> from sage.all import *
>>> A = LaurentPolynomialRing(GF(Integer(3)), names=('t',)); (t,) = A._first_
↪ ngens(1)
>>> B = QuaternionAlgebra(A, -Integer(1), t)
>>> B.free_module()
Ambient free quadratic module of rank 4 over the principal ideal domain
Univariate Laurent Polynomial Ring in t over Finite Field of size 3
Inner product matrix:
[2 0 0 0]
[0 2 0 0]
[0 0 t 0]
[0 0 0 t]

```

`inner_product_matrix()`

Return the inner product matrix associated to `self`.

This is the Gram matrix of the reduced norm as a quadratic form on `self`. The standard basis $1, i, j, k$ is orthogonal, so this matrix is just the diagonal matrix with diagonal entries $2, 2a, 2b, 2ab$.

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(-5,-19)
sage: Q.inner_product_matrix()
[ 2  0  0  0]
[ 0 10  0  0]
[ 0  0 38  0]
[ 0  0  0 190]
```

```
>>> from sage.all import *
>>> Q = QuaternionAlgebra(-Integer(5),-Integer(19), names=('i', 'j', 'k'));
↪(i, j, k) = Q._first_ngens(3)
>>> Q.inner_product_matrix()
[ 2  0  0  0]
[ 0 10  0  0]
[ 0  0 38  0]
[ 0  0  0 190]
```

is_commutative()

Return False always, since all quaternion algebras are noncommutative.

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -3,-7)
sage: Q.is_commutative()
False
```

```
>>> from sage.all import *
>>> Q = QuaternionAlgebra(QQ, -Integer(3),-Integer(7), names=('i', 'j', 'k',
↪)); (i, j, k) = Q._first_ngens(3)
>>> Q.is_commutative()
False
```

is_division_algebra()

Check whether this quaternion algebra is a division algebra, i.e. whether every nonzero element in it is invertible.

Currently only implemented for quaternion algebras defined over a number field.

EXAMPLES:

```
sage: QuaternionAlgebra(QQ,-5,-2).is_division_algebra()
True
sage: QuaternionAlgebra(2,9).is_division_algebra()
False
sage: K.<z> = QuadraticField(3)
sage: QuaternionAlgebra(K, 1+z, 3-z).is_division_algebra()
False
```

```
>>> from sage.all import *
>>> QuaternionAlgebra(QQ,-Integer(5),-Integer(2)).is_division_algebra()
True
>>> QuaternionAlgebra(Integer(2),Integer(9)).is_division_algebra()
False
```

(continues on next page)

(continued from previous page)

```
>>> K = QuadraticField(Integer(3), names=('z',)); (z,) = K._first_ngens(1)
>>> QuaternionAlgebra(K, Integer(1)+z, Integer(3)-z).is_division_algebra()
False
```

By checking ramification, the method correctly recognizes division quaternion algebras over a number field even if they have trivial discriminant:

```
sage: L = QuadraticField(5)
sage: A = QuaternionAlgebra(L, -1, -1)
sage: A.discriminant()
Fractional ideal (1)
sage: A.is_division_algebra()
True
```

```
>>> from sage.all import *
>>> L = QuadraticField(Integer(5))
>>> A = QuaternionAlgebra(L, -Integer(1), -Integer(1))
>>> A.discriminant()
Fractional ideal (1)
>>> A.is_division_algebra()
True
```

The method is not implemented over arbitrary base rings yet:

```
sage: QuaternionAlgebra(RR(2.), 1).is_division_algebra()
Traceback (most recent call last):
...
NotImplementedError: base ring must be rational numbers or a number field
```

```
>>> from sage.all import *
>>> QuaternionAlgebra(RR(RealNumber('2.')), Integer(1)).is_division_algebra()
Traceback (most recent call last):
...
NotImplementedError: base ring must be rational numbers or a number field
```

`is_exact()`

Return True if elements of this quaternion algebra are represented exactly, i.e. there is no precision loss when doing arithmetic. A quaternion algebra is exact if and only if its base field is exact.

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -3, -7)
sage: Q.is_exact()
True
sage: Q.<i,j,k> = QuaternionAlgebra(Qp(7), -3, -7)
sage: Q.is_exact()
False
```

```
>>> from sage.all import *
>>> Q = QuaternionAlgebra(QQ, -Integer(3), -Integer(7), names=('i', 'j', 'k',
↳ )); (i, j, k,) = Q._first_ngens(3)
>>> Q.is_exact()
```

(continues on next page)

(continued from previous page)

```

True
>>> Q = QuaternionAlgebra(Qp(Integer(7)), -Integer(3), -Integer(7), names=('i
↳', 'j', 'k')); (i, j, k) = Q._first_ngens(3)
>>> Q.is_exact()
False

```

is_field(*proof=True*)

Return False always, since all quaternion algebras are noncommutative and all fields are commutative.

EXAMPLES:

```

sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -3, -7)
sage: Q.is_field()
False

```

```

>>> from sage.all import *
>>> Q = QuaternionAlgebra(QQ, -Integer(3), -Integer(7), names=('i', 'j', 'k',
↳)); (i, j, k) = Q._first_ngens(3)
>>> Q.is_field()
False

```

is_finite()

Return True if the quaternion algebra is finite as a set.

Algorithm: A quaternion algebra is finite if and only if the base field is finite.

EXAMPLES:

```

sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -3, -7)
sage: Q.is_finite()
False
sage: Q.<i,j,k> = QuaternionAlgebra(GF(5), -3, -7)
sage: Q.is_finite()
True

```

```

>>> from sage.all import *
>>> Q = QuaternionAlgebra(QQ, -Integer(3), -Integer(7), names=('i', 'j', 'k',
↳)); (i, j, k) = Q._first_ngens(3)
>>> Q.is_finite()
False
>>> Q = QuaternionAlgebra(GF(Integer(5)), -Integer(3), -Integer(7), names=('i
↳', 'j', 'k')); (i, j, k) = Q._first_ngens(3)
>>> Q.is_finite()
True

```

is_integral_domain(*proof=True*)

Return False always, since all quaternion algebras are noncommutative and integral domains are commutative (in Sage).

EXAMPLES:

```

sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -3, -7)
sage: Q.is_integral_domain()
False

```

```
>>> from sage.all import *
>>> Q = QuaternionAlgebra(QQ, -Integer(3), -Integer(7), names=('i', 'j', 'k',
↳)); (i, j, k) = Q._first_ngens(3)
>>> Q.is_integral_domain()
False
```

is_matrix_ring()

Check whether this quaternion algebra is isomorphic to the 2x2 matrix ring over the base ring.

Currently only implemented for quaternion algebras defined over a number field.

EXAMPLES:

```
sage: QuaternionAlgebra(QQ, -5, -2).is_matrix_ring()
False
sage: QuaternionAlgebra(2, 9).is_matrix_ring()
True
sage: K.<z> = QuadraticField(3)
sage: QuaternionAlgebra(K, 1+z, 3-z).is_matrix_ring()
True
```

```
>>> from sage.all import *
>>> QuaternionAlgebra(QQ, -Integer(5), -Integer(2)).is_matrix_ring()
False
>>> QuaternionAlgebra(Integer(2), Integer(9)).is_matrix_ring()
True
>>> K = QuadraticField(Integer(3), names=('z',)); (z,) = K._first_ngens(1)
>>> QuaternionAlgebra(K, Integer(1)+z, Integer(3)-z).is_matrix_ring()
True
```

By checking ramification, the method is able to recognize that quaternion algebras (defined over a number field) with trivial discriminant need not be matrix rings:

```
sage: L = QuadraticField(5)
sage: A = QuaternionAlgebra(L, -1, -1)
sage: A.discriminant()
Fractional ideal (1)
sage: A.is_matrix_ring()
False
```

```
>>> from sage.all import *
>>> L = QuadraticField(Integer(5))
>>> A = QuaternionAlgebra(L, -Integer(1), -Integer(1))
>>> A.discriminant()
Fractional ideal (1)
>>> A.is_matrix_ring()
False
```

The method is not implemented over arbitrary base rings yet:

```
sage: QuaternionAlgebra(RR(2.), 1).is_matrix_ring()
Traceback (most recent call last):
...
NotImplementedError: base ring must be rational numbers or a number field
```

```
>>> from sage.all import *
>>> QuaternionAlgebra(RR(RealNumber('2.')), Integer(1)).is_matrix_ring()
Traceback (most recent call last):
...
NotImplementedError: base ring must be rational numbers or a number field
```

is_noetherian()

Return True always, since any quaternion algebra is a Noetherian ring (because it is a finitely generated module over a field).

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -3, -7)
sage: Q.is_noetherian()
True
```

```
>>> from sage.all import *
>>> Q = QuaternionAlgebra(QQ, -Integer(3), -Integer(7), names=('i', 'j', 'k',
↪ )); (i, j, k,) = Q._first_ngens(3)
>>> Q.is_noetherian()
True
```

ngens()

Return the number of generators of the quaternion algebra as a K-vector space, not including 1.

This value is always 3: the algebra is spanned by the standard basis $1, i, j, k$.

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -5, -2)
sage: Q.ngens()
3
sage: Q.gens()
(i, j, k)
```

```
>>> from sage.all import *
>>> Q = QuaternionAlgebra(QQ, -Integer(5), -Integer(2), names=('i', 'j', 'k',));
↪ (i, j, k,) = Q._first_ngens(3)
>>> Q.ngens()
3
>>> Q.gens()
(i, j, k)
```

order()

Return the number of elements of the quaternion algebra, or +Infinity if the algebra is not finite.

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -3, -7)
sage: Q.order()
+Infinity
sage: Q.<i,j,k> = QuaternionAlgebra(GF(5), -3, -7)
sage: Q.order()
625
```

```

>>> from sage.all import *
>>> Q = QuaternionAlgebra(QQ, -Integer(3), -Integer(7), names=('i', 'j', 'k',
↳)); (i, j, k,) = Q._first_ngens(3)
>>> Q.order()
+Infinity
>>> Q = QuaternionAlgebra(GF(Integer(5)), -Integer(3), -Integer(7), names=('i
↳', 'j', 'k',)); (i, j, k,) = Q._first_ngens(3)
>>> Q.order()
625

```

random_element (*args, **kwargs)

Return a random element of this quaternion algebra.

The args and kwargs are passed to the random_element method of the base ring.

EXAMPLES:

```

sage: g = QuaternionAlgebra(QQ[sqrt(2)], -3, 7).random_element() #_
↳needs sage.symbolic
sage: g.parent() is QuaternionAlgebra(QQ[sqrt(2)], -3, 7) #_
↳needs sage.symbolic
True
sage: g = QuaternionAlgebra(-3, 19).random_element()
sage: g.parent() is QuaternionAlgebra(-3, 19)
True
sage: g = QuaternionAlgebra(GF(17)(2), 3).random_element()
sage: g.parent() is QuaternionAlgebra(GF(17)(2), 3)
True

```

```

>>> from sage.all import *
>>> g = QuaternionAlgebra(QQ[sqrt(Integer(2))], -Integer(3), Integer(7)).
↳random_element() # needs sage.symbolic
>>> g.parent() is QuaternionAlgebra(QQ[sqrt(Integer(2))], -Integer(3),
↳Integer(7)) # needs sage.symbolic
True
>>> g = QuaternionAlgebra(-Integer(3), Integer(19)).random_element()
>>> g.parent() is QuaternionAlgebra(-Integer(3), Integer(19))
True
>>> g = QuaternionAlgebra(GF(Integer(17))(Integer(2)), Integer(3)).random_
↳element()
>>> g.parent() is QuaternionAlgebra(GF(Integer(17))(Integer(2)), Integer(3))
True

```

Specify the numerator and denominator bounds:

```

sage: g = QuaternionAlgebra(-3, 19).random_element(10^6, 10^6)
sage: for h in g:
.....:     assert h.numerator() in range(-10^6, 10^6 + 1)
.....:     assert h.denominator() in range(10^6 + 1)

sage: g = QuaternionAlgebra(-3, 19).random_element(5, 4)
sage: for h in g:
.....:     assert h.numerator() in range(-5, 5 + 1)
.....:     assert h.denominator() in range(4 + 1)

```



```

>>> from sage.all import *
>>> g = QuaternionAlgebra(-Integer(3), Integer(19)).random_
↳element(Integer(10)**Integer(6), Integer(10)**Integer(6))
>>> for h in g:
...     assert h.numerator() in range(-Integer(10)**Integer(6), ↳
↳Integer(10)**Integer(6) + Integer(1))
...     assert h.denominator() in range(Integer(10)**Integer(6) + Integer(1))

>>> g = QuaternionAlgebra(-Integer(3), Integer(19)).random_element(Integer(5), ↳
↳Integer(4))
>>> for h in g:
...     assert h.numerator() in range(-Integer(5), Integer(5) + Integer(1))
...     assert h.denominator() in range(Integer(4) + Integer(1))

```

vector_space()

Alias for `free_module()`.

EXAMPLES:

```

sage: QuaternionAlgebra(-3, 19).vector_space()
Ambient quadratic space of dimension 4 over Rational Field
Inner product matrix:
[  2   0   0   0]
[  0   6   0   0]
[  0   0 -38   0]
[  0   0   0 -114]

```

```

>>> from sage.all import *
>>> QuaternionAlgebra(-Integer(3), Integer(19)).vector_space()
Ambient quadratic space of dimension 4 over Rational Field
Inner product matrix:
[  2   0   0   0]
[  0   6   0   0]
[  0   0 -38   0]
[  0   0   0 -114]

```

```

class sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal(ring, gens,
                                                                           coerce=True,
                                                                           **kwds)

```

Bases: `Ideal_fractional`

```

class sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational(Q,
                                                                           basis,
                                                                           left_or-
                                                                           der=None,
                                                                           right_or-
                                                                           der=None,
                                                                           check=True)

```

Bases: `QuaternionFractionalIdeal`

A fractional ideal in a rational quaternion algebra.

INPUT:

- `left_order` – a quaternion order or None

- `right_order` – a quaternion order or `None`
- `basis` – tuple of length 4 of elements in of ambient quaternion algebra whose $\mathbf{Z}$ -span is an ideal
- `check` – boolean (default: `True`); if `False`, do no type checking.

basis()

Return a basis for this fractional ideal.

OUTPUT: tuple

EXAMPLES:

```
sage: QuaternionAlgebra(-11,-1).maximal_order().unit_ideal().basis()
(1/2 + 1/2*i, 1/2*j - 1/2*k, i, -k)
```

```
>>> from sage.all import *
>>> QuaternionAlgebra(-Integer(11),-Integer(1)).maximal_order().unit_ideal().
↳basis()
(1/2 + 1/2*i, 1/2*j - 1/2*k, i, -k)
```

basis_matrix()

Return basis matrix M in Hermite normal form for `self` as a matrix with rational entries.

If Q is the ambient quaternion algebra, then the $\mathbf{Z}$ -span of the rows of M viewed as linear combinations of $Q.basis() = [1, i, j, k]$ is the fractional ideal `self`. Also, $M * M.denominator()$ is an integer matrix in Hermite normal form.

OUTPUT: matrix over $\mathbf{Q}$

EXAMPLES:

```
sage: QuaternionAlgebra(-11,-1).maximal_order().unit_ideal().basis_matrix()
[1/2 1/2  0  0]
[  0  1  0  0]
[  0  0 1/2 1/2]
[  0  0  0  1]
```

```
>>> from sage.all import *
>>> QuaternionAlgebra(-Integer(11),-Integer(1)).maximal_order().unit_ideal().
↳basis_matrix()
[1/2 1/2  0  0]
[  0  1  0  0]
[  0  0 1/2 1/2]
[  0  0  0  1]
```

conjugate()

Return the ideal with generators the conjugates of the generators for `self`.

OUTPUT: a quaternionic fractional ideal

EXAMPLES:

```
sage: I = BrandtModule(3,5).right_ideals()[1]; I
Fractional ideal (8, 40*i, 6 + 28*i + 2*j, 4 + 18*i + 2*k)
sage: I.conjugate()
Fractional ideal (2 + 2*j + 28*k, 2*i + 4*j + 34*k, 8*j + 32*k, 40*k)
```

```
>>> from sage.all import *
>>> I = BrandtModule(Integer(3), Integer(5)).right_ideals()[Integer(1)]; I
Fractional ideal (8, 40*i, 6 + 28*i + 2*j, 4 + 18*i + 2*k)
>>> I.conjugate()
Fractional ideal (2 + 2*j + 28*k, 2*i + 4*j + 34*k, 8*j + 32*k, 40*k)
```

cyclic_right_subideals (p , α =None)

Let $I = \text{self}$. This function returns the right subideals J of I such that I/J is an $\mathbf{F}_p$ -vector space of dimension 2.

INPUT:

- p – prime number (see below)
- α – (default: None) element of quaternion algebra, which can be used to parameterize the order of the ideals J . More precisely the J 's are the right annihilators of $(1, 0)\alpha^i$ for $i = 0, 1, 2, \dots, p$

OUTPUT: list of right ideals

Note

Currently, p must satisfy a bunch of conditions, or a `NotImplementedError` is raised. In particular, p must be odd and unramified in the quaternion algebra, must be coprime to the index of the right order in the maximal order, and also coprime to the normal of `self`. (The Brandt modules code has a more general algorithm in some cases.)

EXAMPLES:

```
sage: B = BrandtModule(2, 37); I = B.right_ideals()[0]
sage: I.cyclic_right_subideals(3)
[Fractional ideal (12, 444*i, 8 + 404*i + 4*j, 2 + 150*i + 2*j + 2*k),
 Fractional ideal (12, 444*i, 4 + 256*i + 4*j, 10 + 150*i + 2*j + 2*k),
 Fractional ideal (12, 444*i, 8 + 256*i + 4*j, 6 + 298*i + 2*j + 2*k),
 Fractional ideal (12, 444*i, 4 + 404*i + 4*j, 6 + 2*i + 2*j + 2*k)]

sage: B = BrandtModule(5, 389); I = B.right_ideals()[0]
sage: C = I.cyclic_right_subideals(3); C
[Fractional ideal (12, 4668*i, 10 + 3426*i + 2*j, 6 + 379*i + k),
 Fractional ideal (12, 4668*i, 2 + 3426*i + 2*j, 6 + 3491*i + k),
 Fractional ideal (12, 4 + 1556*i, 6 + 942*i + 6*j, 693*i + 2*j + k),
 Fractional ideal (12, 8 + 1556*i, 6 + 942*i + 6*j, 2 + 1007*i + 4*j + k)]
sage: [(I.free_module()/J.free_module()).invariants() for J in C]
[(3, 3), (3, 3), (3, 3), (3, 3)]
sage: I.scale(3).cyclic_right_subideals(3)
[Fractional ideal (36, 14004*i, 30 + 10278*i + 6*j, 18 + 1137*i + 3*k),
 Fractional ideal (36, 14004*i, 6 + 10278*i + 6*j, 18 + 10473*i + 3*k),
 Fractional ideal (36, 12 + 4668*i, 18 + 2826*i + 18*j, 2079*i + 6*j + 3*k),
 Fractional ideal (36, 24 + 4668*i, 18 + 2826*i + 18*j, 6 + 3021*i + 12*j + 3*k)]
sage: C = I.scale(1/9).cyclic_right_subideals(3); C
[Fractional ideal (4/3, 1556/3*i, 10/9 + 1142/3*i + 2/9*j, 2/3 + 379/9*i + 1/9*k),
 Fractional ideal (4/3, 1556/3*i, 2/9 + 1142/3*i + 2/9*j, 2/3 + 3491/9*i + 1/9*k),
```

(continues on next page)

(continued from previous page)

```

Fractional ideal (4/3, 4/9 + 1556/9*i, 2/3 + 314/3*i + 2/3*j, 77*i + 2/9*j +
↪1/9*k),
Fractional ideal (4/3, 8/9 + 1556/9*i, 2/3 + 314/3*i + 2/3*j, 2/9 + 1007/9*i
↪+ 4/9*j + 1/9*k)]
sage: [(I.scale(1/9).free_module()/J.free_module()).invariants() for J in C]
[(3, 3), (3, 3), (3, 3), (3, 3)]

sage: Q.<i,j,k> = QuaternionAlgebra(-2,-5)
sage: I = Q.ideal([Q(1),i,j,k])
sage: I.cyclic_right_subideals(3)
[Fractional ideal (3, 3*i, 2 + j, i + k),
Fractional ideal (3, 3*i, 1 + j, 2*i + k),
Fractional ideal (3, 2 + i, 3*j, 2*j + k),
Fractional ideal (3, 1 + i, 3*j, j + k)]

```

```

>>> from sage.all import *
>>> B = BrandtModule(Integer(2),Integer(37)); I = B.right_ideals()[Integer(0)]
>>> I.cyclic_right_subideals(Integer(3))
[Fractional ideal (12, 444*i, 8 + 404*i + 4*j, 2 + 150*i + 2*j + 2*k),
Fractional ideal (12, 444*i, 4 + 256*i + 4*j, 10 + 150*i + 2*j + 2*k),
Fractional ideal (12, 444*i, 8 + 256*i + 4*j, 6 + 298*i + 2*j + 2*k),
Fractional ideal (12, 444*i, 4 + 404*i + 4*j, 6 + 2*i + 2*j + 2*k)]

>>> B = BrandtModule(Integer(5),Integer(389)); I = B.right_
↪ideals()[Integer(0)]
>>> C = I.cyclic_right_subideals(Integer(3)); C
[Fractional ideal (12, 4668*i, 10 + 3426*i + 2*j, 6 + 379*i + k),
Fractional ideal (12, 4668*i, 2 + 3426*i + 2*j, 6 + 3491*i + k),
Fractional ideal (12, 4 + 1556*i, 6 + 942*i + 6*j, 693*i + 2*j + k),
Fractional ideal (12, 8 + 1556*i, 6 + 942*i + 6*j, 2 + 1007*i + 4*j + k)]
>>> [(I.free_module()/J.free_module()).invariants() for J in C]
[(3, 3), (3, 3), (3, 3), (3, 3)]
>>> I.scale(Integer(3)).cyclic_right_subideals(Integer(3))
[Fractional ideal (36, 14004*i, 30 + 10278*i + 6*j, 18 + 1137*i + 3*k),
Fractional ideal (36, 14004*i, 6 + 10278*i + 6*j, 18 + 10473*i + 3*k),
Fractional ideal (36, 12 + 4668*i, 18 + 2826*i + 18*j, 2079*i + 6*j + 3*k),
Fractional ideal (36, 24 + 4668*i, 18 + 2826*i + 18*j, 6 + 3021*i + 12*j +
↪3*k)]
>>> C = I.scale(Integer(1)/Integer(9)).cyclic_right_subideals(Integer(3)); C
[Fractional ideal (4/3, 1556/3*i, 10/9 + 1142/3*i + 2/9*j, 2/3 + 379/9*i + 1/
↪9*k),
Fractional ideal (4/3, 1556/3*i, 2/9 + 1142/3*i + 2/9*j, 2/3 + 3491/9*i + 1/
↪9*k),
Fractional ideal (4/3, 4/9 + 1556/9*i, 2/3 + 314/3*i + 2/3*j, 77*i + 2/9*j +
↪1/9*k),
Fractional ideal (4/3, 8/9 + 1556/9*i, 2/3 + 314/3*i + 2/3*j, 2/9 + 1007/9*i
↪+ 4/9*j + 1/9*k)]
>>> [(I.scale(Integer(1)/Integer(9)).free_module()/J.free_module()).
↪invariants() for J in C]
[(3, 3), (3, 3), (3, 3), (3, 3)]

>>> Q = QuaternionAlgebra(-Integer(2),-Integer(5), names=('i', 'j', 'k'));

```

(continues on next page)

(continued from previous page)

```

↪(i, j, k,) = Q._first_ngens(3)
>>> I = Q.ideal([Q(Integer(1)),i,j,k])
>>> I.cyclic_right_subideals(Integer(3))
[Fractional ideal (3, 3*i, 2 + j, i + k),
 Fractional ideal (3, 3*i, 1 + j, 2*i + k),
 Fractional ideal (3, 2 + i, 3*j, 2*j + k),
 Fractional ideal (3, 1 + i, 3*j, j + k)]

```

The general algorithm is not yet implemented here:

```

sage: I.cyclic_right_subideals(3)[0].cyclic_right_subideals(3)
Traceback (most recent call last):
...
NotImplementedError: general algorithm not implemented
(The given basis vectors must be linearly independent.)

```

```

>>> from sage.all import *
>>> I.cyclic_right_subideals(Integer(3))[Integer(0)].cyclic_right_
↪subideals(Integer(3))
Traceback (most recent call last):
...
NotImplementedError: general algorithm not implemented
(The given basis vectors must be linearly independent.)

```

free_module()

Return the underlying free $\mathbf{Z}$ -module corresponding to this ideal.

OUTPUT: free $\mathbf{Z}$ -module of rank 4 embedded in an ambient $\mathbf{Q}^4$

EXAMPLES:

```

sage: X = BrandtModule(3,5).right_ideals()
sage: X[0]
Fractional ideal (4, 20*i, 2 + 8*i + 2*j, 18*i + 2*k)
sage: X[0].free_module()
Free module of degree 4 and rank 4 over Integer Ring
Echelon basis matrix:
[ 2  0  2  8]
[ 0  2  0 18]
[ 0  0  4 16]
[ 0  0  0 20]
sage: X[0].scale(1/7).free_module()
Free module of degree 4 and rank 4 over Integer Ring
Echelon basis matrix:
[ 2/7  0  2/7  8/7]
[  0  2/7  0 18/7]
[  0  0  4/7 16/7]
[  0  0  0 20/7]
sage: QuaternionAlgebra(-11,-1).maximal_order().unit_ideal().basis_matrix()
[1/2 1/2  0  0]
[  0  1  0  0]
[  0  0 1/2 1/2]

```

(continues on next page)

(continued from previous page)

```
[ 0 0 0 1]
```

```
>>> from sage.all import *
>>> X = BrandtModule(Integer(3), Integer(5)).right_ideals()
>>> X[Integer(0)]
Fractional ideal (4, 20*i, 2 + 8*i + 2*j, 18*i + 2*k)
>>> X[Integer(0)].free_module()
Free module of degree 4 and rank 4 over Integer Ring
Echelon basis matrix:
[ 2 0 2 8]
[ 0 2 0 18]
[ 0 0 4 16]
[ 0 0 0 20]
>>> X[Integer(0)].scale(Integer(1)/Integer(7)).free_module()
Free module of degree 4 and rank 4 over Integer Ring
Echelon basis matrix:
[ 2/7 0 2/7 8/7]
[ 0 2/7 0 18/7]
[ 0 0 4/7 16/7]
[ 0 0 0 20/7]

>>> QuaternionAlgebra(-Integer(11), -Integer(1)).maximal_order().unit_ideal().
↪basis_matrix()
[1/2 1/2 0 0]
[ 0 1 0 0]
[ 0 0 1/2 1/2]
[ 0 0 0 1]
```

The free module method is also useful since it allows for checking if one ideal is contained in another, computing quotients I/J , etc.:

```
sage: X = BrandtModule(3, 17).right_ideals()
sage: I = X[0].intersection(X[2]); I
Fractional ideal (2 + 2*j + 164*k, 2*i + 4*j + 46*k, 16*j + 224*k, 272*k)
sage: I.free_module().is_submodule(X[3].free_module())
False
sage: I.free_module().is_submodule(X[1].free_module())
True
sage: X[0].free_module() / I.free_module()
Finitely generated module V/W over Integer Ring with invariants (4, 4)
```

```
>>> from sage.all import *
>>> X = BrandtModule(Integer(3), Integer(17)).right_ideals()
>>> I = X[Integer(0)].intersection(X[Integer(2)]); I
Fractional ideal (2 + 2*j + 164*k, 2*i + 4*j + 46*k, 16*j + 224*k, 272*k)
>>> I.free_module().is_submodule(X[Integer(3)].free_module())
False
>>> I.free_module().is_submodule(X[Integer(1)].free_module())
True
>>> X[Integer(0)].free_module() / I.free_module()
Finitely generated module V/W over Integer Ring with invariants (4, 4)
```

This shows that the issue at [Issue #6760](#) is fixed:

```
sage: R.<i,j,k> = QuaternionAlgebra(-1, -13)
sage: I = R.ideal([2+i, 3*i, 5*j, j+k]); I
Fractional ideal (2 + i, 3*i, j + k, 5*k)
sage: I.free_module()
Free module of degree 4 and rank 4 over Integer Ring
Echelon basis matrix:
[2 1 0 0]
[0 3 0 0]
[0 0 1 1]
[0 0 0 5]
```

```
>>> from sage.all import *
>>> R = QuaternionAlgebra(-Integer(1), -Integer(13), names=('i', 'j', 'k'));
↪ (i, j, k) = R._first_ngens(3)
>>> I = R.ideal([Integer(2)+i, Integer(3)*i, Integer(5)*j, j+k]); I
Fractional ideal (2 + i, 3*i, j + k, 5*k)
>>> I.free_module()
Free module of degree 4 and rank 4 over Integer Ring
Echelon basis matrix:
[2 1 0 0]
[0 3 0 0]
[0 0 1 1]
[0 0 0 5]
```

gram_matrix()

Return the Gram matrix of this fractional ideal.

OUTPUT: 4×4 matrix over $\mathbf{Q}$

EXAMPLES:

```
sage: I = BrandtModule(3,5).right_ideals()[1]; I
Fractional ideal (8, 40*i, 6 + 28*i + 2*j, 4 + 18*i + 2*k)
sage: I.gram_matrix()
[ 256    0  192  128]
[    0 6400 4480 2880]
[ 192 4480 3328 2112]
[ 128 2880 2112 1408]
```

```
>>> from sage.all import *
>>> I = BrandtModule(Integer(3),Integer(5)).right_ideals()[Integer(1)]; I
Fractional ideal (8, 40*i, 6 + 28*i + 2*j, 4 + 18*i + 2*k)
>>> I.gram_matrix()
[ 256    0  192  128]
[    0 6400 4480 2880]
[ 192 4480 3328 2112]
[ 128 2880 2112 1408]
```

intersection(J)

Return the intersection of the ideals `self` and `J`.

EXAMPLES:

```
sage: X = BrandtModule(3,5).right_ideals()
sage: I = X[0].intersection(X[1]); I
Fractional ideal (2 + 6*j + 4*k, 2*i + 4*j + 34*k, 8*j + 32*k, 40*k)
```

```
>>> from sage.all import *
>>> X = BrandtModule(Integer(3), Integer(5)).right_ideals()
>>> I = X[Integer(0)].intersection(X[Integer(1)]); I
Fractional ideal (2 + 6*j + 4*k, 2*i + 4*j + 34*k, 8*j + 32*k, 40*k)
```

is_equivalent (*J*, *B=10*, *certificate=False*, *side=None*)

Check whether *self* and *J* are equivalent as ideals. Tests equivalence as right ideals by default. Requires the underlying rational quaternion algebra to be definite.

INPUT:

- *J* – a fractional quaternion ideal with same order as *self*
- *B* – a bound to compute and compare theta series before doing the full equivalence test
- *certificate* – if *True* returns an element α such that $\alpha J = I$ or $J\alpha = I$ for right and left ideals respectively
- *side* – if 'left' performs left equivalence test. If 'right' or *None* performs right ideal equivalence test

OUTPUT: boolean, or (boolean, α) if *certificate* is *True*

EXAMPLES:

```
sage: R = BrandtModule(3,5).right_ideals(); len(R)
2
sage: OO = R[0].left_order()
sage: S = OO.right_ideal([3*a for a in R[0].basis()])
sage: R[0].is_equivalent(S)
doctest:...: DeprecationWarning: is_equivalent is deprecated,
please use is_left_equivalent or is_right_equivalent
accordingly instead
See https://github.com/sagemath/sage/issues/37100 for details.
True
```

```
>>> from sage.all import *
>>> R = BrandtModule(Integer(3), Integer(5)).right_ideals(); len(R)
2
>>> OO = R[Integer(0)].left_order()
>>> S = OO.right_ideal([Integer(3)*a for a in R[Integer(0)].basis()])
>>> R[Integer(0)].is_equivalent(S)
doctest:...: DeprecationWarning: is_equivalent is deprecated,
please use is_left_equivalent or is_right_equivalent
accordingly instead
See https://github.com/sagemath/sage/issues/37100 for details.
True
```

is_integral ()

Check whether the quaternion fractional ideal *self* is integral.

An ideal in a quaternion algebra is integral if and only if it is contained in its left order. If the left order is already defined this method just checks this definition, otherwise it uses one of the alternative definitions

from Lemma 16.2.8 of [Voi2021].

EXAMPLES:

```
sage: R.<i,j,k> = QuaternionAlgebra(QQ, -1,-11)
sage: I = R.ideal([2 + 2*j + 140*k, 2*i + 4*j + 150*k, 8*j + 104*k, 152*k])
sage: I.is_integral()
True
sage: O = I.left_order()
sage: I.is_integral()
True
sage: I = R.ideal([1/2 + 2*j + 140*k, 2*i + 4*j + 150*k, 8*j + 104*k, 152*k])
sage: I.is_integral()
False
```

```
>>> from sage.all import *
>>> R = QuaternionAlgebra(QQ, -Integer(1), -Integer(11), names=('i', 'j', 'k',
↳ )); (i, j, k,) = R._first_ngens(3)
>>> I = R.ideal([Integer(2) + Integer(2)*j + Integer(140)*k, Integer(2)*i +
↳ Integer(4)*j + Integer(150)*k, Integer(8)*j + Integer(104)*k,
↳ Integer(152)*k])
>>> I.is_integral()
True
>>> O = I.left_order()
>>> I.is_integral()
True
>>> I = R.ideal([Integer(1)/Integer(2) + Integer(2)*j + Integer(140)*k,
↳ Integer(2)*i + Integer(4)*j + Integer(150)*k, Integer(8)*j + Integer(104)*k,
↳ Integer(152)*k])
>>> I.is_integral()
False
```

is_left_equivalent (*J*, *B=10*, *certificate=False*)

Check whether *self* and *J* are equivalent as left ideals. Requires the underlying rational quaternion algebra to be definite.

INPUT:

- *J* – a fractional quaternion left ideal with same order as *self*
- *B* – a bound to compute and compare theta series before doing the full equivalence test
- *certificate* – if *True* returns an element α such that $J \cdot \alpha = I$

OUTPUT: boolean, or (boolean, α) if *certificate* is *True*

is_primitive ()

Check whether the quaternion fractional ideal *self* is primitive.

An integral left $\mathcal{O}$ -ideal for some order $\mathcal{O}$ is called primitive if for all integers $n > 1$ it is not contained in $n\mathcal{O}$.

EXAMPLES:

```
sage: A.<i,j,k> = QuaternionAlgebra(QQ, -1,-11)
sage: I = A.ideal([1/2 + 1/2*i + 1/2*j + 3/2*k, i + k, j + k, 2*k])
sage: I.is_primitive()
```

(continues on next page)

(continued from previous page)

```
True
sage: (2*I).is_primitive()
False
```

```
>>> from sage.all import *
>>> A = QuaternionAlgebra(QQ, -Integer(1), -Integer(11), names=('i', 'j', 'k',
↳)); (i, j, k,) = A._first_ngens(3)
>>> I = A.ideal([Integer(1)/Integer(2) + Integer(1)/Integer(2)*i + Integer(1)/
↳ Integer(2)*j + Integer(3)/Integer(2)*k, i + k, j + k, Integer(2)*k])
>>> I.is_primitive()
True
>>> (Integer(2)*I).is_primitive()
False
```

is_principal (*certificate=False*)

Check whether `self` is principal as a full rank quaternion ideal. Requires the underlying quaternion algebra to be definite. Independent of whether `self` is a left or a right ideal.

INPUT:

- `certificate` – if `True` returns a generator α s.t. $I = \alpha O$ where O is the right order of I

OUTPUT: boolean, or (boolean, α) if `certificate` is `True`

EXAMPLES:

```
sage: B.<i,j,k> = QuaternionAlgebra(419)
sage: O = B.quaternion_order([1/2 + 3/2*j, 1/6*i + 2/3*j + 1/2*k, 3*j, k])
sage: beta = O.random_element()
sage: while beta.is_zero():
....:     beta = O.random_element()
sage: I = O*beta
sage: bool, alpha = I.is_principal(True)
sage: bool
True
sage: I == O*alpha
True
```

```
>>> from sage.all import *
>>> B = QuaternionAlgebra(Integer(419), names=('i', 'j', 'k',)); (i, j, k,) = _
↳ B._first_ngens(3)
>>> O = B.quaternion_order([Integer(1)/Integer(2) + Integer(3)/Integer(2)*j, _
↳ Integer(1)/Integer(6)*i + Integer(2)/Integer(3)*j + Integer(1)/Integer(2)*k,
↳ Integer(3)*j, k])
>>> beta = O.random_element()
>>> while beta.is_zero():
...     beta = O.random_element()
>>> I = O*beta
>>> bool, alpha = I.is_principal(True)
>>> bool
True
>>> I == O*alpha
True
```

is_right_equivalent (*J*, *B=10*, *certificate=False*)

Check whether *self* and *J* are equivalent as right ideals. Requires the underlying rational quaternion algebra to be definite.

INPUT:

- *J* – a fractional quaternion right ideal with same order as *self*
- *B* – a bound to compute and compare theta series before doing the full equivalence test
- *certificate* – if *True* returns an element *alpha* such that $\alpha \cdot J = I$

OUTPUT: boolean, or (boolean, *alpha*) if *certificate* is *True*

EXAMPLES:

```
sage: R = BrandtModule(3,5).right_ideals(); len(R)
2
sage: R[0].is_right_equivalent(R[1])
False

sage: R[0].is_right_equivalent(R[0])
True
sage: OO = R[0].left_order()
sage: S = OO.right_ideal([3*a for a in R[0].basis()])
sage: R[0].is_right_equivalent(S, certificate=True)
(True, 1/3)
sage: -1/3*S == R[0]
True

sage: B = QuaternionAlgebra(101)
sage: i,j,k = B.gens()
sage: I = B.maximal_order().unit_ideal()
sage: beta = B.random_element()
sage: while beta.is_zero():
....:     beta = B.random_element()
sage: J = beta*I
sage: bool, alpha = I.is_right_equivalent(J, certificate=True)
sage: bool
True
sage: alpha*J == I
True
```

```
>>> from sage.all import *
>>> R = BrandtModule(Integer(3), Integer(5)).right_ideals(); len(R)
2
>>> R[Integer(0)].is_right_equivalent(R[Integer(1)])
False

>>> R[Integer(0)].is_right_equivalent(R[Integer(0)])
True
>>> OO = R[Integer(0)].left_order()
>>> S = OO.right_ideal([Integer(3)*a for a in R[Integer(0)].basis()])
>>> R[Integer(0)].is_right_equivalent(S, certificate=True)
(True, 1/3)
>>> -Integer(1)/Integer(3)*S == R[Integer(0)]
```

(continues on next page)

(continued from previous page)

```

True

>>> B = QuaternionAlgebra(Integer(101))
>>> i,j,k = B.gens()
>>> I = B.maximal_order().unit_ideal()
>>> beta = B.random_element()
>>> while beta.is_zero():
...     beta = B.random_element()
>>> J = beta*I
>>> bool, alpha = I.is_right_equivalent(J, certificate=True)
>>> bool
True
>>> alpha*J == I
True

```

left_order()

Return the left order associated to this fractional ideal.

OUTPUT: an order in a quaternion algebra

EXAMPLES:

```

sage: B = BrandtModule(11)
sage: R = B.maximal_order()
sage: I = R.unit_ideal()
sage: I.left_order()
Order of Quaternion Algebra (-1, -11) with base ring Rational Field
with basis (1/2 + 1/2*j, 1/2*i + 1/2*k, j, k)

```

```

>>> from sage.all import *
>>> B = BrandtModule(Integer(11))
>>> R = B.maximal_order()
>>> I = R.unit_ideal()
>>> I.left_order()
Order of Quaternion Algebra (-1, -11) with base ring Rational Field
with basis (1/2 + 1/2*j, 1/2*i + 1/2*k, j, k)

```

We do a consistency check:

```

sage: B = BrandtModule(11,19); R = B.right_ideals()
sage: [r.left_order().discriminant() for r in R]
[209, 209, 209, 209, 209, 209, 209, 209, 209, 209, 209, 209, 209, 209, 209,
↪209, 209, 209]

```

```

>>> from sage.all import *
>>> B = BrandtModule(Integer(11),Integer(19)); R = B.right_ideals()
>>> [r.left_order().discriminant() for r in R]
[209, 209, 209, 209, 209, 209, 209, 209, 209, 209, 209, 209, 209, 209, 209,
↪209, 209, 209]

```

minimal_element()

Return an element in this quaternion ideal of minimal norm.

If the ideal is a principal lattice, this method can be used to find a generator; see [Piz1980], Corollary 1.20.

EXAMPLES:

```

sage: Quat.<i,j,k> = QuaternionAlgebra(-3,-101)
sage: O = Quat.maximal_order(); O
Order of Quaternion Algebra (-3, -101) with base ring Rational Field with
↳basis (1/2 + 1/2*i, 1/2*j - 1/2*k, -1/3*i + 1/3*k, -k)
sage: (O * 5).minimal_element()
5
sage: alpha = 1/2 + 1/6*i + j + 55/3*k
sage: I = O*141 + O*alpha; I.norm()
141
sage: el = I.minimal_element(); el
13/2 - 7/6*i + j + 2/3*k
sage: el.reduced_norm()
282

```

```

>>> from sage.all import *
>>> Quat = QuaternionAlgebra(-Integer(3),-Integer(101), names=('i', 'j', 'k',
↳)); (i, j, k,) = Quat._first_ngens(3)
>>> O = Quat.maximal_order(); O
Order of Quaternion Algebra (-3, -101) with base ring Rational Field with
↳basis (1/2 + 1/2*i, 1/2*j - 1/2*k, -1/3*i + 1/3*k, -k)
>>> (O * Integer(5)).minimal_element()
5
>>> alpha = Integer(1)/Integer(2) + Integer(1)/Integer(6)*i + j + Integer(55)/
↳Integer(3)*k
>>> I = O*Integer(141) + O*alpha; I.norm()
141
>>> el = I.minimal_element(); el
13/2 - 7/6*i + j + 2/3*k
>>> el.reduced_norm()
282

```

multiply_by_conjugate(*J*)

Return product of *self* and the conjugate *Jbar* of *J*.

INPUT:

- *J* – a quaternion ideal

OUTPUT: a quaternionic fractional ideal

EXAMPLES:

```

sage: R = BrandtModule(3,5).right_ideals()
sage: R[0].multiply_by_conjugate(R[1])
Fractional ideal (32, 160*i, 8 + 112*i + 8*j, 16 + 72*i + 8*k)
sage: R[0]*R[1].conjugate()
Fractional ideal (32, 160*i, 8 + 112*i + 8*j, 16 + 72*i + 8*k)

```

```

>>> from sage.all import *
>>> R = BrandtModule(Integer(3),Integer(5)).right_ideals()
>>> R[Integer(0)].multiply_by_conjugate(R[Integer(1)])
Fractional ideal (32, 160*i, 8 + 112*i + 8*j, 16 + 72*i + 8*k)

```

(continues on next page)

(continued from previous page)

```
>>> R[Integer(0)]*R[Integer(1)].conjugate()
Fractional ideal (32, 160*i, 8 + 112*i + 8*j, 16 + 72*i + 8*k)
```

norm()

Return the reduced norm of this fractional ideal.

OUTPUT: rational number

EXAMPLES:

```
sage: M = BrandtModule(37)
sage: C = M.right_ideals()
sage: [I.norm() for I in C]
[16, 32, 32]

sage: # optional - magma
sage: (a,b) = M.quaternion_algebra().invariants()
sage: magma.eval('A<i,j,k> := QuaternionAlgebra<Rationals() | %s, %s>' % (a,
↳b))
''
sage: magma.eval('O := QuaternionOrder(%s)' % str(list(C[0].right_order().
↳basis()))))
''
sage: [ magma('ideal<O | %s>' % str(list(I.basis()))).Norm() for I in C]
[16, 32, 32]

sage: A.<i,j,k> = QuaternionAlgebra(-1,-1)
sage: R = A.ideal([i,j,k,1/2 + 1/2*i + 1/2*j + 1/2*k])      # this is_
↳actually an order, so has reduced norm 1
sage: R.norm()
1
sage: [ J.norm() for J in R.cyclic_right_subideals(3) ]      # enumerate_
↳maximal right R-ideals of reduced norm 3, verify their norms
[3, 3, 3, 3]
```

```
>>> from sage.all import *
>>> M = BrandtModule(Integer(37))
>>> C = M.right_ideals()
>>> [I.norm() for I in C]
[16, 32, 32]

>>> # optional - magma
>>> (a,b) = M.quaternion_algebra().invariants()
>>> magma.eval('A<i,j,k> := QuaternionAlgebra<Rationals() | %s, %s>' % (a,b))
''
>>> magma.eval('O := QuaternionOrder(%s)' % str(list(C[Integer(0)].right_
↳order().basis()))))
''
>>> [ magma('ideal<O | %s>' % str(list(I.basis()))).Norm() for I in C]
[16, 32, 32]

>>> A = QuaternionAlgebra(-Integer(1),-Integer(1), names=('i', 'j', 'k'));
↳(i, j, k) = A._first_ngens(3)
```

(continues on next page)

(continued from previous page)

```

>>> R = A.ideal([i,j,k,Integer(1)/Integer(2) + Integer(1)/Integer(2)*i +
↳Integer(1)/Integer(2)*j + Integer(1)/Integer(2)*k])      # this is actually
↳an order, so has reduced norm 1
>>> R.norm()
1
>>> [ J.norm() for J in R.cyclic_right_subideals(Integer(3)) ]      #
↳enumerate maximal right R-ideals of reduced norm 3, verify their norms
[3, 3, 3, 3]

```

primitive_decomposition()

Let $I = \text{self}$. If I is an integral left $\mathcal{O}$ -ideal return its decomposition as an equivalent primitive ideal and an integer such that their product is the initial ideal.

OUTPUTS: A primitive ideal equivalent to I , i.e. an equivalent ideal not contained in $n\mathcal{O}$ for any $n > 0$, and the smallest integer g such that $I \subset g\mathcal{O}$.

EXAMPLES:

```

sage: A.<i,j,k> = QuaternionAlgebra(QQ, -1,-11)
sage: I = A.ideal([1/2 + 1/2*i + 1/2*j + 3/2*k, i + k, j + k, 2*k])
sage: I.primitive_decomposition()
(Fractional ideal (1/2 + 1/2*i + 1/2*j + 3/2*k, i + k, j + k, 2*k), 1)
sage: J = A.ideal([7/2 + 7/2*i + 49/2*j + 91/2*k, 7*i + 21*k, 35*j + 35*k,
↳70*k])
sage: Jequiv, g = J.primitive_decomposition()
sage: Jequiv*g == J
True
sage: Jequiv, g
(Fractional ideal (10, 5 + 5*i, 3 + j, 13/2 + 7/2*i + 1/2*j + 1/2*k), 7)

```

```

>>> from sage.all import *
>>> A = QuaternionAlgebra(QQ, -Integer(1),-Integer(11), names=('i', 'j', 'k',
↳)); (i, j, k,) = A._first_ngens(3)
>>> I = A.ideal([Integer(1)/Integer(2) + Integer(1)/Integer(2)*i + Integer(1)/
↳Integer(2)*j + Integer(3)/Integer(2)*k, i + k, j + k, Integer(2)*k])
>>> I.primitive_decomposition()
(Fractional ideal (1/2 + 1/2*i + 1/2*j + 3/2*k, i + k, j + k, 2*k), 1)
>>> J = A.ideal([Integer(7)/Integer(2) + Integer(7)/Integer(2)*i +
↳Integer(49)/Integer(2)*j + Integer(91)/Integer(2)*k, Integer(7)*i +
↳Integer(21)*k, Integer(35)*j + Integer(35)*k, Integer(70)*k])
>>> Jequiv, g = J.primitive_decomposition()
>>> Jequiv*g == J
True
>>> Jequiv, g
(Fractional ideal (10, 5 + 5*i, 3 + j, 13/2 + 7/2*i + 1/2*j + 1/2*k), 7)

```

pullback(J, side=None)

Compute the ideal which is the pullback of `self` through an ideal J .

Uses Lemma 2.1.7 of [Ler2022]. Only works for integral ideals.

INPUT:

- J – a fractional quaternion ideal with norm coprime to `self` and either left order equal to the right order of `self`, or vice versa

- `side` – string (default: `None`); set to `'left'` or `'right'` to perform pullback of left or right ideals respectively. If `None` the side is determined by the matching left and right orders

OUTPUT: a fractional quaternion ideal

EXAMPLES:

```
sage: B = QuaternionAlgebra(419)
sage: i,j,k = B.gens()
sage: I1 = B.ideal([1/2 + 3/2*j + 2*k, 1/2*i + j + 3/2*k, 3*j, 3*k])
sage: I2 = B.ideal([1/2 + 9/2*j, 1/2*i + 9/2*k, 5*j, 5*k])
sage: I3 = I1.pushforward(I2, side='left')
sage: I3.left_order() == I2.right_order()
True
sage: I3.pullback(I2, side='left') == I1
True
```

```
>>> from sage.all import *
>>> B = QuaternionAlgebra(Integer(419))
>>> i,j,k = B.gens()
>>> I1 = B.ideal([Integer(1)/Integer(2) + Integer(3)/Integer(2)*j +
↳ Integer(2)*k, Integer(1)/Integer(2)*i + j + Integer(3)/Integer(2)*k,
↳ Integer(3)*j, Integer(3)*k])
>>> I2 = B.ideal([Integer(1)/Integer(2) + Integer(9)/Integer(2)*j, Integer(1)/
↳ Integer(2)*i + Integer(9)/Integer(2)*k, Integer(5)*j, Integer(5)*k])
>>> I3 = I1.pushforward(I2, side='left')
>>> I3.left_order() == I2.right_order()
True
>>> I3.pullback(I2, side='left') == I1
True
```

pushforward (*J*, *side=None*)

Compute the ideal which is the pushforward of `self` through an ideal *J*.

Uses Lemma 2.1.7 of [Ler2022]. Only works for integral ideals.

INPUT:

- *J* – a fractional quaternion ideal with norm coprime to `self` and either the same left order or right order as `self`
- `side` – string (default: `None`); set to `'left'` or `'right'` to perform pushforward of left or right ideals respectively. If `None` the side is determined by the matching left or right orders

OUTPUT: a fractional quaternion ideal

EXAMPLES:

```
sage: B = QuaternionAlgebra(419)
sage: i,j,k = B.gens()
sage: I1 = B.ideal([1/2 + 3/2*j + 2*k, 1/2*i + j + 3/2*k, 3*j, 3*k])
sage: I2 = B.ideal([1/2 + 9/2*j, 1/2*i + 9/2*k, 5*j, 5*k])
sage: I1.left_order() == I2.left_order()
True
sage: I1.pushforward(I2, side='left')
Fractional ideal (3, 15*i, 3/2 + 10*i + 1/2*j, 1 + 9/10*i + 1/10*k)
```



```

>>> from sage.all import *
>>> B = QuaternionAlgebra(Integer(419))
>>> i,j,k = B.gens()
>>> I1 = B.ideal([Integer(1)/Integer(2) + Integer(3)/Integer(2)*j +
↳Integer(2)*k, Integer(1)/Integer(2)*i + j + Integer(3)/Integer(2)*k,
↳Integer(3)*j, Integer(3)*k])
>>> I2 = B.ideal([Integer(1)/Integer(2) + Integer(9)/Integer(2)*j, Integer(1)/
↳Integer(2)*i + Integer(9)/Integer(2)*k, Integer(5)*j, Integer(5)*k])
>>> I1.left_order() == I2.left_order()
True
>>> I1.pushforward(I2, side='left')
Fractional ideal (3, 15*i, 3/2 + 10*i + 1/2*j, 1 + 9/10*i + 1/10*k)

```

quadratic_form()

Return the normalized quadratic form associated to this quaternion ideal.

OUTPUT: quadratic form

EXAMPLES:

```

sage: I = BrandtModule(11).right_ideals()[1]
sage: Q = I.quadratic_form(); Q
Quadratic form in 4 variables over Rational Field with coefficients:
[ 2 0 3 2 ]
[ * 2 2 1 ]
[ * * 3 2 ]
[ * * * 2 ]
sage: Q.theta_series(10)
1 + 12*q^2 + 12*q^3 + 12*q^4 + 12*q^5 + 24*q^6 + 24*q^7 + 36*q^8 + 36*q^9 +
↳O(q^10)
sage: I.theta_series(10)
1 + 12*q^2 + 12*q^3 + 12*q^4 + 12*q^5 + 24*q^6 + 24*q^7 + 36*q^8 + 36*q^9 +
↳O(q^10)

```

```

>>> from sage.all import *
>>> I = BrandtModule(Integer(11)).right_ideals()[Integer(1)]
>>> Q = I.quadratic_form(); Q
Quadratic form in 4 variables over Rational Field with coefficients:
[ 2 0 3 2 ]
[ * 2 2 1 ]
[ * * 3 2 ]
[ * * * 2 ]
>>> Q.theta_series(Integer(10))
1 + 12*q^2 + 12*q^3 + 12*q^4 + 12*q^5 + 24*q^6 + 24*q^7 + 36*q^8 + 36*q^9 +
↳O(q^10)
>>> I.theta_series(Integer(10))
1 + 12*q^2 + 12*q^3 + 12*q^4 + 12*q^5 + 24*q^6 + 24*q^7 + 36*q^8 + 36*q^9 +
↳O(q^10)

```

quaternion_algebra()

Return the ambient quaternion algebra that contains this fractional ideal.

This is an alias for *self.ring()*.

EXAMPLES:

```
sage: I = BrandtModule(3, 5).right_ideals()[1]; I
Fractional ideal (8, 40*i, 6 + 28*i + 2*j, 4 + 18*i + 2*k)
sage: I.quaternion_algebra()
Quaternion Algebra (-1, -3) with base ring Rational Field
```

```
>>> from sage.all import *
>>> I = BrandtModule(Integer(3), Integer(5)).right_ideals()[Integer(1)]; I
Fractional ideal (8, 40*i, 6 + 28*i + 2*j, 4 + 18*i + 2*k)
>>> I.quaternion_algebra()
Quaternion Algebra (-1, -3) with base ring Rational Field
```

random_element (*args, **kws)

Return a random element in the rational fractional ideal self.

EXAMPLES:

```
sage: B.<i,j,k> = QuaternionAlgebra(211)
sage: I = B.ideal([1, 1/4*j, 20*(i+k), 2/3*i])
sage: I.random_element() in I
True
```

```
>>> from sage.all import *
>>> B = QuaternionAlgebra(Integer(211), names=('i', 'j', 'k')); (i, j, k,) = B._first_ngens(3)
>>> I = B.ideal([Integer(1), Integer(1)/Integer(4)*j, Integer(20)*(i+k), Integer(2)/Integer(3)*i])
>>> I.random_element() in I
True
```

reduced_basis ()

Let $I = \text{self}$ be a fractional ideal in a (rational) definite quaternion algebra. This function returns an LLL reduced basis of I .

OUTPUT: a tuple of four elements in I forming an LLL reduced basis of I as a lattice

EXAMPLES:

```
sage: B = BrandtModule(2, 37); I = B.right_ideals()[0]
sage: I
Fractional ideal (4, 148*i, 108*i + 4*j, 2 + 2*i + 2*j + 2*k)
sage: I.reduced_basis()
(4, 2 + 2*i + 2*j + 2*k, 2 - 14*i + 14*j - 2*k, 2 - 2*i - 14*j + 14*k)
sage: l = I.reduced_basis()
sage: assert all(l[i].reduced_norm() <= l[i+1].reduced_norm() for i in range(len(l) - 1))

sage: B = QuaternionAlgebra(next_prime(2**50))
sage: O = B.maximal_order()
sage: i, j, k = B.gens()
sage: alpha = 1/2 - 1/2*i + 3/2*j - 7/2*k
sage: I = O*alpha + O*3089622859
sage: I.reduced_basis()[0]
1/2*i + j + 5/2*k
```

```

>>> from sage.all import *
>>> B = BrandtModule(Integer(2), Integer(37)); I = B.right_ideals()[Integer(0)]
>>> I
Fractional ideal (4, 148*i, 108*i + 4*j, 2 + 2*i + 2*j + 2*k)
>>> I.reduced_basis()
(4, 2 + 2*i + 2*j + 2*k, 2 - 14*i + 14*j - 2*k, 2 - 2*i - 14*j + 14*k)
>>> l = I.reduced_basis()
>>> assert all(l[i].reduced_norm() <= l[i+Integer(1)].reduced_norm() for i in
↳ range(len(l) - Integer(1)))

>>> B = QuaternionAlgebra(next_prime(Integer(2)**Integer(50)))
>>> O = B.maximal_order()
>>> i, j, k = B.gens()
>>> alpha = Integer(1)/Integer(2) - Integer(1)/Integer(2)*i + Integer(3)/
↳ Integer(2)*j - Integer(7)/Integer(2)*k
>>> I = O*alpha + O*Integer(3089622859)
>>> I.reduced_basis()[Integer(0)]
1/2*i + j + 5/2*k

```

right_order()

Return the right order associated to this fractional ideal.

OUTPUT: an order in a quaternion algebra

EXAMPLES:

```

sage: I = BrandtModule(389).right_ideals()[1]; I
Fractional ideal (8, 8*i, 2 + 6*i + 2*j, 6 + 3*i + k)
sage: I.right_order()
Order of Quaternion Algebra (-2, -389) with base ring Rational Field
with basis (1/2 + 1/2*j + 1/2*k, 1/4*i + 1/2*j + 1/4*k, j, k)
sage: I.left_order()
Order of Quaternion Algebra (-2, -389) with base ring Rational Field
with basis (1/2 + 1/2*j + 3/2*k, 1/8*i + 1/4*j + 9/8*k, j + k, 2*k)

```

```

>>> from sage.all import *
>>> I = BrandtModule(Integer(389)).right_ideals()[Integer(1)]; I
Fractional ideal (8, 8*i, 2 + 6*i + 2*j, 6 + 3*i + k)
>>> I.right_order()
Order of Quaternion Algebra (-2, -389) with base ring Rational Field
with basis (1/2 + 1/2*j + 1/2*k, 1/4*i + 1/2*j + 1/4*k, j, k)
>>> I.left_order()
Order of Quaternion Algebra (-2, -389) with base ring Rational Field
with basis (1/2 + 1/2*j + 3/2*k, 1/8*i + 1/4*j + 9/8*k, j + k, 2*k)

```

The following is a big consistency check. We take reps for all the right ideal classes of a certain order, take the corresponding left orders, then take ideals in the left orders and from those compute the right order again:

```

sage: B = BrandtModule(11, 19); R = B.right_ideals()
sage: O = [r.left_order() for r in R]
sage: J = [O[i].left_ideal(R[i].basis()) for i in range(len(R))]
sage: len(set(J))
18
sage: len(set([I.right_order() for I in J]))

```

(continues on next page)

(continued from previous page)

```

1
sage: J[0].right_order() == B.order_of_level_N()
True

```

```

>>> from sage.all import *
>>> B = BrandtModule(Integer(11), Integer(19)); R = B.right_ideals()
>>> O = [r.left_order() for r in R]
>>> J = [O[i].left_ideal(R[i].basis()) for i in range(len(R))]
>>> len(set(J))
18
>>> len(set([I.right_order() for I in J]))
1
>>> J[Integer(0)].right_order() == B.order_of_level_N()
True

```

scale(*alpha*, *left=False*)Scale the fractional ideal *self* by multiplying the basis by *alpha*.

INPUT:

- α – nonzero element of quaternion algebra
- *left* – boolean (default: *False*); if *True* multiply α on the left, otherwise multiply α on the right

OUTPUT: a new fractional ideal

EXAMPLES:

```

sage: B = BrandtModule(5, 37); I = B.right_ideals()[0]
sage: i, j, k = B.quaternion_algebra().gens(); I
Fractional ideal (4, 148*i, 2 + 106*i + 2*j, 2 + 147*i + k)
sage: I.scale(i)
Fractional ideal (296, 4*i, 2 + 2*i + 2*j, 212 + 2*i + 2*k)
sage: I.scale(i, left=True)
Fractional ideal (296, 4*i, 294 + 2*i + 2*j, 84 + 2*i + 2*k)
sage: I.scale(i, left=False)
Fractional ideal (296, 4*i, 2 + 2*i + 2*j, 212 + 2*i + 2*k)
sage: i * I.gens()[0]
4*i
sage: I.gens()[0] * i
4*i

```

```

>>> from sage.all import *
>>> B = BrandtModule(Integer(5), Integer(37)); I = B.right_ideals()[Integer(0)]
>>> i, j, k = B.quaternion_algebra().gens(); I
Fractional ideal (4, 148*i, 2 + 106*i + 2*j, 2 + 147*i + k)
>>> I.scale(i)
Fractional ideal (296, 4*i, 2 + 2*i + 2*j, 212 + 2*i + 2*k)
>>> I.scale(i, left=True)
Fractional ideal (296, 4*i, 294 + 2*i + 2*j, 84 + 2*i + 2*k)
>>> I.scale(i, left=False)
Fractional ideal (296, 4*i, 2 + 2*i + 2*j, 212 + 2*i + 2*k)
>>> i * I.gens()[Integer(0)]
4*i

```

(continues on next page)

(continued from previous page)

```
>>> I.gens()[Integer(0)] * i
4*i
```

The scaling element must be nonzero:

```
sage: B.<i,j,k> = QuaternionAlgebra(419)
sage: O = B.quaternion_order([1/2 + 3/2*j, 1/6*i + 2/3*j + 1/2*k, 3*j, k])
sage: O * O.zero()
Traceback (most recent call last):
...
ValueError: the scaling factor must be nonzero
```

```
>>> from sage.all import *
>>> B = QuaternionAlgebra(Integer(419), names=('i', 'j', 'k')); (i, j, k) = B._first_ngens(3)
>>> O = B.quaternion_order([Integer(1)/Integer(2) + Integer(3)/Integer(2)*j, Integer(1)/Integer(6)*i + Integer(2)/Integer(3)*j + Integer(1)/Integer(2)*k, Integer(3)*j, k])
>>> O * O.zero()
Traceback (most recent call last):
...
ValueError: the scaling factor must be nonzero
```

theta_series (*B*, *var*='q')

Return normalized theta series of *self*, as a power series over $\mathbf{Z}$ in the variable *var*, which is 'q' by default.

The normalized theta series is by definition

$$\theta_I(q) = \sum_{x \in I} q^{\frac{N(x)}{N(I)}}.$$

INPUT:

- *B* – positive integer
- *var* – string (default: 'q')

OUTPUT: power series

EXAMPLES:

```
sage: I = BrandtModule(11).right_ideals()[1]; I
Fractional ideal (8, 8*i, 6 + 4*i + 2*j, 4 + 2*i + 2*k)
sage: I.norm()
32
sage: I.theta_series(5)
1 + 12*q^2 + 12*q^3 + 12*q^4 + O(q^5)
sage: I.theta_series(5, 'T')
1 + 12*T^2 + 12*T^3 + 12*T^4 + O(T^5)
sage: I.theta_series(3)
1 + 12*q^2 + O(q^3)
```

```
>>> from sage.all import *
>>> I = BrandtModule(Integer(11)).right_ideals()[Integer(1)]; I
Fractional ideal (8, 8*i, 6 + 4*i + 2*j, 4 + 2*i + 2*k)
```

(continues on next page)

(continued from previous page)

```

>>> I.norm()
32
>>> I.theta_series(Integer(5))
1 + 12*q^2 + 12*q^3 + 12*q^4 + O(q^5)
>>> I.theta_series(Integer(5), 'T')
1 + 12*T^2 + 12*T^3 + 12*T^4 + O(T^5)
>>> I.theta_series(Integer(3))
1 + 12*q^2 + O(q^3)

```

theta_series_vector(*B*)

Return theta series coefficients of *self*, as a vector of *B* integers.

INPUT:

- *B* – positive integer

OUTPUT: vector over $\mathbf{Z}$ with *B* entries

EXAMPLES:

```

sage: I = BrandtModule(37).right_ideals()[1]; I
Fractional ideal (8, 8*i, 2 + 6*i + 2*j, 6 + 3*i + k)
sage: I.theta_series_vector(5)
(1, 0, 2, 2, 6)
sage: I.theta_series_vector(10)
(1, 0, 2, 2, 6, 4, 8, 6, 10, 10)
sage: I.theta_series_vector(5)
(1, 0, 2, 2, 6)

```

```

>>> from sage.all import *
>>> I = BrandtModule(Integer(37)).right_ideals()[Integer(1)]; I
Fractional ideal (8, 8*i, 2 + 6*i + 2*j, 6 + 3*i + k)
>>> I.theta_series_vector(Integer(5))
(1, 0, 2, 2, 6)
>>> I.theta_series_vector(Integer(10))
(1, 0, 2, 2, 6, 4, 8, 6, 10, 10)
>>> I.theta_series_vector(Integer(5))
(1, 0, 2, 2, 6)

```

class sage.algebras.quatalg.quaternion_algebra.**QuaternionOrder**(*A*, *basis*, *check=True*)

Bases: [Parent](#)

An order in a quaternion algebra.

EXAMPLES:

```

sage: QuaternionAlgebra(-1, -7).maximal_order()
Order of Quaternion Algebra (-1, -7) with base ring Rational Field with basis (1/
↪ 2 + 1/2*j, 1/2*i + 1/2*k, j, k)
sage: type(QuaternionAlgebra(-1, -7).maximal_order())
<class 'sage.algebras.quatalg.quaternion_algebra.QuaternionOrder_with_category'>

```

```

>>> from sage.all import *
>>> QuaternionAlgebra(-Integer(1), -Integer(7)).maximal_order()

```

(continues on next page)

(continued from previous page)

```

Order of Quaternion Algebra (-1, -7) with base ring Rational Field with basis (1/
↪2 + 1/2*j, 1/2*i + 1/2*k, j, k)
>>> type(QuaternionAlgebra(-Integer(1), -Integer(7)).maximal_order())
<class 'sage.algebras.quatalg.quaternion_algebra.QuaternionOrder_with_category'>

```

basis()

Return fix choice of basis for this quaternion order.

EXAMPLES:

```

sage: QuaternionAlgebra(-11, -1).maximal_order().basis()
(1/2 + 1/2*i, 1/2*j - 1/2*k, i, -k)

```

```

>>> from sage.all import *
>>> QuaternionAlgebra(-Integer(11), -Integer(1)).maximal_order().basis()
(1/2 + 1/2*i, 1/2*j - 1/2*k, i, -k)

```

basis_matrix()

Return the basis matrix of this quaternion order, for the specific basis returned by `basis()`.

OUTPUT: matrix over $\mathbb{Q}$

EXAMPLES:

```

sage: O = QuaternionAlgebra(-11, -1).maximal_order()
sage: O.basis()
(1/2 + 1/2*i, 1/2*j - 1/2*k, i, -k)
sage: O.basis_matrix()
[ 1/2  1/2   0   0]
[  0   0  1/2 -1/2]
[  0   1   0   0]
[  0   0   0  -1]

```

```

>>> from sage.all import *
>>> O = QuaternionAlgebra(-Integer(11), -Integer(1)).maximal_order()
>>> O.basis()
(1/2 + 1/2*i, 1/2*j - 1/2*k, i, -k)
>>> O.basis_matrix()
[ 1/2  1/2   0   0]
[  0   0  1/2 -1/2]
[  0   1   0   0]
[  0   0   0  -1]

```

Note that the returned matrix is *not* necessarily the same as the basis matrix of the `unit_ideal()`:

```

sage: Q.<i,j,k> = QuaternionAlgebra(-1, -11)
sage: O = Q.quaternion_order([j, i, -1, k])
sage: O.basis_matrix()
[ 0  0  1  0]
[ 0  1  0  0]
[-1  0  0  0]
[ 0  0  0  1]
sage: O.unit_ideal().basis_matrix()

```

(continues on next page)

(continued from previous page)

```
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

```
>>> from sage.all import *
>>> Q = QuaternionAlgebra(-Integer(1), -Integer(11), names=('i', 'j', 'k'));
↳ (i, j, k,) = Q._first_ngens(3)
>>> O = Q.quaternion_order([j, i, -Integer(1), k])
>>> O.basis_matrix()
[ 0  0  1  0]
[ 0  1  0  0]
[-1  0  0  0]
[ 0  0  0  1]
>>> O.unit_ideal().basis_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

discriminant()

Return the discriminant of this order.

This is defined as $\sqrt{\det(\text{Tr}(e_i \bar{e}_j))}$, where $\{e_i\}$ is the basis of the order.

OUTPUT: rational number

EXAMPLES:

```
sage: QuaternionAlgebra(-11, -1).maximal_order().discriminant()
11
sage: S = BrandtModule(11, 5).order_of_level_N()
sage: S.discriminant()
55
sage: type(S.discriminant())
<... 'sage.rings.rational.Rational'>
```

```
>>> from sage.all import *
>>> QuaternionAlgebra(-Integer(11), -Integer(1)).maximal_order().discriminant()
11
>>> S = BrandtModule(Integer(11), Integer(5)).order_of_level_N()
>>> S.discriminant()
55
>>> type(S.discriminant())
<... 'sage.rings.rational.Rational'>
```

free_module()

Return the free $\mathbf{Z}$ -module that corresponds to this order inside the vector space corresponding to the ambient quaternion algebra.

OUTPUT: a free $\mathbf{Z}$ -module of rank 4

EXAMPLES:


```

sage: R = QuaternionAlgebra(-11,-1).maximal_order()
sage: R.basis()
(1/2 + 1/2*i, 1/2*j - 1/2*k, i, -k)
sage: R.free_module()
Free module of degree 4 and rank 4 over Integer Ring
Echelon basis matrix:
[1/2 1/2  0  0]
[  0  1  0  0]
[  0  0 1/2 1/2]
[  0  0  0  1]

```

```

>>> from sage.all import *
>>> R = QuaternionAlgebra(-Integer(11),-Integer(1)).maximal_order()
>>> R.basis()
(1/2 + 1/2*i, 1/2*j - 1/2*k, i, -k)
>>> R.free_module()
Free module of degree 4 and rank 4 over Integer Ring
Echelon basis matrix:
[1/2 1/2  0  0]
[  0  1  0  0]
[  0  0 1/2 1/2]
[  0  0  0  1]

```

gen(*n*)

Return the *n*-th generator.

INPUT:

- *n* – integer between 0 and 3, inclusive

EXAMPLES:

```

sage: R = QuaternionAlgebra(-11,-1).maximal_order(); R
Order of Quaternion Algebra (-11, -1) with base ring Rational Field
with basis (1/2 + 1/2*i, 1/2*j - 1/2*k, i, -k)
sage: R.gen(0)
1/2 + 1/2*i
sage: R.gen(1)
1/2*j - 1/2*k
sage: R.gen(2)
i
sage: R.gen(3)
-k

```

```

>>> from sage.all import *
>>> R = QuaternionAlgebra(-Integer(11),-Integer(1)).maximal_order(); R
Order of Quaternion Algebra (-11, -1) with base ring Rational Field
with basis (1/2 + 1/2*i, 1/2*j - 1/2*k, i, -k)
>>> R.gen(Integer(0))
1/2 + 1/2*i
>>> R.gen(Integer(1))
1/2*j - 1/2*k
>>> R.gen(Integer(2))
i

```

(continues on next page)

(continued from previous page)

```
>>> R.gen(Integer(3))
-k
```

gens()

Return generators for self.

EXAMPLES:

```
sage: QuaternionAlgebra(-1,-7).maximal_order().gens()
(1/2 + 1/2*j, 1/2*i + 1/2*k, j, k)
```

```
>>> from sage.all import *
>>> QuaternionAlgebra(-Integer(1),-Integer(7)).maximal_order().gens()
(1/2 + 1/2*j, 1/2*i + 1/2*k, j, k)
```

intersection(other)

Return the intersection of this order with other.

INPUT:

- other – a quaternion order in the same ambient quaternion algebra

OUTPUT: a quaternion order

EXAMPLES:

```
sage: R = QuaternionAlgebra(-11,-1).maximal_order()
sage: R.intersection(R)
Order of Quaternion Algebra (-11, -1) with base ring Rational Field
with basis (1/2 + 1/2*i, i, 1/2*j + 1/2*k, k)
```

```
>>> from sage.all import *
>>> R = QuaternionAlgebra(-Integer(11),-Integer(1)).maximal_order()
>>> R.intersection(R)
Order of Quaternion Algebra (-11, -1) with base ring Rational Field
with basis (1/2 + 1/2*i, i, 1/2*j + 1/2*k, k)
```

We intersect various orders in the quaternion algebra ramified at 11:

```
sage: B = BrandtModule(11,3)
sage: R = B.maximal_order(); S = B.order_of_level_N()
sage: R.intersection(S)
Order of Quaternion Algebra (-1, -11) with base ring Rational Field
with basis (1/2 + 1/2*j, 1/2*i + 5/2*k, j, 3*k)
sage: R.intersection(S) == S
True
sage: B = BrandtModule(11,5)
sage: T = B.order_of_level_N()
sage: S.intersection(T)
Order of Quaternion Algebra (-1, -11) with base ring Rational Field
with basis (1/2 + 1/2*j, 1/2*i + 23/2*k, j, 15*k)
```

```
>>> from sage.all import *
>>> B = BrandtModule(Integer(11),Integer(3))
```

(continues on next page)

(continued from previous page)

```

>>> R = B.maximal_order(); S = B.order_of_level_N()
>>> R.intersection(S)
Order of Quaternion Algebra (-1, -11) with base ring Rational Field
with basis (1/2 + 1/2*j, 1/2*i + 5/2*k, j, 3*k)
>>> R.intersection(S) == S
True
>>> B = BrandtModule(Integer(11), Integer(5))
>>> T = B.order_of_level_N()
>>> S.intersection(T)
Order of Quaternion Algebra (-1, -11) with base ring Rational Field
with basis (1/2 + 1/2*j, 1/2*i + 23/2*k, j, 15*k)

```

is_maximal()

Check whether the order of `self` is maximal in the ambient quaternion algebra.

Only implemented for quaternion algebras over number fields; for reference, see Theorem 15.5.5 in [Voi2021].

EXAMPLES:

```

sage: p = 11
sage: B = QuaternionAlgebra(QQ, -1, -p)
sage: i, j, k = B.gens()
sage: O0_basis = (1, i, (i+j)/2, (1+i*j)/2)
sage: O0 = B.quaternion_order(O0_basis)
sage: O0.is_maximal()
True
sage: O1 = B.quaternion_order([1, i, j, i*j])
sage: O1.is_maximal()
False

```

```

>>> from sage.all import *
>>> p = Integer(11)
>>> B = QuaternionAlgebra(QQ, -Integer(1), -p)
>>> i, j, k = B.gens()
>>> O0_basis = (Integer(1), i, (i+j)/Integer(2), (Integer(1)+i*j)/Integer(2))
>>> O0 = B.quaternion_order(O0_basis)
>>> O0.is_maximal()
True
>>> O1 = B.quaternion_order([Integer(1), i, j, i*j])
>>> O1.is_maximal()
False

```

isomorphism_to(other, conjugator, B)

Compute an isomorphism from this quaternion order O to another order O' in the same quaternion algebra.

INPUT:

- `conjugator` – boolean (default: `False`); if `True` this method returns a single quaternion $\gamma \in O \cap O'$ of minimal norm such that $O' = \gamma^{-1}O\gamma$, rather than the ring isomorphism it defines
- `B` – positive integer; bound on theta series coefficients to rule out non isomorphic orders

Note

This method is currently only implemented for maximal orders in definite quaternion orders over $\mathbf{Q}$. For a general algorithm, see [KV2010] (Problem IsConjugate).

EXAMPLES:

```
sage: Quat.<i,j,k> = QuaternionAlgebra(-1, -19)
sage: O0 = Quat.quaternion_order([1, i, (i+j)/2, (1+k)/2])
sage: O1 = Quat.quaternion_order([1, 667*i, 1/2+j/2+9*i, (222075/2*i+333*j+k/
↪2)/667])
sage: iso = O0.isomorphism_to(O1)
sage: iso
Ring morphism:
  From: Order of Quaternion Algebra (-1, -19) with base ring Rational Field
        with basis (1, i, 1/2*i + 1/2*j, 1/2 + 1/2*k)
  To:   Order of Quaternion Algebra (-1, -19) with base ring Rational Field
        with basis (1, 667*i, 1/2 + 9*i + 1/2*j, 222075/1334*i + 333/667*j +
↪1/1334*k)
  Defn: i |--> 629/667*i + 36/667*j - 36/667*k
        j |--> -684/667*i + 648/667*j + 19/667*k
        k |--> 684/667*i + 19/667*j + 648/667*k
sage: iso(1)
1
sage: iso(i)
629/667*i + 36/667*j - 36/667*k
sage: iso(i/3)
Traceback (most recent call last):
...
TypeError: 1/3*i fails to convert into the map's domain ...
```

```
>>> from sage.all import *
>>> Quat = QuaternionAlgebra(-Integer(1), -Integer(19), names=('i', 'j', 'k',
↪)); (i, j, k,) = Quat._first_ngens(3)
>>> O0 = Quat.quaternion_order([Integer(1), i, (i+j)/Integer(2),
↪(Integer(1)+k)/Integer(2)])
>>> O1 = Quat.quaternion_order([Integer(1), Integer(667)*i, Integer(1)/
↪Integer(2)+j/Integer(2)+Integer(9)*i, (Integer(222075)/
↪Integer(2)*i+Integer(333)*j+k/Integer(2))/Integer(667)])
>>> iso = O0.isomorphism_to(O1)
>>> iso
Ring morphism:
  From: Order of Quaternion Algebra (-1, -19) with base ring Rational Field
        with basis (1, i, 1/2*i + 1/2*j, 1/2 + 1/2*k)
  To:   Order of Quaternion Algebra (-1, -19) with base ring Rational Field
        with basis (1, 667*i, 1/2 + 9*i + 1/2*j, 222075/1334*i + 333/667*j +
↪1/1334*k)
  Defn: i |--> 629/667*i + 36/667*j - 36/667*k
        j |--> -684/667*i + 648/667*j + 19/667*k
        k |--> 684/667*i + 19/667*j + 648/667*k
>>> iso(Integer(1))
1
```

(continues on next page)

(continued from previous page)

```
>>> iso(i)
629/667*i + 36/667*j - 36/667*k
>>> iso(i/Integer(3))
Traceback (most recent call last):
...
TypeError: 1/3*i fails to convert into the map's domain ...
```

```
sage: gamma = O0.isomorphism_to(O1, conjugator=True); gamma
-36 + j + k
sage: gamma in O0
True
sage: gamma in O1
True
sage: O1.unit_ideal() == ~gamma * O0 * gamma
True
```

```
>>> from sage.all import *
>>> gamma = O0.isomorphism_to(O1, conjugator=True); gamma
-36 + j + k
>>> gamma in O0
True
>>> gamma in O1
True
>>> O1.unit_ideal() == ~gamma * O0 * gamma
True
```

ALGORITHM:

Find a generator of the principal lattice $N \cdot O \cdot O'$ where $N = [O : OcapO']$ using `QuaternionFractionalIdeal_rational.minimal_element()`. An isomorphism is given by conjugation by such an element. Works providing reduced norm of conjugation element is not a ramified prime times a square. To cover cases where it is we repeat the check for orders conjugated by i , j , and k .

left_ideal(gens, check, is_basis=True)

Return the left ideal of this order generated by the given generators.

INPUT:

- gens – list of elements of this quaternion order
- check – boolean (default: True)
- is_basis – boolean (default: False); if True then gens must be a $\mathbb{Z}$ -basis of the ideal

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(-11,-1)
sage: R = Q.maximal_order()
sage: R.left_ideal([a^2 for a in R.basis()], is_basis=True)
Fractional ideal (1 + i, 2*i, j + k, 2*k)
sage: R.left_ideal([a*(i+j) for a in R.basis()], is_basis=True)
Fractional ideal (1/2 + 1/2*i + 1/2*j + 13/2*k, i + j, 6*j + 6*k, 12*k)
```

```
>>> from sage.all import *
>>> Q = QuaternionAlgebra(-Integer(11), -Integer(1), names=('i', 'j', 'k')); _
```

(continues on next page)

(continued from previous page)

```

↪(i, j, k,) = Q._first_ngens(3)
>>> R = Q.maximal_order()
>>> R.left_ideal([a*Integer(2) for a in R.basis()], is_basis=True)
Fractional ideal (1 + i, 2*i, j + k, 2*k)
>>> R.left_ideal([a*(i+j) for a in R.basis()], is_basis=True)
Fractional ideal (1/2 + 1/2*i + 1/2*j + 13/2*k, i + j, 6*j + 6*k, 12*k)

```

It is also possible to pass a generating set (rather than a basis), or a single generator:

```

sage: R.left_ideal([i+j])
Fractional ideal (12, 6 + 6*i, i + j, 13/2 + 1/2*i + 1/2*j + 1/2*k)
sage: R.left_ideal(i+j)
Fractional ideal (12, 6 + 6*i, i + j, 13/2 + 1/2*i + 1/2*j + 1/2*k)
sage: R.left_ideal([2, 1+j]) == R*2 + R*(1+j)
True

```

```

>>> from sage.all import *
>>> R.left_ideal([i+j])
Fractional ideal (12, 6 + 6*i, i + j, 13/2 + 1/2*i + 1/2*j + 1/2*k)
>>> R.left_ideal(i+j)
Fractional ideal (12, 6 + 6*i, i + j, 13/2 + 1/2*i + 1/2*j + 1/2*k)
>>> R.left_ideal([Integer(2), Integer(1)+j]) == R*Integer(2) +
↪R*(Integer(1)+j)
True

```

ngens()

Return the number of generators (which is 4).

EXAMPLES:

```

sage: QuaternionAlgebra(-1,-7).maximal_order().ngens()
4

```

```

>>> from sage.all import *
>>> QuaternionAlgebra(-Integer(1),-Integer(7)).maximal_order().ngens()
4

```

one()

Return the multiplicative unit of this quaternion order.

EXAMPLES:

```

sage: QuaternionAlgebra(-1,-7).maximal_order().one()
1

```

```

>>> from sage.all import *
>>> QuaternionAlgebra(-Integer(1),-Integer(7)).maximal_order().one()
1

```

quadratic_form()

Return the normalized quadratic form associated to this quaternion order.

OUTPUT: quadratic form

EXAMPLES:

```

sage: R = BrandtModule(11,13).order_of_level_N()
sage: Q = R.quadratic_form(); Q
Quadratic form in 4 variables over Rational Field with coefficients:
[ 14 253 55 286 ]
[ * 1455 506 3289 ]
[ * * 55 572 ]
[ * * * 1859 ]
sage: Q.theta_series(10)
1 + 2*q + 2*q^4 + 4*q^6 + 4*q^8 + 2*q^9 + O(q^10)

```

```

>>> from sage.all import *
>>> R = BrandtModule(Integer(11),Integer(13)).order_of_level_N()
>>> Q = R.quadratic_form(); Q
Quadratic form in 4 variables over Rational Field with coefficients:
[ 14 253 55 286 ]
[ * 1455 506 3289 ]
[ * * 55 572 ]
[ * * * 1859 ]
>>> Q.theta_series(Integer(10))
1 + 2*q + 2*q^4 + 4*q^6 + 4*q^8 + 2*q^9 + O(q^10)

```

quaternion_algebra()

Return ambient quaternion algebra that contains this quaternion order.

EXAMPLES:

```

sage: QuaternionAlgebra(-11,-1).maximal_order().quaternion_algebra()
Quaternion Algebra (-11, -1) with base ring Rational Field

```

```

>>> from sage.all import *
>>> QuaternionAlgebra(-Integer(11),-Integer(1)).maximal_order().quaternion_
↪algebra()
Quaternion Algebra (-11, -1) with base ring Rational Field

```

random_element(*args,kws)**

Return a random element of this order.

The args and kws are passed to the random_element method of the integer ring, and we return an element of the form

$$ae_1 + be_2 + ce_3 + de_4$$

where $e_1, \dots, e_4$ are the basis of this order and a, b, c, d are random integers.

EXAMPLES:

```

sage: QuaternionAlgebra(-11,-1).maximal_order().random_element() # random
-4 - 4*i + j - k
sage: QuaternionAlgebra(-11,-1).maximal_order().random_element(-10,10) #_
↪random
-9/2 - 7/2*i - 7/2*j - 3/2*k

```

```
>>> from sage.all import *
>>> QuaternionAlgebra(-Integer(11), -Integer(1)).maximal_order().random_
↪element() # random
-4 - 4*i + j - k
>>> QuaternionAlgebra(-Integer(11), -Integer(1)).maximal_order().random_
↪element(-Integer(10), Integer(10)) # random
-9/2 - 7/2*i - 7/2*j - 3/2*k
```

right_ideal (*gens*, *check*, *is_basis=True*)

Return the right ideal of this order generated by the given generators.

INPUT:

- *gens* – list of elements of this quaternion order
- *check* – boolean (default: True)
- *is_basis* – boolean (default: False); if True then *gens* must be a **Z**-basis of the ideal

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(-11,-1)
sage: R = Q.maximal_order()
sage: R.right_ideal([2*a for a in R.basis()], is_basis=True)
Fractional ideal (1 + i, 2*i, j + k, 2*k)
sage: R.right_ideal([(i+j)*a for a in R.basis()], is_basis=True)
Fractional ideal (1/2 + 1/2*i + 1/2*j + 11/2*k, i + j, 6*j + 6*k, 12*k)
```

```
>>> from sage.all import *
>>> Q = QuaternionAlgebra(-Integer(11), -Integer(1), names=('i', 'j', 'k'));
↪(i, j, k) = Q._first_ngens(3)
>>> R = Q.maximal_order()
>>> R.right_ideal([Integer(2)*a for a in R.basis()], is_basis=True)
Fractional ideal (1 + i, 2*i, j + k, 2*k)
>>> R.right_ideal([(i+j)*a for a in R.basis()], is_basis=True)
Fractional ideal (1/2 + 1/2*i + 1/2*j + 11/2*k, i + j, 6*j + 6*k, 12*k)
```

It is also possible to pass a generating set (rather than a basis), or a single generator:

```
sage: R.right_ideal([i+j])
Fractional ideal (12, 6 + 6*i, i + j, 11/2 + 1/2*i + 1/2*j + 1/2*k)
sage: R.right_ideal(i+j)
Fractional ideal (12, 6 + 6*i, i + j, 11/2 + 1/2*i + 1/2*j + 1/2*k)
sage: R.right_ideal([2, 1+j]) == 2*R + (1+j)*R
True
```

```
>>> from sage.all import *
>>> R.right_ideal([i+j])
Fractional ideal (12, 6 + 6*i, i + j, 11/2 + 1/2*i + 1/2*j + 1/2*k)
>>> R.right_ideal(i+j)
Fractional ideal (12, 6 + 6*i, i + j, 11/2 + 1/2*i + 1/2*j + 1/2*k)
>>> R.right_ideal([Integer(2), Integer(1)+j]) == Integer(2)*R +
↪(Integer(1)+j)*R
True
```


ternary_quadratic_form (*include_basis=False*)

Return the ternary quadratic form associated to this order.

INPUT:

- *include_basis* – boolean (default: False); if True also return a basis for the dimension 3 subspace G

OUTPUT: QuadraticForm

- optional basis for dimension 3 subspace

This function computes the positive definition quadratic form obtained by letting G be the trace zero subspace of $\mathbf{Z} + 2 * \text{self}$, which has rank 3, and restricting the pairing `QuaternionAlgebraElement_abstract.pair()`:

```
(x,y) = (x.conjugate()*y).reduced_trace()
```

to G .

APPLICATIONS: Ternary quadratic forms associated to an order in a rational quaternion algebra are useful in computing with Gross points, in deciding whether quaternion orders have embeddings from orders in quadratic imaginary fields, and in computing elements of the Kohnen plus subspace of modular forms of weight $3/2$.

EXAMPLES:

```
sage: R = BrandtModule(11,13).order_of_level_N()
sage: Q = R.ternary_quadratic_form(); Q
Quadratic form in 3 variables over Rational Field with coefficients:
[ 5820 1012 13156 ]
[ * 55 1144 ]
[ * * 7436 ]
sage: factor(Q.disc())
2^4 * 11^2 * 13^2
```

```
>>> from sage.all import *
>>> R = BrandtModule(Integer(11), Integer(13)).order_of_level_N()
>>> Q = R.ternary_quadratic_form(); Q
Quadratic form in 3 variables over Rational Field with coefficients:
[ 5820 1012 13156 ]
[ * 55 1144 ]
[ * * 7436 ]
>>> factor(Q.disc())
2^4 * 11^2 * 13^2
```

The following theta series is a modular form of weight $3/2$ and level $4*11*13$:

```
sage: Q.theta_series(100)
1 + 2*q^23 + 2*q^55 + 2*q^56 + 2*q^75 + 4*q^92 + O(q^100)
```

```
>>> from sage.all import *
>>> Q.theta_series(Integer(100))
1 + 2*q^23 + 2*q^55 + 2*q^56 + 2*q^75 + 4*q^92 + O(q^100)
```

unit_ideal ()

Return the unit ideal in this quaternion order.

EXAMPLES:

```
sage: R = QuaternionAlgebra(-11,-1).maximal_order()
sage: I = R.unit_ideal(); I
Fractional ideal (1/2 + 1/2*i, 1/2*j - 1/2*k, i, -k)
```

```
>>> from sage.all import *
>>> R = QuaternionAlgebra(-Integer(11),-Integer(1)).maximal_order()
>>> I = R.unit_ideal(); I
Fractional ideal (1/2 + 1/2*i, 1/2*j - 1/2*k, i, -k)
```

`sage.algebras.quatalg.quaternion_algebra.basis_for_quaternion_lattice(gens, reverse=True)`

Return a basis for the $\mathbf{Z}$ -lattice in a quaternion algebra spanned by the given gens.

INPUT:

- `gens` – list of elements of a single quaternion algebra
- `reverse` – when computing the HNF do it on the basis $(k, j, i, 1)$ instead of $(1, i, j, k)$; this ensures that if `gens` are the generators for a fractional ideal (in particular, an order), the first returned basis vector equals the norm of the ideal (in case of an order, 1)

EXAMPLES:

```
sage: from sage.algebras.quatalg.quaternion_algebra import basis_for_quaternion_
      ↪ lattice
sage: A.<i,j,k> = QuaternionAlgebra(-1,-7)
sage: basis_for_quaternion_lattice([i+j, i-j, 2*k, A(1/3)])
[1/3, 2*i, i + j, 2*k]
sage: basis_for_quaternion_lattice([A(1), i, j, k])
[1, i, j, k]
```

```
>>> from sage.all import *
>>> from sage.algebras.quatalg.quaternion_algebra import basis_for_quaternion_
      ↪ lattice
>>> A = QuaternionAlgebra(-Integer(1),-Integer(7), names=('i', 'j', 'k')); (i, j,
      ↪ k) = A._first_ngens(3)
>>> basis_for_quaternion_lattice([i+j, i-j, Integer(2)*k, A(Integer(1)/
      ↪ Integer(3))])
[1/3, 2*i, i + j, 2*k]
>>> basis_for_quaternion_lattice([A(Integer(1)), i, j, k])
[1, i, j, k]
```

`sage.algebras.quatalg.quaternion_algebra.intersection_of_row_modules_over_ZZ(v)`

Intersect the $\mathbf{Z}$ -modules with basis matrices the full rank 4×4 $\mathbf{Q}$ -matrices in the list `v`.

The returned intersection is represented by a 4×4 matrix over $\mathbf{Q}$. This can also be done using modules and intersection, but that would take over twice as long because of overhead, hence this function.

EXAMPLES:

```
sage: a = matrix(QQ,4,[-2, 0, 0, 0, 0, -1, -1, 1, 2, -1/2, 0, 0, 1, 1, -1, 0])
sage: b = matrix(QQ,4,[0, -1/2, 0, -1/2, 2, 1/2, -1, -1/2, 1, 2, 1, -2, 0, -1/2, -
      ↪ 2, 0])
sage: c = matrix(QQ,4,[0, 1, 0, -1/2, 0, 0, 2, 2, 0, -1/2, 1/2, -1, 1, -1, -1/2, -
      ↪ 0])
sage: v = [a,b,c]
```

(continues on next page)

(continued from previous page)

```

sage: from sage.algebras.quatalg.quaternion_algebra import intersection_of_row_
      ↪modules_over_ZZ
sage: M = intersection_of_row_modules_over_ZZ(v); M
[  2    0   -1   -1]
[ -4    1    1   -3]
[ -3 19/2   -1   -4]
[  2   -3   -8    4]
sage: M2 = a.row_module(ZZ).intersection(b.row_module(ZZ)).intersection(c.row_
      ↪module(ZZ))
sage: M.row_module(ZZ) == M2
True

```

```

>>> from sage.all import *
>>> a = matrix(QQ,Integer(4),[-Integer(2), Integer(0), Integer(0), Integer(0),
      ↪Integer(0), -Integer(1), -Integer(1), Integer(1), Integer(2), -Integer(1)/
      ↪Integer(2), Integer(0), Integer(0), Integer(1), Integer(1), -Integer(1),
      ↪Integer(0)])
>>> b = matrix(QQ,Integer(4),[Integer(0), -Integer(1)/Integer(2), Integer(0), -
      ↪Integer(1)/Integer(2), Integer(2), Integer(1)/Integer(2), -Integer(1), -
      ↪Integer(1)/Integer(2), Integer(1), Integer(2), Integer(1), -Integer(2),
      ↪Integer(0), -Integer(1)/Integer(2), -Integer(2), Integer(0)])
>>> c = matrix(QQ,Integer(4),[Integer(0), Integer(1), Integer(0), -Integer(1)/
      ↪Integer(2), Integer(0), Integer(0), Integer(2), Integer(2), Integer(0), -
      ↪Integer(1)/Integer(2), Integer(1)/Integer(2), -Integer(1), Integer(1), -
      ↪Integer(1), -Integer(1)/Integer(2), Integer(0)])
>>> v = [a,b,c]
>>> from sage.algebras.quatalg.quaternion_algebra import intersection_of_row_
      ↪modules_over_ZZ
>>> M = intersection_of_row_modules_over_ZZ(v); M
[  2    0   -1   -1]
[ -4    1    1   -3]
[ -3 19/2   -1   -4]
[  2   -3   -8    4]
>>> M2 = a.row_module(ZZ).intersection(b.row_module(ZZ)).intersection(c.row_
      ↪module(ZZ))
>>> M.row_module(ZZ) == M2
True

```

`sage.algebras.quatalg.quaternion_algebra.is_QuaternionAlgebra(A)`

Return True if A is of the QuaternionAlgebra data type.

EXAMPLES:

```

sage: sage.algebras.quatalg.quaternion_algebra.is_
      ↪QuaternionAlgebra(QuaternionAlgebra(QQ,-1,-1))
doctest:warning...
DeprecationWarning: the function is_QuaternionAlgebra is deprecated;
use 'isinstance(..., QuaternionAlgebra_abstract)' instead
See https://github.com/sagemath/sage/issues/37896 for details.
True
sage: sage.algebras.quatalg.quaternion_algebra.is_QuaternionAlgebra(ZZ)
False

```

```

>>> from sage.all import *
>>> sage.algebras.quatalg.quaternion_algebra.is_
↳QuaternionAlgebra(QuaternionAlgebra(QQ,-Integer(1),-Integer(1)))
doctest:warning...
DeprecationWarning: the function is_QuaternionAlgebra is deprecated;
use 'isinstance(..., QuaternionAlgebra_abstract)' instead
See https://github.com/sagemath/sage/issues/37896 for details.
True
>>> sage.algebras.quatalg.quaternion_algebra.is_QuaternionAlgebra(ZZ)
False

```

`sage.algebras.quatalg.quaternion_algebra.maxord_solve_aux_eq(a, b, p)`

Given a and b and an even prime ideal p find (y, z, w) with y a unit mod p^{2e} such that

$$1 - ay^2 - bz^2 + abw^2 \equiv 0 \pmod{p^{2e}},$$

where e is the ramification index of p .

Currently only $p = 2$ is implemented by hardcoding solutions.

INPUT:

- a – integer with $v_p(a) = 0$
- b – integer with $v_p(b) \in \{0, 1\}$
- p – even prime ideal (actually only $p = \mathbb{Z}(2)$ is implemented)

OUTPUT: a tuple (y, z, w)

EXAMPLES:

```

sage: from sage.algebras.quatalg.quaternion_algebra import maxord_solve_aux_eq
sage: for a in [1, 3]:
....:     for b in [1, 2, 3]:
....:         (y, z, w) = maxord_solve_aux_eq(a, b, 2)
....:         assert mod(y, 4) == 1 or mod(y, 4) == 3
....:         assert mod(1 - a*y^2 - b*z^2 + a*b*w^2, 4) == 0

```

```

>>> from sage.all import *
>>> from sage.algebras.quatalg.quaternion_algebra import maxord_solve_aux_eq
>>> for a in [Integer(1), Integer(3)]:
...     for b in [Integer(1), Integer(2), Integer(3)]:
...         (y, z, w) = maxord_solve_aux_eq(a, b, Integer(2))
...         assert mod(y, Integer(4)) == Integer(1) or mod(y, Integer(4)) ==
↳Integer(3)
...         assert mod(Integer(1) - a*y**Integer(2) - b*z**Integer(2) +
↳a*b*w**Integer(2), Integer(4)) == Integer(0)

```

`sage.algebras.quatalg.quaternion_algebra.normalize_basis_at_p(e, p, B=<method 'pair' of 'sage.algebras.quatalg.quaternion_algebra_element.QuaternionAlgebraElement_abstract' objects>)`

Compute a (at p) normalized basis from the given basis e of a $\mathbb{Z}$ -module.

The returned basis is (at p) a $\mathbb{Z}_p$ basis for the same module, and has the property that with respect to it the quadratic form induced by the bilinear form B is represented as a orthogonal sum of atomic forms multiplied by p -powers.

If $p \neq 2$ this means that the form is diagonal with respect to this basis.

If $p = 2$ there may be additional 2-dimensional subspaces on which the form is represented as $2^e(ax^2 + bxy + cx^2)$ with $0 = v_2(b) = v_2(a) \leq v_2(c)$.

INPUT:

- e – list; basis of a $\mathbf{Z}$ module (WARNING: will be modified!)
- p – prime for at which the basis should be normalized
- B – (default: `QuaternionAlgebraElement_abstract.pair()`) a bilinear form with respect to which to normalize

OUTPUT:

- A list containing two-element tuples: The first element of each tuple is a basis element, the second the valuation of the orthogonal summand to which it belongs. The list is sorted by ascending valuation.

EXAMPLES:

```
sage: from sage.algebras.quatalg.quaternion_algebra import normalize_basis_at_p
sage: A.<i,j,k> = QuaternionAlgebra(-1, -1)
sage: e = [A(1), i, j, k]
sage: normalize_basis_at_p(e, 2)
[(1, 0), (i, 0), (j, 0), (k, 0)]

sage: A.<i,j,k> = QuaternionAlgebra(210)
sage: e = [A(1), i, j, k]
sage: normalize_basis_at_p(e, 2)
[(1, 0), (i, 1), (j, 1), (k, 2)]

sage: A.<i,j,k> = QuaternionAlgebra(286)
sage: e = [A(1), k, 1/2*j + 1/2*k, 1/2 + 1/2*i + 1/2*k]
sage: normalize_basis_at_p(e, 5)
[(1, 0), (1/2*j + 1/2*k, 0), (-5/6*j + 1/6*k, 1), (1/2*i, 1)]

sage: A.<i,j,k> = QuaternionAlgebra(-1,-7)
sage: e = [A(1), k, j, 1/2 + 1/2*i + 1/2*j + 1/2*k]
sage: normalize_basis_at_p(e, 2)
[(1, 0), (1/2 + 1/2*i + 1/2*j + 1/2*k, 0), (-34/105*i - 463/735*j + 71/105*k, 1),
(1/7*i - 8/49*j + 1/7*k, 1)]
```

```
>>> from sage.all import *
>>> from sage.algebras.quatalg.quaternion_algebra import normalize_basis_at_p
>>> A = QuaternionAlgebra(-Integer(1), -Integer(1), names=('i', 'j', 'k')); (i, j, k) = A._first_ngens(3)
>>> e = [A(Integer(1)), i, j, k]
>>> normalize_basis_at_p(e, Integer(2))
[(1, 0), (i, 0), (j, 0), (k, 0)]

>>> A = QuaternionAlgebra(Integer(210), names=('i', 'j', 'k')); (i, j, k) = A._first_ngens(3)
>>> e = [A(Integer(1)), i, j, k]
>>> normalize_basis_at_p(e, Integer(2))
[(1, 0), (i, 1), (j, 1), (k, 2)]
```

(continues on next page)

(continued from previous page)

```

>>> A = QuaternionAlgebra(Integer(286), names=('i', 'j', 'k',)); (i, j, k,) = A._
↳ first_ngens(3)
>>> e = [A(Integer(1)), k, Integer(1)/Integer(2)*j + Integer(1)/Integer(2)*k,
↳ Integer(1)/Integer(2) + Integer(1)/Integer(2)*i + Integer(1)/Integer(2)*k]
>>> normalize_basis_at_p(e, Integer(5))
[(1, 0), (1/2*j + 1/2*k, 0), (-5/6*j + 1/6*k, 1), (1/2*i, 1)]

>>> A = QuaternionAlgebra(-Integer(1), -Integer(7), names=('i', 'j', 'k',)); (i, j,
↳ k,) = A._first_ngens(3)
>>> e = [A(Integer(1)), k, j, Integer(1)/Integer(2) + Integer(1)/Integer(2)*i +
↳ Integer(1)/Integer(2)*j + Integer(1)/Integer(2)*k]
>>> normalize_basis_at_p(e, Integer(2))
[(1, 0), (1/2 + 1/2*i + 1/2*j + 1/2*k, 0), (-34/105*i - 463/735*j + 71/105*k, 1),
(1/7*i - 8/49*j + 1/7*k, 1)]

```

sage.algebras.quatalg.quaternion_algebra.**unpickle_QuaternionAlgebra_v0**(*key)

The 0-th version of pickling for quaternion algebras.

EXAMPLES:

```

sage: Q = QuaternionAlgebra(-5, -19)
sage: t = (QQ, -5, -19, ('i', 'j', 'k'))
sage: sage.algebras.quatalg.quaternion_algebra.unpickle_QuaternionAlgebra_v0(*t)
Quaternion Algebra (-5, -19) with base ring Rational Field
sage: loads(dumps(Q)) == Q
True
sage: loads(dumps(Q)) is Q
True

```

```

>>> from sage.all import *
>>> Q = QuaternionAlgebra(-Integer(5), -Integer(19))
>>> t = (QQ, -Integer(5), -Integer(19), ('i', 'j', 'k'))
>>> sage.algebras.quatalg.quaternion_algebra.unpickle_QuaternionAlgebra_v0(*t)
Quaternion Algebra (-5, -19) with base ring Rational Field
>>> loads(dumps(Q)) == Q
True
>>> loads(dumps(Q)) is Q
True

```

5.22 Rational Cherednik Algebras

class sage.algebras.rational_cherednik_algebra.**RationalCherednikAlgebra**(*ct, c, t, base_ring, prefix*)

Bases: `CombinatorialFreeModule`

A rational Cherednik algebra.

Let k be a field. Let W be a complex reflection group acting on a vector space $\mathfrak{h}$ (over k). Let $\mathfrak{h}^*$ denote the corresponding dual vector space. Let $\cdot$ denote the natural action of w on $\mathfrak{h}$ and $\mathfrak{h}^*$. Let $\mathcal{S}$ denote the set of reflections of W and α_s and $\alpha_s^\vee$ are the associated root and coroot of s . Let $c = (c_s)_{s \in W}$ such that $c_s = c_{tst^{-1}}$ for all $t \in W$.

The *rational Cherednik algebra* is the k -algebra $H_{c,t}(W) = T(\mathfrak{h} \oplus \mathfrak{h}^*) \otimes kW$ with parameters $c, t \in k$ that is subject to the relations:

$$\begin{aligned} w\alpha &= (w \cdot \alpha)w, \\ \alpha^\vee w &= w(w^{-1} \cdot \alpha^\vee), \\ \alpha\alpha^\vee &= \alpha^\vee\alpha + t\langle \alpha^\vee, \alpha \rangle + \sum_{s \in S} c_s \frac{\langle \alpha^\vee, \alpha_s \rangle \langle \alpha_s^\vee, \alpha \rangle}{\langle \alpha^\vee, \alpha \rangle} s, \end{aligned}$$

where $w \in W$ and $\alpha \in \mathfrak{h}$ and $\alpha^\vee \in \mathfrak{h}^*$.

INPUT:

- `ct` – a finite Cartan type
- `c` – the parameters c_s given as an element or a tuple, where the first entry is the one for the long roots and (for non-simply-laced types) the second is for the short roots
- `t` – the parameter t
- `base_ring` – (optional) the base ring
- `prefix` – (default: ('a', 's', 'ac')) the prefixes

Todo

Implement a version for complex reflection groups.

REFERENCES:

- [GGOR2003]
- [EM2001]

algebra_generators()

Return the algebra generators of `self`.

EXAMPLES:

```
sage: R = algebras.RationalCherednik(['A', 2], 1, 1, QQ)
sage: list(R.algebra_generators())
[a1, a2, s1, s2, ac1, ac2]
```

```
>>> from sage.all import *
>>> R = algebras.RationalCherednik(['A', Integer(2)], Integer(1), Integer(1),
↳ QQ)
>>> list(R.algebra_generators())
[a1, a2, s1, s2, ac1, ac2]
```

an_element()

Return an element of `self`.

EXAMPLES:

```
sage: R = algebras.RationalCherednik(['A', 2], 1, 1, QQ)
sage: R.an_element()
3*ac1 + 2*s1 + a1
```

```
>>> from sage.all import *
>>> R = algebras.RationalCherednik(['A', Integer(2)], Integer(1), Integer(1),
↳ QQ)
>>> R.an_element()
3*ac1 + 2*s1 + a1
```

deformed_euler()

Return the element eu_k .

EXAMPLES:

```
sage: R = algebras.RationalCherednik(['A', 2], 1, 1, QQ)
sage: R.deformed_euler()
2*I + 2/3*a1*ac1 + 1/3*a1*ac2 + 1/3*a2*ac1 + 2/3*a2*ac2
+ s1 + s2 + s1*s2*s1
```

```
>>> from sage.all import *
>>> R = algebras.RationalCherednik(['A', Integer(2)], Integer(1), Integer(1),
↳ QQ)
>>> R.deformed_euler()
2*I + 2/3*a1*ac1 + 1/3*a1*ac2 + 1/3*a2*ac1 + 2/3*a2*ac2
+ s1 + s2 + s1*s2*s1
```

degree_on_basis(m)

Return the degree on the monomial indexed by m .

EXAMPLES:

```
sage: R = algebras.RationalCherednik(['A', 2], 1, 1, QQ)
sage: [R.degree_on_basis(g.leading_support())
....: for g in R.algebra_generators()]
[1, 1, 0, 0, -1, -1]
```

```
>>> from sage.all import *
>>> R = algebras.RationalCherednik(['A', Integer(2)], Integer(1), Integer(1),
↳ QQ)
>>> [R.degree_on_basis(g.leading_support())
... for g in R.algebra_generators()]
[1, 1, 0, 0, -1, -1]
```

one_basis()

Return the index of the element 1.

EXAMPLES:

```
sage: R = algebras.RationalCherednik(['A', 2], 1, 1, QQ)
sage: R.one_basis()
(1, 1, 1)
```

```
>>> from sage.all import *
>>> R = algebras.RationalCherednik(['A', Integer(2)], Integer(1), Integer(1),
↳ QQ)
>>> R.one_basis()
(1, 1, 1)
```


product_on_basis (*left, right*)

Return left multiplied by right in self.

EXAMPLES:

```
sage: R = algebras.RationalCherednik(['A', 2], 1, 1, QQ)
sage: a2 = R.algebra_generators()['a2']
sage: ac1 = R.algebra_generators()['ac1']
sage: a2 * ac1 # indirect doctest
a2*ac1
sage: ac1 * a2
-I + a2*ac1 - s1 - s2 + 1/2*s1*s2*s1
sage: x = R.an_element()
sage: [y * x for y in R.some_elements()]
[0,
 3*ac1 + 2*s1 + a1,
 9*ac1^2 + 10*I + 6*a1*ac1 + 6*s1 + 3/2*s2 + 3/2*s1*s2*s1 + a1^2,
 3*a1*ac1 + 2*a1*s1 + a1^2,
 3*a2*ac1 + 2*a2*s1 + a1*a2,
 3*s1*ac1 + 2*I - a1*s1,
 3*s2*ac1 + 2*s2*s1 + a1*s2 + a2*s2,
 3*ac1^2 - 2*s1*ac1 + 2*I + a1*ac1 + 2*s1 + 1/2*s2 + 1/2*s1*s2*s1,
 3*ac1*ac2 + 2*s1*ac1 + 2*s1*ac2 - I + a1*ac2 - s1 - s2 + 1/2*s1*s2*s1]
sage: [x * y for y in R.some_elements()]
[0,
 3*ac1 + 2*s1 + a1,
 9*ac1^2 + 10*I + 6*a1*ac1 + 6*s1 + 3/2*s2 + 3/2*s1*s2*s1 + a1^2,
 6*I + 3*a1*ac1 + 6*s1 + 3/2*s2 + 3/2*s1*s2*s1 - 2*a1*s1 + a1^2,
 -3*I + 3*a2*ac1 - 3*s1 - 3*s2 + 3/2*s1*s2*s1 + 2*a1*s1 + 2*a2*s1 + a1*a2,
 -3*s1*ac1 + 2*I + a1*s1,
 3*s2*ac1 + 3*s2*ac2 + 2*s1*s2 + a1*s2,
 3*ac1^2 + 2*s1*ac1 + a1*ac1,
 3*ac1*ac2 + 2*s1*ac2 + a1*ac2]
```

```
>>> from sage.all import *
>>> R = algebras.RationalCherednik(['A', Integer(2)], Integer(1), Integer(1), QQ)
>>> a2 = R.algebra_generators()['a2']
>>> ac1 = R.algebra_generators()['ac1']
>>> a2 * ac1 # indirect doctest
a2*ac1
>>> ac1 * a2
-I + a2*ac1 - s1 - s2 + 1/2*s1*s2*s1
>>> x = R.an_element()
>>> [y * x for y in R.some_elements()]
[0,
 3*ac1 + 2*s1 + a1,
 9*ac1^2 + 10*I + 6*a1*ac1 + 6*s1 + 3/2*s2 + 3/2*s1*s2*s1 + a1^2,
 3*a1*ac1 + 2*a1*s1 + a1^2,
 3*a2*ac1 + 2*a2*s1 + a1*a2,
 3*s1*ac1 + 2*I - a1*s1,
 3*s2*ac1 + 2*s2*s1 + a1*s2 + a2*s2,
 3*ac1^2 - 2*s1*ac1 + 2*I + a1*ac1 + 2*s1 + 1/2*s2 + 1/2*s1*s2*s1,
```

(continues on next page)

(continued from previous page)

```

3*ac1*ac2 + 2*s1*ac1 + 2*s1*ac2 - I + a1*ac2 - s1 - s2 + 1/2*s1*s2*s1]
>>> [x * y for y in R.some_elements()]
[0,
 3*ac1 + 2*s1 + a1,
 9*ac1^2 + 10*I + 6*a1*ac1 + 6*s1 + 3/2*s2 + 3/2*s1*s2*s1 + a1^2,
 6*I + 3*a1*ac1 + 6*s1 + 3/2*s2 + 3/2*s1*s2*s1 - 2*a1*s1 + a1^2,
 -3*I + 3*a2*ac1 - 3*s1 - 3*s2 + 3/2*s1*s2*s1 + 2*a1*s1 + 2*a2*s1 + a1*a2,
 -3*s1*ac1 + 2*I + a1*s1,
 3*s2*ac1 + 3*s2*ac2 + 2*s1*s2 + a1*s2,
 3*ac1^2 + 2*s1*ac1 + a1*ac1,
 3*ac1*ac2 + 2*s1*ac2 + a1*ac2]

```

some_elements()Return some elements of `self`.

EXAMPLES:

```

sage: R = algebras.RationalCherednik(['A',2], 1, 1, QQ)
sage: R.some_elements()
[0, I, 3*ac1 + 2*s1 + a1, a1, a2, s1, s2, ac1, ac2]

```

```

>>> from sage.all import *
>>> R = algebras.RationalCherednik(['A',Integer(2)], Integer(1), Integer(1), QQ)
>>> R.some_elements()
[0, I, 3*ac1 + 2*s1 + a1, a1, a2, s1, s2, ac1, ac2]

```

trivial_idempotent()Return the trivial idempotent of `self`.

Let $e = |W|^{-1} \sum_{w \in W} w$ is the trivial idempotent. Thus $e^2 = e$ and $eW = We$. The trivial idempotent is used in the construction of the spherical Cherednik algebra from the rational Cherednik algebra by $U_{c,t}(W) = eH_{c,t}(W)e$.

EXAMPLES:

```

sage: R = algebras.RationalCherednik(['A',2], 1, 1, QQ)
sage: R.trivial_idempotent()
1/6*I + 1/6*s1 + 1/6*s2 + 1/6*s2*s1 + 1/6*s1*s2 + 1/6*s1*s2*s1

```

```

>>> from sage.all import *
>>> R = algebras.RationalCherednik(['A',Integer(2)], Integer(1), Integer(1), QQ)
>>> R.trivial_idempotent()
1/6*I + 1/6*s1 + 1/6*s2 + 1/6*s2*s1 + 1/6*s1*s2 + 1/6*s1*s2*s1

```

5.23 Schur algebras for GL_n

This file implements:

- Schur algebras for GL_n over an arbitrary field.
- The canonical action of the Schur algebra on a tensor power of the standard representation.

- Using the above to calculate the characters of irreducible GL_n modules.

AUTHORS:

- Eric Webster (2010-07-01): implement Schur algebra
- Hugh Thomas (2011-05-08): implement action of Schur algebra and characters of irreducible modules

`sage.algebras.schur_algebra.GL_irreducible_character(n, mu, KK)`

Return the character of the irreducible module indexed by `mu` of $GL(n)$ over the field `KK`.

INPUT:

- `n` – positive integer
- `mu` – a partition of at most n parts
- `KK` – a field

OUTPUT: a symmetric function which should be interpreted in n variables to be meaningful as a character

EXAMPLES:

Over $\mathbf{Q}$, the irreducible character for μ is the Schur function associated to μ , plus garbage terms (Schur functions associated to partitions with more than n parts):

```
sage: from sage.algebras.schur_algebra import GL_irreducible_character
sage: sbasis = SymmetricFunctions(QQ).s()
sage: z = GL_irreducible_character(2, [2], QQ)
sage: sbasis(z)
s[2]

sage: z = GL_irreducible_character(4, [3, 2], QQ)
sage: sbasis(z)
-5*s[1, 1, 1, 1, 1] + s[3, 2]
```

```
>>> from sage.all import *
>>> from sage.algebras.schur_algebra import GL_irreducible_character
>>> sbasis = SymmetricFunctions(QQ).s()
>>> z = GL_irreducible_character(Integer(2), [Integer(2)], QQ)
>>> sbasis(z)
s[2]

>>> z = GL_irreducible_character(Integer(4), [Integer(3), Integer(2)], QQ)
>>> sbasis(z)
-5*s[1, 1, 1, 1, 1] + s[3, 2]
```

Over a Galois field, the irreducible character for μ will in general be smaller.

In characteristic p , for a one-part partition (r) , where $r = a_0 + pa_1 + p^2a_2 + \dots$, the result is (see [Gr2007], after 5.5d) the product of $h[a_0], h[a_1](pbasis[p]), h[a_2](pbasis[p^2]), \dots$, which is consistent with the following

```
sage: from sage.algebras.schur_algebra import GL_irreducible_character
sage: GL_irreducible_character(2, [7], GF(3))
m[4, 3] + m[6, 1] + m[7]
```

```
>>> from sage.all import *
>>> from sage.algebras.schur_algebra import GL_irreducible_character
```

(continues on next page)

(continued from previous page)

```
>>> GL_irreducible_character(Integer(2), [Integer(7)], GF(Integer(3)))
m[4, 3] + m[6, 1] + m[7]
```

class sage.algebras.schur_algebra.SchurAlgebra(R, n, r)

Bases: CombinatorialFreeModule

A Schur algebra.

Let R be a commutative ring, n be a positive integer, and r be a nonnegative integer. Define $A_R(n, r)$ to be the set of homogeneous polynomials of degree r in n^2 variables x_{ij} . Therefore we can write $R[x_{ij}] = \bigoplus_{r \geq 0} A_R(n, r)$, and $R[x_{ij}]$ is known to be a bialgebra with coproduct given by $\Delta(x_{ij}) = \sum_l x_{il} \otimes x_{lj}$ and counit $\varepsilon(x_{ij}) = \delta_{ij}$. Therefore $A_R(n, r)$ is a subcoalgebra of $R[x_{ij}]$. The Schur algebra $S_R(n, r)$ is the linear dual to $A_R(n, r)$, that is $S_R(n, r) := \text{hom}(A_R(n, r), R)$, and $S_R(n, r)$ obtains its algebra structure naturally by dualizing the comultiplication of $A_R(n, r)$.

Let $V = R^n$. One of the most important properties of the Schur algebra $S_R(n, r)$ is that it is isomorphic to the endomorphisms of $V^{\otimes r}$ which commute with the natural action of S_r .

EXAMPLES:

```
sage: S = SchurAlgebra(ZZ, 2, 2); S
Schur algebra (2, 2) over Integer Ring
```

```
>>> from sage.all import *
>>> S = SchurAlgebra(ZZ, Integer(2), Integer(2)); S
Schur algebra (2, 2) over Integer Ring
```

REFERENCES:

- [Gr2007]
- [Wikipedia article Schur_algebra](#)

dimension()

Return the dimension of `self`.

The dimension of the Schur algebra $S_R(n, r)$ is

$$\dim S_R(n, r) = \binom{n^2 + r - 1}{r}.$$

EXAMPLES:

```
sage: S = SchurAlgebra(QQ, 4, 2)
sage: S.dimension()
136
sage: S = SchurAlgebra(QQ, 2, 4)
sage: S.dimension()
35
```

```
>>> from sage.all import *
>>> S = SchurAlgebra(QQ, Integer(4), Integer(2))
>>> S.dimension()
136
>>> S = SchurAlgebra(QQ, Integer(2), Integer(4))
>>> S.dimension()
35
```

one()Return the element 1 of `self`.

EXAMPLES:

```

sage: S = SchurAlgebra(ZZ, 2, 2)
sage: e = S.one(); e
S((1, 1), (1, 1)) + S((1, 2), (1, 2)) + S((2, 2), (2, 2))

sage: x = S.an_element()
sage: x * e == x
True
sage: all(e * x == x for x in S.basis())
True

sage: S = SchurAlgebra(ZZ, 4, 4)
sage: e = S.one()
sage: x = S.an_element()
sage: x * e == x
True

```

```

>>> from sage.all import *
>>> S = SchurAlgebra(ZZ, Integer(2), Integer(2))
>>> e = S.one(); e
S((1, 1), (1, 1)) + S((1, 2), (1, 2)) + S((2, 2), (2, 2))

>>> x = S.an_element()
>>> x * e == x
True
>>> all(e * x == x for x in S.basis())
True

>>> S = SchurAlgebra(ZZ, Integer(4), Integer(4))
>>> e = S.one()
>>> x = S.an_element()
>>> x * e == x
True

```

product_on_basis(*e_{ij}*, *e_{kl}*)

Return the product of basis elements.

EXAMPLES:

```

sage: S = SchurAlgebra(QQ, 2, 3)
sage: B = S.basis()

```

```

>>> from sage.all import *
>>> S = SchurAlgebra(QQ, Integer(2), Integer(3))
>>> B = S.basis()

```

If we multiply two basis elements x and y , such that $x[1]$ and $y[0]$ are not permutations of each other, the result is zero:

```
sage: S.product_on_basis(((1, 1, 1), (1, 1, 2)), ((1, 2, 2), (1, 1, 2)))
0
```

```
>>> from sage.all import *
>>> S.product_on_basis(((Integer(1), Integer(1), Integer(1)), (Integer(1), Integer(1), Integer(2))), ((Integer(1), Integer(2), Integer(2)), (Integer(1), Integer(1), Integer(2))))
0
```

If we multiply a basis element x by a basis element which consists of the same tuple repeated twice (on either side), the result is either zero (if the previous case applies) or x :

```
sage: ww = B[((1, 2, 2), (1, 2, 2))]
sage: x = B[((1, 2, 2), (1, 1, 2))]
sage: ww * x
S((1, 2, 2), (1, 1, 2))
```

```
>>> from sage.all import *
>>> ww = B[((Integer(1), Integer(2), Integer(2)), (Integer(1), Integer(2), Integer(2)))]
>>> x = B[((Integer(1), Integer(2), Integer(2)), (Integer(1), Integer(1), Integer(2)))]
>>> ww * x
S((1, 2, 2), (1, 1, 2))
```

An arbitrary product, on the other hand, may have multiplicities:

```
sage: x = B[((1, 1, 1), (1, 1, 2))]
sage: y = B[((1, 1, 2), (1, 2, 2))]
sage: x * y
2*S((1, 1, 1), (1, 2, 2))
```

```
>>> from sage.all import *
>>> x = B[((Integer(1), Integer(1), Integer(1)), (Integer(1), Integer(1), Integer(2)))]
>>> y = B[((Integer(1), Integer(1), Integer(2)), (Integer(1), Integer(2), Integer(2)))]
>>> x * y
2*S((1, 1, 1), (1, 2, 2))
```

```
class sage.algebras.schur_algebra.SchurTensorModule(R, n, r)
```

Bases: CombinatorialFreeModule_Tensor

The space $V^{\otimes r}$ where $V = R^n$ equipped with a left action of the Schur algebra $S_R(n, r)$ and a right action of the symmetric group S_r .

Let R be a commutative ring and $V = R^n$. We consider the module $V^{\otimes r}$ equipped with a natural right action of the symmetric group S_r given by

$$(v_1 \otimes v_2 \otimes \cdots \otimes v_n)\sigma = v_{\sigma(1)} \otimes v_{\sigma(2)} \otimes \cdots \otimes v_{\sigma(n)}.$$

The Schur algebra $S_R(n, r)$ is naturally isomorphic to the endomorphisms of $V^{\otimes r}$ which commutes with the S_r action. We get the natural left action of $S_R(n, r)$ by this isomorphism.

EXAMPLES:

```

sage: T = SchurTensorModule(QQ, 2, 3); T
The 3-fold tensor product of a free module of dimension 2
over Rational Field
sage: A = SchurAlgebra(QQ, 2, 3)
sage: P = Permutations(3)
sage: t = T.an_element(); t
2*B[1] # B[1] # B[1] + 2*B[1] # B[1] # B[2] + 3*B[1] # B[2] # B[1]
sage: a = A.an_element(); a
2*S((1, 1, 1), (1, 1, 1)) + 2*S((1, 1, 1), (1, 1, 2))
+ 3*S((1, 1, 1), (1, 2, 2))
sage: p = P.an_element(); p
[3, 1, 2]
sage: y = a * t; y
14*B[1] # B[1] # B[1]
sage: y * p
14*B[1] # B[1] # B[1]
sage: z = t * p; z
2*B[1] # B[1] # B[1] + 3*B[1] # B[1] # B[2] + 2*B[2] # B[1] # B[1]
sage: a * z
14*B[1] # B[1] # B[1]

```

```

>>> from sage.all import *
>>> T = SchurTensorModule(QQ, Integer(2), Integer(3)); T
The 3-fold tensor product of a free module of dimension 2
over Rational Field
>>> A = SchurAlgebra(QQ, Integer(2), Integer(3))
>>> P = Permutations(Integer(3))
>>> t = T.an_element(); t
2*B[1] # B[1] # B[1] + 2*B[1] # B[1] # B[2] + 3*B[1] # B[2] # B[1]
>>> a = A.an_element(); a
2*S((1, 1, 1), (1, 1, 1)) + 2*S((1, 1, 1), (1, 1, 2))
+ 3*S((1, 1, 1), (1, 2, 2))
>>> p = P.an_element(); p
[3, 1, 2]
>>> y = a * t; y
14*B[1] # B[1] # B[1]
>>> y * p
14*B[1] # B[1] # B[1]
>>> z = t * p; z
2*B[1] # B[1] # B[1] + 3*B[1] # B[1] # B[2] + 2*B[2] # B[1] # B[1]
>>> a * z
14*B[1] # B[1] # B[1]

```

We check the commuting action property:

```

sage: all( (bA * bT) * p == bA * (bT * p)
.....:      for bT in T.basis() for bA in A.basis() for p in P)
True

```

```

>>> from sage.all import *
>>> all( (bA * bT) * p == bA * (bT * p)
...      for bT in T.basis() for bA in A.basis() for p in P)
True

```

class `Element`

Bases: `IndexedFreeModuleElement`

construction()

Return `None`.

There is no functorial construction for `self`.

EXAMPLES:

```
sage: T = SchurTensorModule(QQ, 2, 3)
sage: T.construction()
```

```
>>> from sage.all import *
>>> T = SchurTensorModule(QQ, Integer(2), Integer(3))
>>> T.construction()
```

`sage.algebras.schur_algebra.schur_representative_from_index(i0, i1)`

Simultaneously reorder a pair of tuples to obtain the equivalent element of the distinguished basis of the Schur algebra.

See also

`schur_representative_indices()`

INPUT:

- A pair of tuples of length r with elements in $\{1, \dots, n\}$

OUTPUT: the corresponding pair of tuples ordered correctly

EXAMPLES:

```
sage: from sage.algebras.schur_algebra import schur_representative_from_index
sage: schur_representative_from_index([2, 1, 2, 2], [1, 3, 0, 0])
((1, 2, 2, 2), (3, 0, 0, 1))
```

```
>>> from sage.all import *
>>> from sage.algebras.schur_algebra import schur_representative_from_index
>>> schur_representative_from_index([Integer(2), Integer(1), Integer(2), Integer(2)],
→ [Integer(1), Integer(3), Integer(0), Integer(0)])
((1, 2, 2, 2), (3, 0, 0, 1))
```

`sage.algebras.schur_algebra.schur_representative_indices(n, r)`

Return a set which functions as a basis of $S_K(n, r)$.

More specifically, the basis for $S_K(n, r)$ consists of equivalence classes of pairs of tuples of length r on the alphabet $\{1, \dots, n\}$, where the equivalence relation is simultaneous permutation of the two tuples. We can therefore fix a representative for each equivalence class in which the entries of the first tuple weakly increase, and the entries of the second tuple whose corresponding values in the first tuple are equal, also weakly increase.

EXAMPLES:

```
sage: from sage.algebras.schur_algebra import schur_representative_indices
sage: schur_representative_indices(2, 2)
```

(continues on next page)

(continued from previous page)

```
[((1, 1), (1, 1)), ((1, 1), (1, 2)),
 ((1, 1), (2, 2)), ((1, 2), (1, 1)),
 ((1, 2), (1, 2)), ((1, 2), (2, 1)),
 ((1, 2), (2, 2)), ((2, 2), (1, 1)),
 ((2, 2), (1, 2)), ((2, 2), (2, 2))]
```

```
>>> from sage.all import *
>>> from sage.algebras.schur_algebra import schur_representative_indices
>>> schur_representative_indices(Integer(2), Integer(2))
[((1, 1), (1, 1)), ((1, 1), (1, 2)),
 ((1, 1), (2, 2)), ((1, 2), (1, 1)),
 ((1, 2), (1, 2)), ((1, 2), (2, 1)),
 ((1, 2), (2, 2)), ((2, 2), (1, 1)),
 ((2, 2), (1, 2)), ((2, 2), (2, 2))]
```

5.24 The Steenrod algebra

AUTHORS:

- John H. Palmieri (2008-07-30): version 0.9: Initial implementation.
- John H. Palmieri (2010-06-30): version 1.0: Implemented sub-Hopf algebras and profile functions; direct multiplication of admissible sequences (rather than conversion to the Milnor basis); implemented the Steenrod algebra using CombinatorialFreeModule; improved the test suite.

This module defines the mod p Steenrod algebra $\mathcal{A}_p$, some of its properties, and ways to define elements of it.

From a topological point of view, $\mathcal{A}_p$ is the algebra of stable cohomology operations on mod p cohomology; thus for any topological space X , its mod p cohomology algebra $H^*(X, \mathbf{F}_p)$ is a module over $\mathcal{A}_p$.

From an algebraic point of view, $\mathcal{A}_p$ is an $\mathbf{F}_p$ -algebra; when $p = 2$, it is generated by elements Sq^i for $i \geq 0$ (the *Steenrod squares*), and when p is odd, it is generated by elements $\mathcal{P}^i$ for $i \geq 0$ (the *Steenrod reduced p -th powers*) along with an element β (the *mod p Bockstein*). The Steenrod algebra is graded: Sq^i is in degree i for each i , β is in degree 1, and $\mathcal{P}^i$ is in degree $2(p-1)i$.

The unit element is Sq^0 when $p = 2$ and $\mathcal{P}^0$ when p is odd. The generating elements also satisfy the *Adem relations*. At the prime 2, these have the form

$$\text{Sq}^a \text{Sq}^b = \sum_{c=0}^{\lfloor a/2 \rfloor} \binom{b-c-1}{a-2c} \text{Sq}^{a+b-c} \text{Sq}^c.$$

At odd primes, they are a bit more complicated; see Steenrod and Epstein [SE1962] or `sage.algebras.steenrod.steenrod_algebra_bases` for full details. These relations lead to the existence of the *Serre-Cartan* basis for $\mathcal{A}_p$.

The mod p Steenrod algebra has the structure of a Hopf algebra, and Milnor [Mil1958] has a beautiful description of the dual, leading to a construction of the *Milnor basis* for $\mathcal{A}_p$. In this module, elements in the Steenrod algebra are represented, by default, using the Milnor basis.

Bases for the Steenrod algebra

There are a handful of other bases studied in the literature; the paper by Monks [Mon1998] is a good reference. Here is a quick summary:

- The *Milnor basis*. When $p = 2$, the Milnor basis consists of symbols of the form $\text{Sq}(m_1, m_2, \dots, m_t)$, where each m_i is a nonnegative integer and if $t > 1$, then the last entry $m_t > 0$. When p is odd, the Milnor basis consists of

symbols of the form $Q_{e_1}Q_{e_2}\dots\mathcal{P}(m_1, m_2, \dots, m_t)$, where $0 \leq e_1 < e_2 < \dots$, each m_i is a nonnegative integer, and if $t > 1$, then the last entry $m_t > 0$.

When $p = 2$, it can be convenient to use the notation $\mathcal{P}(-)$ to mean $\text{Sq}(-)$, so that there is consistent notation for all primes.

- *The Serre-Cartan basis.* This basis consists of ‘admissible monomials’ in the Steenrod operations. Thus at the prime 2, it consists of monomials $\text{Sq}^{m_1}\text{Sq}^{m_2}\dots\text{Sq}^{m_t}$ with $m_i \geq 2m_{i+1}$ for each i . At odd primes, this basis consists of monomials $\beta^{\epsilon_0}\mathcal{P}^{s_1}\beta^{\epsilon_1}\mathcal{P}^{s_2}\dots\mathcal{P}^{s_k}\beta^{\epsilon_k}$ with each ϵ_i either 0 or 1, $s_i \geq ps_{i+1} + \epsilon_i$, and $s_k \geq 1$.

Most of the rest of the bases are only defined when $p = 2$. The only exceptions are the P_t^s -bases and the commutator bases, which are defined at all primes.

- *Wood’s Y basis.* For pairs of nonnegative integers (m, k) , let $w(m, k) = \text{Sq}^{2^m(2^{k+1}-1)}$. Wood’s Y basis consists of monomials $w(m_0, k_0)\dots w(m_t, k_t)$ with $(m_i, k_i) > (m_{i+1}, k_{i+1})$, in left lex order.
- *Wood’s Z basis.* For pairs of nonnegative integers (m, k) , let $w(m, k) = \text{Sq}^{2^m(2^{k+1}-1)}$. Wood’s Z basis consists of monomials $w(m_0, k_0)\dots w(m_t, k_t)$ with $(m_i + k_i, m_i) > (m_{i+1} + k_{i+1}, m_{i+1})$, in left lex order.
- *Wall’s basis.* For any pair of integers (m, k) with $m \geq k \geq 0$, let $Q_k^m = \text{Sq}^{2^k}\text{Sq}^{2^{k+1}}\dots\text{Sq}^{2^m}$. The elements of Wall’s basis are monomials $Q_{k_0}^{m_0}\dots Q_{k_t}^{m_t}$ with $(m_i, k_i) > (m_{i+1}, k_{i+1})$, ordered left lexicographically.

(Note that Q_k^m is the reverse of the element X_k^m used in defining Arnon’s A basis.)

- *Arnon’s A basis.* For any pair of integers (m, k) with $m \geq k \geq 0$, let $X_k^m = \text{Sq}^{2^m}\text{Sq}^{2^{m-1}}\dots\text{Sq}^{2^k}$. The elements of Arnon’s A basis are monomials $X_{k_0}^{m_0}\dots X_{k_t}^{m_t}$ with $(m_i, k_i) < (m_{i+1}, k_{i+1})$, ordered left lexicographically.

(Note that X_k^m is the reverse of the element Q_k^m used in defining Wall’s basis.)

- *Arnon’s C basis.* The elements of Arnon’s C basis are monomials of the form $\text{Sq}^{t_1}\dots\text{Sq}^{t_m}$ where for each i , we have $t_i \leq 2t_{i+1}$ and $2^i | t_{m-i}$.
- *P_t^s bases.* Let $p = 2$. For integers $s \geq 0$ and $t > 0$, the element P_t^s is the Milnor basis element $\mathcal{P}(0, \dots, 0, p^s, 0, \dots)$, with the nonzero entry in position t . To obtain a P_t^s -basis, for each set $\{P_{t_1}^{s_1}, \dots, P_{t_k}^{s_k}\}$ of (distinct) P_t^s ’s, one chooses an ordering and forms the monomials

$$(P_{t_1}^{s_1})^{i_1}\dots(P_{t_k}^{s_k})^{i_k}$$

for all exponents i_j with $0 < i_j < p$. When $p = 2$, the set of all such monomials then forms a basis, and when p is odd, if one multiplies each such monomial on the left by products of the form $Q_{e_1}Q_{e_2}\dots$ with $0 \leq e_1 < e_2 < \dots$, one obtains a basis.

Thus one gets a basis by choosing an ordering on each set of P_t^s ’s. There are infinitely many orderings possible, and we have implemented four of them:

- ‘rlex’: right lexicographic ordering
- ‘llex’: left lexicographic ordering
- ‘deg’: ordered by degree, which is the same as left lexicographic ordering on the pair $(s + t, t)$
- ‘revz’: left lexicographic ordering on the pair $(s + t, s)$, which is the reverse of the ordering used (on elements in the same degrees as the P_t^s ’s) in Wood’s Z basis: ‘revz’ stands for ‘reversed Z’. This is the default: ‘pst’ is the same as ‘pst_revz’.
- *Commutator bases.* Let $c_{i,1} = \mathcal{P}(p^i)$, let $c_{i,2} = [c_{i+1,1}, c_{i,1}]$, and inductively define $c_{i,k} = [c_{i+k-1,1}, c_{i,k-1}]$. Thus $c_{i,k}$ is a k -fold iterated commutator of the elements $\mathcal{P}(p^i), \dots, \mathcal{P}(p^{i+k-1})$. Note that $\dim c_{i,k} = \dim P_k^i$.

Commutator bases are obtained in much the same way as P_t^s -bases: for each set $\{c_{s_1,t_1}, \dots, c_{s_k,t_k}\}$ of (distinct) $c_{s,t}$ ’s, one chooses an ordering and forms the resulting monomials

$$c_{s_1,t_1}^{i_1}\dots c_{s_k,t_k}^{i_k}$$

for all exponents i_j with $0 < i_j < p$. When p is odd, one also needs to left-multiply by products of the Q_i 's. As for P_t^s -bases, every ordering on each set of iterated commutators determines a basis, and the same four orderings have been defined for these bases as for the P_t^s bases: 'rlex', 'llex', 'deg', 'revz'.

Sub-Hopf algebras of the Steenrod algebra

The sub-Hopf algebras of the Steenrod algebra have been classified. Milnor proved that at the prime 2, the dual of the Steenrod algebra A_* is isomorphic to a polynomial algebra

$$A_* \cong \mathbf{F}_2[\xi_1, \xi_2, \xi_3, \dots].$$

The Milnor basis is dual to the monomial basis. Furthermore, any sub-Hopf algebra corresponds to a quotient of this of the form

$$A_*/(\xi_1^{2^{e_1}}, \xi_2^{2^{e_2}}, \xi_3^{2^{e_3}}, \dots).$$

The list of exponents $(e_1, e_2, \dots)$ may be considered a function e from the positive integers to the extended nonnegative integers (the nonnegative integers and ∞); this is called the *profile function* for the sub-Hopf algebra. The profile function must satisfy the condition

- $e(r) \geq \min(e(r-i) - i, e(i))$ for all $0 < i < r$.

At odd primes, the situation is similar: the dual is isomorphic to the tensor product of a polynomial algebra and an exterior algebra,

$$A_* = \mathbf{F}_p[\xi_1, \xi_2, \xi_3, \dots] \otimes \Lambda(\tau_0, \tau_1, \dots),$$

and any sub-Hopf algebra corresponds to a quotient of this of the form

$$A_*/(\xi_1^{p^{e_1}}, \xi_2^{p^{e_2}}, \dots; \tau_0^{k_0}, \tau_1^{k_1}, \dots).$$

Here the profile function has two pieces, e as at the prime 2, and k , which maps the nonnegative integers to the set $\{1, 2\}$. These must satisfy the following conditions:

- $e(r) \geq \min(e(r-i) - i, e(i))$ for all $0 < i < r$.
- if $k(i+j) = 1$, then either $e(i) \leq j$ or $k(j) = 1$ for all $i \geq 1, j \geq 0$.

(See Adams-Margolis [AM1974], for example, for these results on profile functions.)

This module allows one to construct the Steenrod algebra or any of its sub-Hopf algebras, at any prime. When defining a sub-Hopf algebra, you must work with the Milnor basis or a P_t^s -basis.

Elements of the Steenrod algebra

Basic arithmetic, $p = 2$. To construct an element of the mod 2 Steenrod algebra, use the function `Sq`:

```
sage: a = Sq(1, 2)
sage: b = Sq(4, 1)
sage: z = a + b
sage: z
Sq(1, 2) + Sq(4, 1)
sage: Sq(4) * Sq(1, 2)
Sq(1, 1, 1) + Sq(2, 3) + Sq(5, 2)
sage: z**2          # nonnegative exponents work as they should
Sq(1, 2, 1) + Sq(4, 1, 1)
sage: z**0
1
```

```

>>> from sage.all import *
>>> a = Sq(Integer(1), Integer(2))
>>> b = Sq(Integer(4), Integer(1))
>>> z = a + b
>>> z
Sq(1, 2) + Sq(4, 1)
>>> Sq(Integer(4)) * Sq(Integer(1), Integer(2))
Sq(1, 1, 1) + Sq(2, 3) + Sq(5, 2)
>>> z**Integer(2)           # nonnegative exponents work as they should
Sq(1, 2, 1) + Sq(4, 1, 1)
>>> z**Integer(0)
1

```

Basic arithmetic, $p > 2$. To construct an element of the mod p Steenrod algebra when p is odd, you should first define a Steenrod algebra, using the `SteenrodAlgebra` command:

```
sage: A3 = SteenrodAlgebra(3)
```

```

>>> from sage.all import *
>>> A3 = SteenrodAlgebra(Integer(3))

```

Having done this, the newly created algebra `A3` has methods `Q` and `P` which construct elements of `A3`:

```

sage: c = A3.Q(1, 3, 6); c
Q_1 Q_3 Q_6
sage: d = A3.P(2, 0, 1); d
P(2, 0, 1)
sage: c * d
Q_1 Q_3 Q_6 P(2, 0, 1)
sage: e = A3.P(3)
sage: d * e
P(5, 0, 1)
sage: e * d
P(1, 1, 1) + P(5, 0, 1)
sage: c * c
0
sage: e ** 3
2 P(1, 2)

```

```

>>> from sage.all import *
>>> c = A3.Q(Integer(1), Integer(3), Integer(6)); c
Q_1 Q_3 Q_6
>>> d = A3.P(Integer(2), Integer(0), Integer(1)); d
P(2, 0, 1)
>>> c * d
Q_1 Q_3 Q_6 P(2, 0, 1)
>>> e = A3.P(Integer(3))
>>> d * e
P(5, 0, 1)
>>> e * d
P(1, 1, 1) + P(5, 0, 1)
>>> c * c
0

```

(continues on next page)

(continued from previous page)

```
>>> e ** Integer(3)
2 P(1,2)
```

Note that one can construct an element like `c` above in one step, without first constructing the algebra:

```
sage: c = SteenrodAlgebra(3).Q(1,3,6)
sage: c
Q_1 Q_3 Q_6
```

```
>>> from sage.all import *
>>> c = SteenrodAlgebra(Integer(3)).Q(Integer(1), Integer(3), Integer(6))
>>> c
Q_1 Q_3 Q_6
```

And of course, you can do similar constructions with the mod 2 Steenrod algebra:

```
sage: A = SteenrodAlgebra(2); A
mod 2 Steenrod algebra, milnor basis
sage: A.Sq(2,3,5)
Sq(2,3,5)
sage: A.P(2,3,5)    # when p=2, P = Sq
Sq(2,3,5)
sage: A.Q(1,4)      # when p=2, this gives a product of Milnor primitives
Sq(0,1,0,0,1)
```

```
>>> from sage.all import *
>>> A = SteenrodAlgebra(Integer(2)); A
mod 2 Steenrod algebra, milnor basis
>>> A.Sq(Integer(2), Integer(3), Integer(5))
Sq(2,3,5)
>>> A.P(Integer(2), Integer(3), Integer(5))    # when p=2, P = Sq
Sq(2,3,5)
>>> A.Q(Integer(1), Integer(4))                # when p=2, this gives a product of Milnor
↳primitives
Sq(0,1,0,0,1)
```

Associated to each element is its prime (the characteristic of the underlying base field) and its basis (the basis for the Steenrod algebra in which it lies):

```
sage: a = SteenrodAlgebra(basis='milnor').Sq(1,2,1)
sage: a.prime()
2
sage: a.basis_name()
'milnor'
sage: a.degree()
14
```

```
>>> from sage.all import *
>>> a = SteenrodAlgebra(basis='milnor').Sq(Integer(1), Integer(2), Integer(1))
>>> a.prime()
2
>>> a.basis_name()
```

(continues on next page)

(continued from previous page)

```
'milnor'
>>> a.degree()
14
```

It can be viewed in other bases:

```
sage: a.milnor() # same as a
Sq(1,2,1)
sage: a.change_basis('adem')
Sq^9 Sq^4 Sq^1 + Sq^11 Sq^2 Sq^1 + Sq^13 Sq^1
sage: a.change_basis('adem').change_basis('milnor')
Sq(1,2,1)
```

```
>>> from sage.all import *
>>> a.milnor() # same as a
Sq(1,2,1)
>>> a.change_basis('adem')
Sq^9 Sq^4 Sq^1 + Sq^11 Sq^2 Sq^1 + Sq^13 Sq^1
>>> a.change_basis('adem').change_basis('milnor')
Sq(1,2,1)
```

Regardless of the prime, each element has an excess, and if the element is homogeneous, a degree. The excess of $Sq(i_1, i_2, i_3, \dots)$ is $i_1 + i_2 + i_3 + \dots$; when p is odd, the excess of $Q_0^{e_0} Q_1^{e_1} \dots \mathcal{P}(r_1, r_2, \dots)$ is $\sum e_i + 2 \sum r_i$. The excess of a linear combination of Milnor basis elements is the minimum of the excesses of those basis elements.

The degree of $Sq(i_1, i_2, i_3, \dots)$ is $\sum (2^n - 1) i_n$, and when p is odd, the degree of $Q_0^{e_0} Q_1^{e_1} \dots \mathcal{P}(r_1, r_2, \dots)$ is $\sum \epsilon_i (2p^i - 1) + \sum r_j (2p^j - 2)$. The degree of a linear combination of such terms is only defined if the terms all have the same degree.

Here are some simple examples:

```
sage: z = Sq(1,2) + Sq(4,1)
sage: z.degree()
7
sage: (Sq(0,0,1) + Sq(5,3)).degree()
Traceback (most recent call last):
...
ValueError: element is not homogeneous
sage: Sq(7,2,1).excess()
10
sage: z.excess()
3
sage: B = SteenrodAlgebra(3)
sage: x = B.Q(1,4)
sage: y = B.P(1,2,3)
sage: x.degree()
166
sage: x.excess()
2
sage: y.excess()
12
```

```

>>> from sage.all import *
>>> z = Sq(Integer(1), Integer(2)) + Sq(Integer(4), Integer(1))
>>> z.degree()
7
>>> (Sq(Integer(0), Integer(0), Integer(1)) + Sq(Integer(5), Integer(3))).degree()
Traceback (most recent call last):
...
ValueError: element is not homogeneous
>>> Sq(Integer(7), Integer(2), Integer(1)).excess()
10
>>> z.excess()
3
>>> B = SteenrodAlgebra(Integer(3))
>>> x = B.Q(Integer(1), Integer(4))
>>> y = B.P(Integer(1), Integer(2), Integer(3))
>>> x.degree()
166
>>> x.excess()
2
>>> y.excess()
12

```

Elements have a weight in the May filtration, which (when $p = 2$) is related to the height function defined by Wall:

```

sage: Sq(2, 1, 5).may_weight()
9
sage: Sq(2, 1, 5).wall_height()
[2, 3, 2, 1, 1]
sage: b = Sq(4)*Sq(8) + Sq(8)*Sq(4)
sage: b.may_weight()
2
sage: b.wall_height()
[0, 0, 1, 1]

```

```

>>> from sage.all import *
>>> Sq(Integer(2), Integer(1), Integer(5)).may_weight()
9
>>> Sq(Integer(2), Integer(1), Integer(5)).wall_height()
[2, 3, 2, 1, 1]
>>> b = Sq(Integer(4))*Sq(Integer(8)) + Sq(Integer(8))*Sq(Integer(4))
>>> b.may_weight()
2
>>> b.wall_height()
[0, 0, 1, 1]

```

Odd primary May weights:

```

sage: A5 = SteenrodAlgebra(5)
sage: a = A5.Q(1, 2, 4)
sage: b = A5.P(1, 2, 1)
sage: a.may_weight()
10
sage: b.may_weight()

```

(continues on next page)

(continued from previous page)

```

8
sage: (a * b).may_weight()
18
sage: A5.P(0,0,1).may_weight()
3

```

```

>>> from sage.all import *
>>> A5 = SteenrodAlgebra(Integer(5))
>>> a = A5.Q(Integer(1), Integer(2), Integer(4))
>>> b = A5.P(Integer(1), Integer(2), Integer(1))
>>> a.may_weight()
10
>>> b.may_weight()
8
>>> (a * b).may_weight()
18
>>> A5.P(Integer(0), Integer(0), Integer(1)).may_weight()
3

```

Since the Steenrod algebra is a Hopf algebra, every element has a coproduct and an antipode:

```

sage: Sq(5).coproduct()
1 # Sq(5) + Sq(1) # Sq(4) + Sq(2) # Sq(3) + Sq(3) # Sq(2) + Sq(4) # Sq(1) + Sq(5) # 1
sage: Sq(5).antipode()
Sq(2,1) + Sq(5)
sage: d = Sq(0,0,1); d
Sq(0,0,1)
sage: d.antipode()
Sq(0,0,1)
sage: Sq(4).antipode()
Sq(1,1) + Sq(4)
sage: (Sq(4) * Sq(2)).antipode()
Sq(6)
sage: SteenrodAlgebra(7).P(3,1).antipode()
P(3,1)

```

```

>>> from sage.all import *
>>> Sq(Integer(5)).coproduct()
1 # Sq(5) + Sq(1) # Sq(4) + Sq(2) # Sq(3) + Sq(3) # Sq(2) + Sq(4) # Sq(1) + Sq(5) # 1
>>> Sq(Integer(5)).antipode()
Sq(2,1) + Sq(5)
>>> d = Sq(Integer(0), Integer(0), Integer(1)); d
Sq(0,0,1)
>>> d.antipode()
Sq(0,0,1)
>>> Sq(Integer(4)).antipode()
Sq(1,1) + Sq(4)
>>> (Sq(Integer(4)) * Sq(Integer(2))).antipode()
Sq(6)
>>> SteenrodAlgebra(Integer(7)).P(Integer(3), Integer(1)).antipode()
P(3,1)

```


Applying the antipode twice returns the original element:

```
sage: y = Sq(8)*Sq(4)
sage: y == (y.antipode()).antipode()
True
```

```
>>> from sage.all import *
>>> y = Sq(Integer(8))*Sq(Integer(4))
>>> y == (y.antipode()).antipode()
True
```

Internal representation: you can use any element as an iterator (for x in a: ...), and the method `monomial_coefficients()` returns a dictionary with keys tuples representing basis elements and with corresponding value representing the coefficient of that term:

```
sage: c = Sq(5).antipode(); c
Sq(2,1) + Sq(5)
sage: for mono, coeff in c: print((coeff, mono))
(1, (5,))
(1, (2, 1))
sage: c.monomial_coefficients() == {(2, 1): 1, (5,): 1}
True
sage: sorted(c.monomials(), key=lambda x: tuple(x.support()))
[Sq(2,1), Sq(5)]
sage: sorted(c.support())
[(2, 1), (5,)]
sage: Adem = SteenrodAlgebra(basis='adem')
sage: elt = Adem.Sq(10) + Adem.Sq(9) * Adem.Sq(1)
sage: sorted(elt.monomials(), key=lambda x: tuple(x.support()))
[Sq^9 Sq^1, Sq^10]

sage: A7 = SteenrodAlgebra(p=7)
sage: a = A7.P(1) * A7.P(1); a
2 P(2)
sage: a.leading_coefficient()
2
sage: a.leading_monomial()
P(2)
sage: a.leading_term()
2 P(2)
sage: a.change_basis('adem').monomial_coefficients()
{(0, 2, 0): 2}
```

```
>>> from sage.all import *
>>> c = Sq(Integer(5)).antipode(); c
Sq(2,1) + Sq(5)
>>> for mono, coeff in c: print((coeff, mono))
(1, (5,))
(1, (2, 1))
>>> c.monomial_coefficients() == {(Integer(2), Integer(1)): Integer(1), (Integer(5),
↪): Integer(1)}
True
>>> sorted(c.monomials(), key=lambda x: tuple(x.support()))
```

(continues on next page)

(continued from previous page)

```
[Sq(2,1), Sq(5)]
>>> sorted(c.support())
[(2, 1), (5,)]
>>> Adem = SteenrodAlgebra(basis='adem')
>>> elt = Adem.Sq(Integer(10)) + Adem.Sq(Integer(9)) * Adem.Sq(Integer(1))
>>> sorted(elt.monomials(), key=lambda x: tuple(x.support()))
[Sq^9 Sq^1, Sq^10]

>>> A7 = SteenrodAlgebra(p=Integer(7))
>>> a = A7.P(Integer(1)) * A7.P(Integer(1)); a
2 P(2)
>>> a.leading_coefficient()
2
>>> a.leading_monomial()
P(2)
>>> a.leading_term()
2 P(2)
>>> a.change_basis('adem').monomial_coefficients()
{(0, 2, 0): 2}
```

The tuple in the previous output stands for the element $\beta^0 P^2 \beta^0$, i.e., P^2 . Going in the other direction, if you want to specify a basis element by giving the corresponding tuple, you can use the `monomial()` method on the algebra:

```
sage: SteenrodAlgebra(p=7, basis='adem').monomial((0, 2, 0))
P^2
sage: 10 * SteenrodAlgebra(p=7, basis='adem').monomial((0, 2, 0))
3 P^2
```

```
>>> from sage.all import *
>>> SteenrodAlgebra(p=Integer(7), basis='adem').monomial((Integer(0), Integer(2),
↳ Integer(0)))
P^2
>>> Integer(10) * SteenrodAlgebra(p=Integer(7), basis='adem').monomial((Integer(0),
↳ Integer(2), Integer(0)))
3 P^2
```

In the following example, elements in Wood's Z basis are certain products of the elements $w(m, k) = \text{Sq}^{2^m(2^{k+1}-1)}$. Internally, each $w(m, k)$ is represented by the pair (m, k) , and products of them are represented by tuples of such pairs.

```
sage: A = SteenrodAlgebra(basis='wood_z')
sage: t = ((2, 0), (0, 0))
sage: A.monomial(t)
Sq^4 Sq^1
```

```
>>> from sage.all import *
>>> A = SteenrodAlgebra(basis='wood_z')
>>> t = ((Integer(2), Integer(0)), (Integer(0), Integer(0)))
>>> A.monomial(t)
Sq^4 Sq^1
```

See the documentation for `SteenrodAlgebra()` for more details and examples.

`sage.algebras.steenrod.steenrod_algebra.AA(n=None, p=2)`

This returns the Steenrod algebra A or its sub-Hopf algebra $A(n)$.

INPUT:

- n – nonnegative integer (default: None)
- p – prime number (default: 2)

OUTPUT:

If n is None, then return the full Steenrod algebra. Otherwise, return $A(n)$.

When $p = 2$, $A(n)$ is the sub-Hopf algebra generated by the elements Sq^i for $i \leq 2^n$. Its profile function is $(n+1, n, n-1, \dots)$. When p is odd, $A(n)$ is the sub-Hopf algebra generated by the elements Q_0 and $\mathcal{P}^i$ for $i \leq p^{n-1}$. Its profile function is $e = (n, n-1, n-2, \dots)$ and $k = (2, 2, \dots, 2)$ (length $n+1$).

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra import AA as A
sage: A()
mod 2 Steenrod algebra, milnor basis
sage: A(2)
sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis,
profile function [3, 2, 1]
sage: A(2, p=5)
sub-Hopf algebra of mod 5 Steenrod algebra, milnor basis,
profile function ([2, 1], [2, 2, 2])
```

```
>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra import AA as A
>>> A()
mod 2 Steenrod algebra, milnor basis
>>> A(Integer(2))
sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis,
profile function [3, 2, 1]
>>> A(Integer(2), p=Integer(5))
sub-Hopf algebra of mod 5 Steenrod algebra, milnor basis,
profile function ([2, 1], [2, 2, 2])
```

`sage.algebras.steenrod.steenrod_algebra.Sq(*nums)`

Milnor element $Sq(a, b, c, \dots)$.

INPUT:

- $a, b, c, \dots$ – nonnegative integers

OUTPUT: element of the Steenrod algebra

This returns the Milnor basis element $Sq(a, b, c, \dots)$.

EXAMPLES:

```
sage: Sq(5)
Sq(5)
sage: Sq(5) + Sq(2,1) + Sq(5) # addition is mod 2:
Sq(2,1)
sage: (Sq(4,3) + Sq(7,2)).degree()
13
```

```

>>> from sage.all import *
>>> Sq(Integer(5))
Sq(5)
>>> Sq(Integer(5)) + Sq(Integer(2), Integer(1)) + Sq(Integer(5)) # addition is
↪mod 2:
Sq(2, 1)
>>> (Sq(Integer(4), Integer(3)) + Sq(Integer(7), Integer(2))).degree()
13

```

Entries must be nonnegative integers; otherwise, an error results.

This function is a good way to define elements of the Steenrod algebra.

```

sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra (p=2, basis='milnor', generic='auto',
**kws)

```

The mod p Steenrod algebra.

INPUT:

- p – positive prime integer (default: 2)
- *basis* – string (default: 'milnor')
- *profile* – a profile function in form specified below (default: None)
- *truncation_type* – 0 or ∞ or 'auto' (default: 'auto')
- *precision* – integer or None (default: None)
- *generic* – (default: 'auto')

OUTPUT:

mod p Steenrod algebra or one of its sub-Hopf algebras, elements of which are printed using *basis*

See below for information about *basis*, *profile*, etc.

EXAMPLES:

Some properties of the Steenrod algebra are available:

```

sage: A = SteenrodAlgebra(2)
sage: A.order()
+Infinity
sage: A.is_finite()
False
sage: A.is_commutative()
False
sage: A.is_noetherian()
False
sage: A.is_integral_domain()
False
sage: A.is_field()
False
sage: A.is_division_algebra()
False
sage: A.category()
Category of supercocommutative super Hopf algebras
with basis over Finite Field of size 2

```

```

>>> from sage.all import *
>>> A = SteenrodAlgebra(Integer(2))
>>> A.order()
+Infinity
>>> A.is_finite()
False
>>> A.is_commutative()
False
>>> A.is_noetherian()
False
>>> A.is_integral_domain()
False
>>> A.is_field()
False
>>> A.is_division_algebra()
False
>>> A.category()
Category of supercocommutative super Hopf algebras
with basis over Finite Field of size 2

```

There are methods for constructing elements of the Steenrod algebra:

```

sage: A2 = SteenrodAlgebra(2); A2
mod 2 Steenrod algebra, milnor basis
sage: A2.Sq(1,2,6)
Sq(1,2,6)
sage: A2.Q(3,4)  # product of Milnor primitives Q_3 and Q_4
Sq(0,0,0,1,1)
sage: A2.pst(2,3)  # Margolis pst element
Sq(0,0,4)
sage: A5 = SteenrodAlgebra(5); A5
mod 5 Steenrod algebra, milnor basis
sage: A5.P(1,2,6)
P(1,2,6)
sage: A5.Q(3,4)
Q_3 Q_4
sage: A5.Q(3,4) * A5.P(1,2,6)
Q_3 Q_4 P(1,2,6)
sage: A5.pst(2,3)
P(0,0,25)

```

```

>>> from sage.all import *
>>> A2 = SteenrodAlgebra(Integer(2)); A2
mod 2 Steenrod algebra, milnor basis
>>> A2.Sq(Integer(1), Integer(2), Integer(6))
Sq(1,2,6)
>>> A2.Q(Integer(3), Integer(4))  # product of Milnor primitives Q_3 and Q_4
Sq(0,0,0,1,1)
>>> A2.pst(Integer(2), Integer(3))  # Margolis pst element
Sq(0,0,4)
>>> A5 = SteenrodAlgebra(Integer(5)); A5
mod 5 Steenrod algebra, milnor basis
>>> A5.P(Integer(1), Integer(2), Integer(6))

```

(continues on next page)

(continued from previous page)

```

P(1,2,6)
>>> A5.Q(Integer(3), Integer(4))
Q_3 Q_4
>>> A5.Q(Integer(3), Integer(4)) * A5.P(Integer(1), Integer(2), Integer(6))
Q_3 Q_4 P(1,2,6)
>>> A5.pst(Integer(2), Integer(3))
P(0,0,25)

```

You can test whether elements are contained in the Steenrod algebra:

```

sage: w = Sq(2) * Sq(4)
sage: w in SteenrodAlgebra(2)
True
sage: w in SteenrodAlgebra(17)
False

```

```

>>> from sage.all import *
>>> w = Sq(Integer(2)) * Sq(Integer(4))
>>> w in SteenrodAlgebra(Integer(2))
True
>>> w in SteenrodAlgebra(Integer(17))
False

```

Different bases for the Steenrod algebra:

There are two standard vector space bases for the mod p Steenrod algebra: the Milnor basis and the Serre-Cartan basis. When $p = 2$, there are also several other, less well-known, bases. See the documentation for this module (type `sage.algebras.steenrod.steenrod_algebra?`) and the function `steenrod_algebra_basis` for full descriptions of each of the implemented bases.

This module implements the following bases at all primes:

- ‘milnor’: Milnor basis.
- ‘serre-cartan’ or ‘adem’ or ‘admissible’: Serre-Cartan basis.
- ‘pst’, ‘pst_rlex’, ‘pst_llex’, ‘pst_deg’, ‘pst_revz’: various P_t^s -bases.
- ‘comm’, ‘comm_rlex’, ‘comm_llex’, ‘comm_deg’, ‘comm_revz’, or these with ‘_long’ appended: various commutator bases.

It implements the following bases when $p = 2$:

- ‘wood_y’: Wood’s Y basis.
- ‘wood_z’: Wood’s Z basis.
- ‘wall’, ‘wall_long’: Wall’s basis.
- ‘arnon_a’, ‘arnon_a_long’: Arnon’s A basis.
- ‘arnon_c’: Arnon’s C basis.

When defining a Steenrod algebra, you can specify a basis. Then elements of that Steenrod algebra are printed in that basis:

```

sage: adem = SteenrodAlgebra(2, 'adem')
sage: x = adem.Sq(2,1)  # Sq(-) always means a Milnor basis element
sage: x
Sq^4 Sq^1 + Sq^5
sage: y = Sq(0,1)      # unadorned Sq defines elements w.r.t. Milnor basis
sage: y
Sq(0,1)
sage: adem(y)
Sq^2 Sq^1 + Sq^3
sage: adem5 = SteenrodAlgebra(5, 'serre-cartan')
sage: adem5.P(0,2)
P^10 P^2 + 4 P^11 P^1 + P^12

```

```

>>> from sage.all import *
>>> adem = SteenrodAlgebra(Integer(2), 'adem')
>>> x = adem.Sq(Integer(2), Integer(1))  # Sq(-) always means a Milnor basis_
↪element
>>> x
Sq^4 Sq^1 + Sq^5
>>> y = Sq(Integer(0), Integer(1))      # unadorned Sq defines elements w.r.t._
↪Milnor basis
>>> y
Sq(0,1)
>>> adem(y)
Sq^2 Sq^1 + Sq^3
>>> adem5 = SteenrodAlgebra(Integer(5), 'serre-cartan')
>>> adem5.P(Integer(0), Integer(2))
P^10 P^2 + 4 P^11 P^1 + P^12

```

If you add or multiply elements defined using different bases, the left-hand factor determines the form of the output:

```

sage: SteenrodAlgebra(basis='adem').Sq(3) + SteenrodAlgebra(basis='pst').Sq(0,1)
Sq^2 Sq^1
sage: SteenrodAlgebra(basis='pst').Sq(3) + SteenrodAlgebra(basis='milnor').Sq(0,1)
P^0_1 P^1_1 + P^0_2
sage: SteenrodAlgebra(basis='milnor').Sq(2) * SteenrodAlgebra(basis='arnonc').
↪Sq(2)
Sq(1,1)

```

```

>>> from sage.all import *
>>> SteenrodAlgebra(basis='adem').Sq(Integer(3)) + SteenrodAlgebra(basis='pst').
↪Sq(Integer(0), Integer(1))
Sq^2 Sq^1
>>> SteenrodAlgebra(basis='pst').Sq(Integer(3)) + SteenrodAlgebra(basis='milnor').
↪Sq(Integer(0), Integer(1))
P^0_1 P^1_1 + P^0_2
>>> SteenrodAlgebra(basis='milnor').Sq(Integer(2)) * SteenrodAlgebra(basis='arnonc
↪').Sq(Integer(2))
Sq(1,1)

```

You can get a list of basis elements in a given dimension:

```
sage: A3 = SteenrodAlgebra(3, 'milnor')
sage: A3.basis(13)
Family (Q_1 P(2), Q_0 P(3))
```

```
>>> from sage.all import *
>>> A3 = SteenrodAlgebra(Integer(3), 'milnor')
>>> A3.basis(Integer(13))
Family (Q_1 P(2), Q_0 P(3))
```

Algebras defined over different bases are not equal:

```
sage: SteenrodAlgebra(basis='milnor') == SteenrodAlgebra(basis='pst')
False
```

```
>>> from sage.all import *
>>> SteenrodAlgebra(basis='milnor') == SteenrodAlgebra(basis='pst')
False
```

Bases have various synonyms, and in general Sage tries to figure out what basis you meant:

```
sage: SteenrodAlgebra(basis='MiLNOor')
mod 2 Steenrod algebra, milnor basis
sage: SteenrodAlgebra(basis='MiLNOor') == SteenrodAlgebra(basis='milnor')
True
sage: SteenrodAlgebra(basis='adem')
mod 2 Steenrod algebra, serre-cartan basis
sage: SteenrodAlgebra(basis='adem').basis_name()
'serre-cartan'
sage: SteenrodAlgebra(basis='wood---z---').basis_name()
'woodz'
```

```
>>> from sage.all import *
>>> SteenrodAlgebra(basis='MiLNOor')
mod 2 Steenrod algebra, milnor basis
>>> SteenrodAlgebra(basis='MiLNOor') == SteenrodAlgebra(basis='milnor')
True
>>> SteenrodAlgebra(basis='adem')
mod 2 Steenrod algebra, serre-cartan basis
>>> SteenrodAlgebra(basis='adem').basis_name()
'serre-cartan'
>>> SteenrodAlgebra(basis='wood---z---').basis_name()
'woodz'
```

As noted above, several of the bases ('arnon_a', 'wall', 'comm') have alternate, sometimes longer, representations. These provide ways of expressing elements of the Steenrod algebra in terms of the Sq^{2^n} .

```
sage: A_long = SteenrodAlgebra(2, 'arnon_a_long')
sage: A_long(Sq(6))
Sq^1 Sq^2 Sq^1 Sq^2 + Sq^2 Sq^4
sage: SteenrodAlgebra(2, 'wall_long')(Sq(6))
Sq^2 Sq^1 Sq^2 Sq^1 + Sq^2 Sq^4
sage: SteenrodAlgebra(2, 'comm_deg_long')(Sq(6))
s_1 s_2 s_12 + s_2 s_4
```



```

>>> from sage.all import *
>>> A_long = SteenrodAlgebra(Integer(2), 'arnon_a_long')
>>> A_long(Sq(Integer(6)))
Sq^1 Sq^2 Sq^1 Sq^2 + Sq^2 Sq^4
>>> SteenrodAlgebra(Integer(2), 'wall_long')(Sq(Integer(6)))
Sq^2 Sq^1 Sq^2 Sq^1 + Sq^2 Sq^4
>>> SteenrodAlgebra(Integer(2), 'comm_deg_long')(Sq(Integer(6)))
s_1 s_2 s_12 + s_2 s_4

```

Sub-Hopf algebras of the Steenrod algebra:

These are specified using the argument `profile`, along with, optionally, `truncation_type` and `precision`. The `profile` argument specifies the profile function for this algebra. Any sub-Hopf algebra of the Steenrod algebra is determined by its *profile function*. When $p = 2$, this is a map e from the positive integers to the set of nonnegative integers, plus ∞ , corresponding to the sub-Hopf algebra dual to this quotient of the dual Steenrod algebra:

$$\mathbf{F}_2[\xi_1, \xi_2, \xi_3, \dots] / (\xi_1^{2^{e(1)}}, \xi_2^{2^{e(2)}}, \xi_3^{2^{e(3)}}, \dots).$$

The profile function e must satisfy the condition

- $e(r) \geq \min(e(r-i) - i, e(i))$ for all $0 < i < r$.

This is specified via `profile`, and optionally `precision` and `truncation_type`. First, `profile` must have one of the following forms:

- a list or tuple, e.g., `[3, 2, 1]`, corresponding to the function sending 1 to 3, 2 to 2, 3 to 1, and all other integers to the value of `truncation_type`.
- a function from positive integers to nonnegative integers (and ∞), e.g., `lambda n: n+2`.
- `None` or `Infinity` – use this for the profile function for the whole Steenrod algebra

In the first and third cases, `precision` is ignored. In the second case, this function is converted to a tuple of length one less than `precision`, which has default value 100. The function is truncated at this point, and all remaining values are set to the value of `truncation_type`.

`truncation_type` may be 0, ∞ , or 'auto'. If it's 'auto', then it gets converted to 0 in the first case above (when `profile` is a list), and otherwise (when `profile` is a function, `None`, or `Infinity`) it gets converted to ∞ .

For example, the sub-Hopf algebra $A(2)$ has profile function `[3, 2, 1, 0, 0, 0, ...]`, so it can be defined by any of the following:

```

sage: A2 = SteenrodAlgebra(profile=[3,2,1])
sage: B2 = SteenrodAlgebra(profile=[3,2,1,0,0]) # trailing 0s ignored
sage: A2 == B2
True
sage: C2 = SteenrodAlgebra(profile=lambda n: max(4-n, 0), truncation_type=0)
sage: A2 == C2
True

```

```

>>> from sage.all import *
>>> A2 = SteenrodAlgebra(profile=[Integer(3), Integer(2), Integer(1)])
>>> B2 = SteenrodAlgebra(profile=[Integer(3), Integer(2), Integer(1), Integer(0),
  ↳ Integer(0)]) # trailing 0s ignored
>>> A2 == B2

```

(continues on next page)

(continued from previous page)

```

True
>>> C2 = SteenrodAlgebra(profile=lambda n: max(Integer(4)-n, Integer(0)), ↵
↵truncation_type=Integer(0))
>>> A2 == C2
True

```

In the following case, the profile function is specified by a function and `truncation_type` isn't specified, so it defaults to ∞ ; therefore this gives a different sub-Hopf algebra:

```

sage: D2 = SteenrodAlgebra(profile=lambda n: max(4-n, 0))
sage: A2 == D2
False
sage: D2.is_finite()
False
sage: E2 = SteenrodAlgebra(profile=lambda n: max(4-n, 0), truncation_
↵type=Infinity)
sage: D2 == E2
True

```

```

>>> from sage.all import *
>>> D2 = SteenrodAlgebra(profile=lambda n: max(Integer(4)-n, Integer(0)))
>>> A2 == D2
False
>>> D2.is_finite()
False
>>> E2 = SteenrodAlgebra(profile=lambda n: max(Integer(4)-n, Integer(0)), ↵
↵truncation_type=Infinity)
>>> D2 == E2
True

```

The argument `precision` only needs to be specified if the profile function is defined by a function and you want to control when the profile switches from the given function to the truncation type. For example:

```

sage: D3 = SteenrodAlgebra(profile=lambda n: n, precision=3)
sage: D3
sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function [1, 2, ↵
↵+Infinity, +Infinity, +Infinity, ...]
sage: D4 = SteenrodAlgebra(profile=lambda n: n, precision=4); D4
sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function [1, 2, ↵
↵3, +Infinity, +Infinity, +Infinity, ...]
sage: D3 == D4
False

```

```

>>> from sage.all import *
>>> D3 = SteenrodAlgebra(profile=lambda n: n, precision=Integer(3))
>>> D3
sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function [1, 2, ↵
↵+Infinity, +Infinity, +Infinity, ...]
>>> D4 = SteenrodAlgebra(profile=lambda n: n, precision=Integer(4)); D4
sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function [1, 2, ↵
↵3, +Infinity, +Infinity, +Infinity, ...]
>>> D3 == D4

```

(continues on next page)

(continued from previous page)

False

When p is odd, `profile` is a pair of functions e and k , corresponding to the quotient

$$\mathbf{F}_p[\xi_1, \xi_2, \xi_3, \dots] \otimes \Lambda(\tau_0, \tau_1, \dots) / (\xi_1^{p^{e_1}}, \xi_2^{p^{e_2}}, \dots; \tau_0^{k_0}, \tau_1^{k_1}, \dots).$$

Together, the functions e and k must satisfy the conditions

- $e(r) \geq \min(e(r-i) - i, e(i))$ for all $0 < i < r$,
- if $k(i+j) = 1$, then either $e(i) \leq j$ or $k(j) = 1$ for all $i \geq 1, j \geq 0$.

Therefore `profile` must have one of the following forms:

- a pair of lists or tuples, the second of which takes values in the set $\{1, 2\}$, e.g., $([3, 2, 1, 1], [1, 1, 2, 2, 1])$.
- a pair of functions, one from the positive integers to nonnegative integers (and ∞), one from the nonnegative integers to the set $\{1, 2\}$, e.g., $(\text{lambda } n: n+2, \text{lambda } n: 1 \text{ if } n < 3 \text{ else } 2)$.
- `None` or `Infinity` – use this for the profile function for the whole Steenrod algebra

You can also mix and match the first two, passing a pair with first entry a list and second entry a function, for instance. The values of `precision` and `truncation_type` are determined by the first entry.

More examples:

```
sage: E = SteenrodAlgebra(profile=lambda n: 0 if n<3 else 3, truncation_type=0)
sage: E.is_commutative()
True

sage: A2 = SteenrodAlgebra(profile=[3,2,1]) # the algebra A(2)
sage: Sq(7,3,1) in A2
True
sage: Sq(8) in A2
False
sage: Sq(8) in SteenrodAlgebra().basis(8)
True
sage: Sq(8) in A2.basis(8)
False
sage: A2.basis(8)
Family (Sq(1,0,1), Sq(2,2), Sq(5,1))

sage: A5 = SteenrodAlgebra(p=5)
sage: A51 = SteenrodAlgebra(p=5, profile=([1], [2,2]))
sage: A5.Q(0,1) * A5.P(4) in A51
True
sage: A5.Q(2) in A51
False
sage: A5.P(5) in A51
False
```

```
>>> from sage.all import *
>>> E = SteenrodAlgebra(profile=lambda n: Integer(0) if n<Integer(3) else_
↳ Integer(3), truncation_type=Integer(0))
>>> E.is_commutative()
```

(continues on next page)

(continued from previous page)

```

True
>>> A2 = SteenrodAlgebra(profile=[Integer(3), Integer(2), Integer(1)]) # the
↪ algebra A(2)
>>> Sq(Integer(7), Integer(3), Integer(1)) in A2
True
>>> Sq(Integer(8)) in A2
False
>>> Sq(Integer(8)) in SteenrodAlgebra().basis(Integer(8))
True
>>> Sq(Integer(8)) in A2.basis(Integer(8))
False
>>> A2.basis(Integer(8))
Family (Sq(1,0,1), Sq(2,2), Sq(5,1))

>>> A5 = SteenrodAlgebra(p=Integer(5))
>>> A51 = SteenrodAlgebra(p=Integer(5), profile=([Integer(1)], [Integer(2),
↪ Integer(2)]))
>>> A5.Q(Integer(0), Integer(1)) * A5.P(Integer(4)) in A51
True
>>> A5.Q(Integer(2)) in A51
False
>>> A5.P(Integer(5)) in A51
False

```

For sub-Hopf algebras of the Steenrod algebra, only the Milnor basis or the various P_t^s -bases may be used.

```

sage: SteenrodAlgebra(profile=[1,2,1,1], basis='adem')
Traceback (most recent call last):
...
NotImplementedError: for sub-Hopf algebras of the Steenrod algebra, only the
↪ Milnor basis and the pst bases are implemented

```

```

>>> from sage.all import *
>>> SteenrodAlgebra(profile=[Integer(1), Integer(2), Integer(1), Integer(1)], basis=
↪ 'adem')
Traceback (most recent call last):
...
NotImplementedError: for sub-Hopf algebras of the Steenrod algebra, only the
↪ Milnor basis and the pst bases are implemented

```

The generic Steenrod algebra at the prime 2:

The structure formulas for the Steenrod algebra at odd primes p also make sense when p is set to 2. We refer to the resulting algebra as the “generic Steenrod algebra” for the prime 2. The dual Hopf algebra is given by

$$A_* = \mathbf{F}_2[\xi_1, \xi_2, \xi_3, \dots] \otimes \Lambda(\tau_0, \tau_1, \dots)$$

The degree of ξ_k is $2^{k+1} - 2$ and the degree of τ_k is $2^{k+1} - 1$.

The generic Steenrod algebra is an associated graded algebra of the usual Steenrod algebra that is occasionally useful. Its cohomology, for example, is the E_2 -term of a spectral sequence that computes the E_2 -term of the

Novikov spectral sequence. It can also be obtained as a specialisation of Voevodsky’s “motivic Steenrod algebra”: in the notation of [Voe2003], Remark 12.12, it corresponds to setting $\rho = \tau = 0$. The usual Steenrod algebra is given by $\rho = 0$ and $\tau = 1$.

In Sage this algebra is constructed using the ‘generic’ keyword.

Example:

```
sage: EA = SteenrodAlgebra(p=2, generic=True) ; EA
generic mod 2 Steenrod algebra, milnor basis
sage: EA[8]
Vector space spanned by (Q_0 Q_2, Q_0 Q_1 P(2), P(1,1), P(4))
over Finite Field of size 2
```

```
>>> from sage.all import *
>>> EA = SteenrodAlgebra(p=Integer(2), generic=True) ; EA
generic mod 2 Steenrod algebra, milnor basis
>>> EA[Integer(8)]
Vector space spanned by (Q_0 Q_2, Q_0 Q_1 P(2), P(1,1), P(4))
over Finite Field of size 2
```

```
class sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic (p=2, basis='milnor',
                                                                    **kws)
```

Bases: `CombinatorialFreeModule`

The mod p Steenrod algebra.

Users should not call this, but use the function `SteenrodAlgebra()` instead. See that function for extensive documentation.

EXAMPLES:

```
sage: sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic()
mod 2 Steenrod algebra, milnor basis
sage: sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic(5)
mod 5 Steenrod algebra, milnor basis
sage: sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic(5, 'adem')
mod 5 Steenrod algebra, serre-cartan basis
```

```
>>> from sage.all import *
>>> sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic()
mod 2 Steenrod algebra, milnor basis
>>> sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic(Integer(5))
mod 5 Steenrod algebra, milnor basis
>>> sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic(Integer(5),
↪ 'adem')
mod 5 Steenrod algebra, serre-cartan basis
```

```
class Element
```

Bases: `IndexedFreeModuleElement`

Class for elements of the Steenrod algebra. Since the Steenrod algebra class is based on `CombinatorialFreeModule`, this is based on `IndexedFreeModuleElement`. It has new methods reflecting its role, like `degree()` for computing the degree of an element.

EXAMPLES:

Since this class inherits from `IndexedFreeModuleElement`, elements can be used as iterators, and there are other useful methods:

```
sage: c = Sq(5).antipode(); c
Sq(2,1) + Sq(5)
sage: for mono, coeff in c: print((coeff, mono))
(1, (5,))
(1, (2, 1))
sage: c.monomial_coefficients() == {(2, 1): 1, (5,): 1}
True
sage: sorted(c.monomials(), key=lambda x: tuple(x.support()))
[Sq(2,1), Sq(5)]
sage: sorted(c.support())
[(2, 1), (5,)]
```

```
>>> from sage.all import *
>>> c = Sq(Integer(5)).antipode(); c
Sq(2,1) + Sq(5)
>>> for mono, coeff in c: print((coeff, mono))
(1, (5,))
(1, (2, 1))
>>> c.monomial_coefficients() == {(Integer(2), Integer(1)): Integer(1),
↪ (Integer(5),): Integer(1)}
True
>>> sorted(c.monomials(), key=lambda x: tuple(x.support()))
[Sq(2,1), Sq(5)]
>>> sorted(c.support())
[(2, 1), (5,)]
```

See the documentation for this module (type `sage.algebras.steenrod.steenrod_algebra?`) for more information about elements of the Steenrod algebra.

`additive_order()`

The additive order of any nonzero element of the mod p Steenrod algebra is p .

OUTPUT: 1 (for the zero element) or p (for anything else)

EXAMPLES:

```
sage: z = Sq(4) + Sq(6) + 1
sage: z.additive_order()
2
sage: (Sq(3) + Sq(3)).additive_order()
1
```

```
>>> from sage.all import *
>>> z = Sq(Integer(4)) + Sq(Integer(6)) + Integer(1)
>>> z.additive_order()
2
>>> (Sq(Integer(3)) + Sq(Integer(3))).additive_order()
1
```

`basis_name()`

The basis name associated to self.

EXAMPLES:

```

sage: a = SteenrodAlgebra().Sq(3,2,1)
sage: a.basis_name()
'milnor'
sage: a.change_basis('adem').basis_name()
'serre-cartan'
sage: a.change_basis('wood____y').basis_name()
'woody'
sage: b = SteenrodAlgebra(p=7).basis(36)[0]
sage: b.basis_name()
'milnor'
sage: a.change_basis('adem').basis_name()
'serre-cartan'

```

```

>>> from sage.all import *
>>> a = SteenrodAlgebra().Sq(Integer(3), Integer(2), Integer(1))
>>> a.basis_name()
'milnor'
>>> a.change_basis('adem').basis_name()
'serre-cartan'
>>> a.change_basis('wood____y').basis_name()
'woody'
>>> b = SteenrodAlgebra(p=Integer(7)).basis(Integer(36))[Integer(0)]
>>> b.basis_name()
'milnor'
>>> a.change_basis('adem').basis_name()
'serre-cartan'

```

change_basis (*basis*='milnor')

Representation of element with respect to basis.

INPUT:

- *basis* – string; basis in which to work

OUTPUT: representation of *self* in given basis

The choices for *basis* are:

- 'milnor' for the Milnor basis.
- 'serre-cartan', 'serre_cartan', 'sc', 'adem', 'admissible' for the Serre-Cartan basis.
- 'wood_y' for Wood's Y basis.
- 'wood_z' for Wood's Z basis.
- 'wall' for Wall's basis.
- 'wall_long' for Wall's basis, alternate representation
- 'arnon_a' for Arnon's A basis.
- 'arnon_a_long' for Arnon's A basis, alternate representation.
- 'arnon_c' for Arnon's C basis.
- 'pst', 'pst_rlex', 'pst_llex', 'pst_deg', 'pst_revz' for various P_t^s -bases.
- 'comm', 'comm_rlex', 'comm_llex', 'comm_deg', 'comm_revz' for various commutator bases.
- 'comm_long', 'comm_rlex_long', etc., for commutator bases, alternate representations.

See documentation for this module (by browsing the reference manual or by typing `sage.algebras.steenrod.steenrod_algebra?`) for descriptions of the different bases.

EXAMPLES:

```

sage: c = Sq(2) * Sq(1)
sage: c.change_basis('milnor')

```

(continues on next page)

(continued from previous page)

```

Sq(0,1) + Sq(3)
sage: c.change_basis('serre-cartan')
Sq^2 Sq^1
sage: d = Sq(0,0,1)
sage: d.change_basis('arnonc')
Sq^2 Sq^5 + Sq^4 Sq^2 Sq^1 + Sq^4 Sq^3 + Sq^7

```

```

>>> from sage.all import *
>>> c = Sq(Integer(2)) * Sq(Integer(1))
>>> c.change_basis('milnor')
Sq(0,1) + Sq(3)
>>> c.change_basis('serre-cartan')
Sq^2 Sq^1
>>> d = Sq(Integer(0),Integer(0),Integer(1))
>>> d.change_basis('arnonc')
Sq^2 Sq^5 + Sq^4 Sq^2 Sq^1 + Sq^4 Sq^3 + Sq^7

```

coproduct (*algorithm*='milnor')

The coproduct of this element.

INPUT:

- *algorithm* – None or a string, either 'milnor' or 'serre-cartan' (or anything which will be converted to one of these by the function `get_basis_name`). If None, default to 'serre-cartan' if current basis is 'serre-cartan'; otherwise use 'milnor'.

See `SteenrodAlgebra_generic.coproduct_on_basis()` for more information on computing the coproduct.

EXAMPLES:

```

sage: a = Sq(2)
sage: a.coproduct()
1 # Sq(2) + Sq(1) # Sq(1) + Sq(2) # 1
sage: b = Sq(4)
sage: (a*b).coproduct() == (a.coproduct()) * (b.coproduct())
True

sage: c = a.change_basis('adem'); c.coproduct(algorithm='milnor')
1 # Sq^2 + Sq^1 # Sq^1 + Sq^2 # 1
sage: c = a.change_basis('adem'); c.coproduct(algorithm='adem')
1 # Sq^2 + Sq^1 # Sq^1 + Sq^2 # 1

sage: d = a.change_basis('comm_long'); d.coproduct()
1 # s_2 + s_1 # s_1 + s_2 # 1

sage: A7 = SteenrodAlgebra(p=7)
sage: a = A7.Q(1) * A7.P(1); a
Q_1 P(1)
sage: a.coproduct()
1 # Q_1 P(1) + P(1) # Q_1 + Q_1 # P(1) + Q_1 P(1) # 1
sage: a.coproduct(algorithm='adem')
1 # Q_1 P(1) + P(1) # Q_1 + Q_1 # P(1) + Q_1 P(1) # 1

```



```

>>> from sage.all import *
>>> a = Sq(Integer(2))
>>> a.coproduct()
1 # Sq(2) + Sq(1) # Sq(1) + Sq(2) # 1
>>> b = Sq(Integer(4))
>>> (a*b).coproduct() == (a.coproduct()) * (b.coproduct())
True

>>> c = a.change_basis('adem'); c.coproduct(algorithm='milnor')
1 # Sq^2 + Sq^1 # Sq^1 + Sq^2 # 1
>>> c = a.change_basis('adem'); c.coproduct(algorithm='adem')
1 # Sq^2 + Sq^1 # Sq^1 + Sq^2 # 1

>>> d = a.change_basis('comm_long'); d.coproduct()
1 # s_2 + s_1 # s_1 + s_2 # 1

>>> A7 = SteenrodAlgebra(p=Integer(7))
>>> a = A7.Q(Integer(1)) * A7.P(Integer(1)); a
Q_1 P(1)
>>> a.coproduct()
1 # Q_1 P(1) + P(1) # Q_1 + Q_1 # P(1) + Q_1 P(1) # 1
>>> a.coproduct(algorithm='adem')
1 # Q_1 P(1) + P(1) # Q_1 + Q_1 # P(1) + Q_1 P(1) # 1

```

Once you have an element of the tensor product, you may want to extract the tensor factors of its summands.

```

sage: b = Sq(2).coproduct()
sage: b
1 # Sq(2) + Sq(1) # Sq(1) + Sq(2) # 1
sage: supp = sorted(b.support()); supp
[((), (2,)), ((1,), (1,)), ((2,), ())]
sage: Sq(*supp[0][0])
1
sage: Sq(*supp[0][1])
Sq(2)
sage: [(Sq(*x), Sq(*y)) for (x,y) in supp]
[(1, Sq(2)), (Sq(1), Sq(1)), (Sq(2), 1)]

```

```

>>> from sage.all import *
>>> b = Sq(Integer(2)).coproduct()
>>> b
1 # Sq(2) + Sq(1) # Sq(1) + Sq(2) # 1
>>> supp = sorted(b.support()); supp
[((), (2,)), ((1,), (1,)), ((2,), ())]
>>> Sq(*supp[Integer(0)][Integer(0)])
1
>>> Sq(*supp[Integer(0)][Integer(1)])
Sq(2)
>>> [(Sq(*x), Sq(*y)) for (x,y) in supp]
[(1, Sq(2)), (Sq(1), Sq(1)), (Sq(2), 1)]

```

The support of an element does not include the coefficients, so at odd primes it may be better to use

```
monomial_coefficients:
```

```
sage: A3 = SteenrodAlgebra(p=3)
sage: b = (A3.P(1)**2).coproduct()
sage: b
2*1 # P(2) + 2*P(1) # P(1) + 2*P(2) # 1
sage: sorted(b.support())
[(((), ()), ((), (2,))), (((), (1,)), ((), (1,))), (((), (2,)), ((), ()))]
sage: b.monomial_coefficients()
{(((), ()), ((), (2,))): 2,
 (((), (1,)), ((), (1,))): 2,
 (((), (2,)), ((), ())) : 2}
sage: mc = b.monomial_coefficients()
sage: sorted([(A3.monomial(x), A3.monomial(y), mc[x,y]) for (x,y) in mc])
[(1, P(2), 2), (P(1), P(1), 2), (P(2), 1, 2)]
```

```
>>> from sage.all import *
>>> A3 = SteenrodAlgebra(p=Integer(3))
>>> b = (A3.P(Integer(1))**Integer(2)).coproduct()
>>> b
2*1 # P(2) + 2*P(1) # P(1) + 2*P(2) # 1
>>> sorted(b.support())
[(((), ()), ((), (2,))), (((), (1,)), ((), (1,))), (((), (2,)), ((), ()))]
>>> b.monomial_coefficients()
{(((), ()), ((), (2,))): 2,
 (((), (1,)), ((), (1,))): 2,
 (((), (2,)), ((), ())) : 2}
>>> mc = b.monomial_coefficients()
>>> sorted([(A3.monomial(x), A3.monomial(y), mc[x,y]) for (x,y) in mc])
[(1, P(2), 2), (P(1), P(1), 2), (P(2), 1, 2)]
```

degree()

The degree of self.

The degree of $Sq(i_1, i_2, i_3, \dots)$ is

$$i_1 + 3i_2 + 7i_3 + \dots + (2^k - 1)i_k + \dots$$

At an odd prime p , the degree of Q_k is $2p^k - 1$ and the degree of $\mathcal{P}(i_1, i_2, \dots)$ is

$$\sum_{k \geq 0} 2(p^k - 1)i_k.$$

ALGORITHM: If `is_homogeneous()` returns `True`, call `SteenrodAlgebra_generic.degree_on_basis()` on the leading summand.

EXAMPLES:

```
sage: Sq(0,0,1).degree()
7
sage: (Sq(0,0,1) + Sq(7)).degree()
7
sage: (Sq(0,0,1) + Sq(2)).degree()
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
ValueError: element is not homogeneous
```

```
sage: A11 = SteenrodAlgebra(p=11)
```

```
sage: A11.P(1).degree()
```

```
20
```

```
sage: A11.P(1,1).degree()
```

```
260
```

```
sage: A11.Q(2).degree()
```

```
241
```

```
>>> from sage.all import *
```

```
>>> Sq(Integer(0), Integer(0), Integer(1)).degree()
```

```
7
```

```
>>> (Sq(Integer(0), Integer(0), Integer(1)) + Sq(Integer(7))).degree()
```

```
7
```

```
>>> (Sq(Integer(0), Integer(0), Integer(1)) + Sq(Integer(2))).degree()
```

```
Traceback (most recent call last):
```

```
...
```

```
ValueError: element is not homogeneous
```

```
>>> A11 = SteenrodAlgebra(p=Integer(11))
```

```
>>> A11.P(Integer(1)).degree()
```

```
20
```

```
>>> A11.P(Integer(1), Integer(1)).degree()
```

```
260
```

```
>>> A11.Q(Integer(2)).degree()
```

```
241
```

excess()

Excess of element.

OUTPUT: `excess` – nonnegative integer

The excess of a Milnor basis element $Sq(a, b, c, \dots)$ is $a + b + c + \dots$. When p is odd, the excess of $Q_0^{e_0} Q_1^{e_1} \cdots P(r_1, r_2, \dots)$ is $\sum e_i + 2 \sum r_i$. The excess of a linear combination of Milnor basis elements is the minimum of the excesses of those basis elements.

See [Kr1971] for the proofs of these assertions.

EXAMPLES:

```
sage: a = Sq(1, 2, 3)
```

```
sage: a.excess()
```

```
6
```

```
sage: (Sq(0, 0, 1) + Sq(4, 1) + Sq(7)).excess()
```

```
1
```

```
sage: elt = Sq(0, 0, 1) + Sq(4, 1) + Sq(7)
```

```
sage: M = sorted(elt.monomials(), key=lambda x: tuple(x.support()))
```

```
sage: [m.excess() for m in M]
```

```
[1, 5, 7]
```

```
sage: [m for m in M]
```

```
[Sq(0, 0, 1), Sq(4, 1), Sq(7)]
```

```
sage: B = SteenrodAlgebra(7)
```

```
sage: a = B.Q(1, 2, 5)
```

(continues on next page)

(continued from previous page)

```

sage: b = B.P(2,2,3)
sage: a.excess()
3
sage: b.excess()
14
sage: (a + b).excess()
3
sage: (a * b).excess()
17

```

```

>>> from sage.all import *
>>> a = Sq(Integer(1), Integer(2), Integer(3))
>>> a.excess()
6
>>> (Sq(Integer(0), Integer(0), Integer(1)) + Sq(Integer(4), Integer(1)) +
↳ Sq(Integer(7))) .excess()
1
>>> elt = Sq(Integer(0), Integer(0), Integer(1)) + Sq(Integer(4),
↳ Integer(1)) + Sq(Integer(7))
>>> M = sorted(elt.monomials(), key=lambda x: tuple(x.support()))
>>> [m.excess() for m in M]
[1, 5, 7]
>>> [m for m in M]
[Sq(0,0,1), Sq(4,1), Sq(7)]
>>> B = SteenrodAlgebra(Integer(7))
>>> a = B.Q(Integer(1), Integer(2), Integer(5))
>>> b = B.P(Integer(2), Integer(2), Integer(3))
>>> a.excess()
3
>>> b.excess()
14
>>> (a + b).excess()
3
>>> (a * b).excess()
17

```

is_decomposable()

Return True if element is decomposable, False otherwise.

That is, if element is in the square of the augmentation ideal, return True; otherwise, return False.

OUTPUT: boolean

EXAMPLES:

```

sage: a = Sq(6)
sage: a.is_decomposable()
True
sage: for i in range(9):
.....:     if not Sq(i).is_decomposable():
.....:         print(Sq(i))
1
Sq(1)

```

(continues on next page)

(continued from previous page)

```

Sq(2)
Sq(4)
Sq(8)
sage: A3 = SteenrodAlgebra(p=3, basis='adem')
sage: [A3.P(n) for n in range(30) if not A3.P(n).is_decomposable()]
[1, P^1, P^3, P^9, P^27]

```

```

>>> from sage.all import *
>>> a = Sq(Integer(6))
>>> a.is_decomposable()
True
>>> for i in range(Integer(9)):
...     if not Sq(i).is_decomposable():
...         print(Sq(i))
1
Sq(1)
Sq(2)
Sq(4)
Sq(8)
>>> A3 = SteenrodAlgebra(p=Integer(3), basis='adem')
>>> [A3.P(n) for n in range(Integer(30)) if not A3.P(n).is_decomposable()]
[1, P^1, P^3, P^9, P^27]

```

is_homogeneous()

Return True iff this element is homogeneous.

EXAMPLES:

```

sage: (Sq(0,0,1) + Sq(7)).is_homogeneous()
True
sage: (Sq(0,0,1) + Sq(2)).is_homogeneous()
False

```

```

>>> from sage.all import *
>>> (Sq(Integer(0), Integer(0), Integer(1)) + Sq(Integer(7))).is_
↪homogeneous()
True
>>> (Sq(Integer(0), Integer(0), Integer(1)) + Sq(Integer(2))).is_
↪homogeneous()
False

```

is_nilpotent()

Return True if element is not a unit, False otherwise.

EXAMPLES:

```

sage: z = Sq(4,2) + Sq(7,1) + Sq(3,0,1)
sage: z.is_nilpotent()
True
sage: u = 1 + Sq(3,1)
sage: u == 1 + Sq(3,1)
True

```

(continues on next page)

(continued from previous page)

```
sage: u.is_nilpotent()
False
```

```
>>> from sage.all import *
>>> z = Sq(Integer(4), Integer(2)) + Sq(Integer(7), Integer(1)) +
↳ Sq(Integer(3), Integer(0), Integer(1))
>>> z.is_nilpotent()
True
>>> u = Integer(1) + Sq(Integer(3), Integer(1))
>>> u == Integer(1) + Sq(Integer(3), Integer(1))
True
>>> u.is_nilpotent()
False
```

is_unit()

Return True if element has a nonzero scalar multiple of $P(0)$ as a summand, False otherwise.

EXAMPLES:

```
sage: z = Sq(4, 2) + Sq(7, 1) + Sq(3, 0, 1)
sage: z.is_unit()
False
sage: u = Sq(0) + Sq(3, 1)
sage: u == 1 + Sq(3, 1)
True
sage: u.is_unit()
True
sage: A5 = SteenrodAlgebra(5)
sage: v = A5.P(0)
sage: (v + v + v).is_unit()
True
```

```
>>> from sage.all import *
>>> z = Sq(Integer(4), Integer(2)) + Sq(Integer(7), Integer(1)) +
↳ Sq(Integer(3), Integer(0), Integer(1))
>>> z.is_unit()
False
>>> u = Sq(Integer(0)) + Sq(Integer(3), Integer(1))
>>> u == Integer(1) + Sq(Integer(3), Integer(1))
True
>>> u.is_unit()
True
>>> A5 = SteenrodAlgebra(Integer(5))
>>> v = A5.P(Integer(0))
>>> (v + v + v).is_unit()
True
```

may_weight()

May's 'weight' of element.

OUTPUT: weight – nonnegative integer

If we let $F_*(A)$ be the May filtration of the Steenrod algebra, the weight of an element x is the integer k so that x is in $F_k(A)$ and not in $F_{k+1}(A)$. According to Theorem 2.6 in May's thesis [May1964], the

weight of a Milnor basis element is computed as follows: first, to compute the weight of $P(r_1, r_2, \dots)$, write each r_i in base p as $r_i = \sum_j p^j r_{ij}$. Then each nonzero binary digit r_{ij} contributes i to the weight: the weight is $\sum_{i,j} i r_{ij}$. When p is odd, the weight of Q_i is $i + 1$, so the weight of a product $Q_{i_1} Q_{i_2} \dots$ equals $(i_1 + 1) + (i_2 + 1) + \dots$. Then the weight of $Q_{i_1} Q_{i_2} \dots P(r_1, r_2, \dots)$ is the sum of $(i_1 + 1) + (i_2 + 1) + \dots$ and $\sum_{i,j} i r_{ij}$.

The weight of a sum of Milnor basis elements is the minimum of the weights of the summands.

When $p = 2$, we compute the weight on Milnor basis elements by adding up the terms in their ‘height’ - see `wall_height()` for documentation. (When p is odd, the height of an element is not defined.)

EXAMPLES:

```
sage: Sq(0).may_weight()
0
sage: a = Sq(4)
sage: a.may_weight()
1
sage: b = Sq(4)*Sq(8) + Sq(8)*Sq(4)
sage: b.may_weight()
2
sage: Sq(2,1,5).wall_height()
[2, 3, 2, 1, 1]
sage: Sq(2,1,5).may_weight()
9
sage: A5 = SteenrodAlgebra(5)
sage: a = A5.Q(1,2,4)
sage: b = A5.P(1,2,1)
sage: a.may_weight()
10
sage: b.may_weight()
8
sage: (a * b).may_weight()
18
sage: A5.P(0,0,1).may_weight()
3
```

```
>>> from sage.all import *
>>> Sq(Integer(0)).may_weight()
0
>>> a = Sq(Integer(4))
>>> a.may_weight()
1
>>> b = Sq(Integer(4))*Sq(Integer(8)) + Sq(Integer(8))*Sq(Integer(4))
>>> b.may_weight()
2
>>> Sq(Integer(2), Integer(1), Integer(5)).wall_height()
[2, 3, 2, 1, 1]
>>> Sq(Integer(2), Integer(1), Integer(5)).may_weight()
9
>>> A5 = SteenrodAlgebra(Integer(5))
>>> a = A5.Q(Integer(1), Integer(2), Integer(4))
>>> b = A5.P(Integer(1), Integer(2), Integer(1))
>>> a.may_weight()
10
```

(continues on next page)

(continued from previous page)

```
>>> b.may_weight()
8
>>> (a * b).may_weight()
18
>>> A5.P(Integer(0), Integer(0), Integer(1)).may_weight()
3
```

milnor()

Return this element in the Milnor basis; that is, as an element of the appropriate Steenrod algebra.

This just calls the method `SteenrodAlgebra_generic.milnor()`.

EXAMPLES:

```
sage: Adem = SteenrodAlgebra(basis='adem')
sage: a = Adem.basis(4)[1]; a
Sq^3 Sq^1
sage: a.milnor()
Sq(1,1)
```

```
>>> from sage.all import *
>>> Adem = SteenrodAlgebra(basis='adem')
>>> a = Adem.basis(Integer(4))[Integer(1)]; a
Sq^3 Sq^1
>>> a.milnor()
Sq(1,1)
```

prime()

The prime associated to self.

EXAMPLES:

```
sage: a = SteenrodAlgebra().Sq(3,2,1)
sage: a.prime()
2
sage: a.change_basis('adem').prime()
2
sage: b = SteenrodAlgebra(p=7).basis(36)[0]
sage: b.prime()
7
sage: SteenrodAlgebra(p=3, basis='adem').one().prime()
3
```

```
>>> from sage.all import *
>>> a = SteenrodAlgebra().Sq(Integer(3), Integer(2), Integer(1))
>>> a.prime()
2
>>> a.change_basis('adem').prime()
2
>>> b = SteenrodAlgebra(p=Integer(7)).basis(Integer(36))[Integer(0)]
>>> b.prime()
7
```

(continues on next page)

(continued from previous page)

```
>>> SteenrodAlgebra(p=Integer(3), basis='adem').one().prime()
3
```

wall_height()

Wall's 'height' of element.

OUTPUT: list of nonnegative integers

The height of an element of the mod 2 Steenrod algebra is a list of nonnegative integers, defined as follows: if the element is a monomial in the generators $Sq(2^i)$, then the i -th entry in the list is the number of times $Sq(2^i)$ appears. For an arbitrary element, write it as a sum of such monomials; then its height is the maximum, ordered right-lexicographically, of the heights of those monomials.

When p is odd, the height of an element is not defined.

According to Theorem 3 in [Wal1960], the height of the Milnor basis element $Sq(r_1, r_2, \dots)$ is obtained as follows: write each r_i in binary as $r_i = \sum_j 2^j r_{ij}$. Then each nonzero binary digit r_{ij} contributes 1 to the k -th entry in the height, for $j \leq k \leq i + j - 1$.

EXAMPLES:

```
sage: Sq(0).wall_height()
[]
sage: a = Sq(4)
sage: a.wall_height()
[0, 0, 1]
sage: b = Sq(4)*Sq(8) + Sq(8)*Sq(4)
sage: b.wall_height()
[0, 0, 1, 1]
sage: Sq(0,0,3).wall_height()
[1, 2, 2, 1]
```

```
>>> from sage.all import *
>>> Sq(Integer(0)).wall_height()
[]
>>> a = Sq(Integer(4))
>>> a.wall_height()
[0, 0, 1]
>>> b = Sq(Integer(4))*Sq(Integer(8)) + Sq(Integer(8))*Sq(Integer(4))
>>> b.wall_height()
[0, 0, 1, 1]
>>> Sq(Integer(0), Integer(0), Integer(3)).wall_height()
[1, 2, 2, 1]
```

P(*nums)The element $P(a, b, c, \dots)$.

INPUT:

- $a, b, c, \dots$ – nonnegative integers

OUTPUT:

element of the Steenrod algebra given by the Milnor single basis element $P(a, b, c, \dots)$ Note that at the prime 2, this is the same element as $Sq(a, b, c, \dots)$.

EXAMPLES:

```

sage: A = SteenrodAlgebra(2)
sage: A.P(5)
Sq(5)
sage: B = SteenrodAlgebra(3)
sage: B.P(5,1,1)
P(5,1,1)
sage: B.P(1,1,-12,1)
Traceback (most recent call last):
...
TypeError: entries must be nonnegative integers

sage: SteenrodAlgebra(basis='serre-cartan').P(0,1)
Sq^2 Sq^1 + Sq^3
sage: SteenrodAlgebra(generic=True).P(2,0,1)
P(2,0,1)

```

```

>>> from sage.all import *
>>> A = SteenrodAlgebra(Integer(2))
>>> A.P(Integer(5))
Sq(5)
>>> B = SteenrodAlgebra(Integer(3))
>>> B.P(Integer(5), Integer(1), Integer(1))
P(5,1,1)
>>> B.P(Integer(1), Integer(1), -Integer(12), Integer(1))
Traceback (most recent call last):
...
TypeError: entries must be nonnegative integers

>>> SteenrodAlgebra(basis='serre-cartan').P(Integer(0), Integer(1))
Sq^2 Sq^1 + Sq^3
>>> SteenrodAlgebra(generic=True).P(Integer(2), Integer(0), Integer(1))
P(2,0,1)

```

$Q(*nums)$

The element $Q_{n_0}Q_{n_1}\dots$, given by specifying the subscripts.

INPUT:

- $n_0, n_1, \dots$ – nonnegative integers

OUTPUT: the element $Q_{n_0}Q_{n_1}\dots$

Note that at the prime 2, Q_n is the element $Sq(0, 0, \dots, 1)$, where the 1 is in the $(n+1)^{st}$ position.

Compare this to the method `Q_exp()`, which defines a similar element, but by specifying the tuple of exponents.

EXAMPLES:

```

sage: A2 = SteenrodAlgebra(2)
sage: A2.Q(2,3)
Sq(0,0,1,1)
sage: A5 = SteenrodAlgebra(5)
sage: A5.Q(1,4)
Q_1 Q_4

```

(continues on next page)

(continued from previous page)

```

sage: A5.Q(1,4) == A5.Q_exp(0,1,0,0,1)
True
sage: H = SteenrodAlgebra(p=5, profile=[[2,1], [2,2,2]])
sage: H.Q(2)
Q_2
sage: H.Q(4)
Traceback (most recent call last):
...
ValueError: Element not in this algebra

```

```

>>> from sage.all import *
>>> A2 = SteenrodAlgebra(Integer(2))
>>> A2.Q(Integer(2), Integer(3))
Sq(0,0,1,1)
>>> A5 = SteenrodAlgebra(Integer(5))
>>> A5.Q(Integer(1), Integer(4))
Q_1 Q_4
>>> A5.Q(Integer(1), Integer(4)) == A5.Q_exp(Integer(0), Integer(1), Integer(0),
↪ Integer(0), Integer(1))
True
>>> H = SteenrodAlgebra(p=Integer(5), profile=[[Integer(2), Integer(1)], ↪
↪ [Integer(2), Integer(2), Integer(2)]])
>>> H.Q(Integer(2))
Q_2
>>> H.Q(Integer(4))
Traceback (most recent call last):
...
ValueError: Element not in this algebra

```

Q_exp(*nums)

The element $Q_0^{e_0} Q_1^{e_1} \dots$, given by specifying the exponents.

INPUT:

- $e_0, e_1, \dots$ – sequence of 0s and 1s

OUTPUT: the element $Q_0^{e_0} Q_1^{e_1} \dots$

Note that at the prime 2, Q_n is the element $Sq(0, 0, \dots, 1)$, where the 1 is in the $(n+1)^{st}$ position.

Compare this to the method `Q()`, which defines a similar element, but by specifying the tuple of subscripts of terms with exponent 1.

EXAMPLES:

```

sage: A2 = SteenrodAlgebra(2)
sage: A5 = SteenrodAlgebra(5)
sage: A2.Q_exp(0,0,1,1,0)
Sq(0,0,1,1)
sage: A5.Q_exp(0,0,1,1,0)
Q_2 Q_3
sage: A5.Q(2,3)
Q_2 Q_3
sage: A5.Q_exp(0,0,1,1,0) == A5.Q(2,3)
True

```

(continues on next page)

(continued from previous page)

```
sage: SteenrodAlgebra(2, generic=True).Q_exp(1, 0, 1)
Q_0 Q_2
```

```
>>> from sage.all import *
>>> A2 = SteenrodAlgebra(Integer(2))
>>> A5 = SteenrodAlgebra(Integer(5))
>>> A2.Q_exp(Integer(0), Integer(0), Integer(1), Integer(1), Integer(0))
Sq(0, 0, 1, 1)
>>> A5.Q_exp(Integer(0), Integer(0), Integer(1), Integer(1), Integer(0))
Q_2 Q_3
>>> A5.Q(Integer(2), Integer(3))
Q_2 Q_3
>>> A5.Q_exp(Integer(0), Integer(0), Integer(1), Integer(1), Integer(0)) == A5.
↳Q(Integer(2), Integer(3))
True
>>> SteenrodAlgebra(Integer(2), generic=True).Q_exp(Integer(1), Integer(0),
↳Integer(1))
Q_0 Q_2
```

algebra_generators()

Family of generators for this algebra.

OUTPUT: family of elements of this algebra

At the prime 2, the Steenrod algebra is generated by the elements Sq^{2^i} for $i \geq 0$. At odd primes, it is generated by the elements Q_0 and $\mathcal{P}^{p^i}$ for $i \geq 0$. So if this algebra is the entire Steenrod algebra, return an infinite family made up of these elements.

For sub-Hopf algebras of the Steenrod algebra, it is not always clear what a minimal generating set is. The sub-Hopf algebra $A(n)$ is minimally generated by the elements Sq^{2^i} for $0 \leq i \leq n$ at the prime 2. At odd primes, $A(n)$ is minimally generated by Q_0 along with $\mathcal{P}^{p^i}$ for $0 \leq i \leq n - 1$. So if this algebra is $A(n)$, return the appropriate list of generators.

For other sub-Hopf algebras: return a non-minimal generating set: the family of P_t^s 's and Q_n 's contained in the algebra.

EXAMPLES:

```
sage: A3 = SteenrodAlgebra(3, 'adem')
sage: A3.gens()
Lazy family (<bound method SteenrodAlgebra_generic.gen of mod 3 Steenrod_
↳algebra,
  serre-cartan basis>(i))_{i in Non negative integers}
sage: A3.gens()[0]
beta
sage: A3.gens()[1]
P^1
sage: A3.gens()[2]
P^3
sage: SteenrodAlgebra(profile=[3, 2, 1]).gens()
Family (Sq(1), Sq(2), Sq(4))
```

```

>>> from sage.all import *
>>> A3 = SteenrodAlgebra(Integer(3), 'adem')
>>> A3.gens()
Lazy family (<bound method SteenrodAlgebra_generic.gen of mod 3 Steenrod_
↳algebra,
  serre-cartan basis>(i))_{i in Non negative integers}
>>> A3.gens()[Integer(0)]
beta
>>> A3.gens()[Integer(1)]
P^1
>>> A3.gens()[Integer(2)]
P^3
>>> SteenrodAlgebra(profile=[Integer(3),Integer(2),Integer(1)]).gens()
Family (Sq(1), Sq(2), Sq(4))

```

In the following case, return a non-minimal generating set. (It is not minimal because $Sq(0,0,1)$ is the commutator of $Sq(1)$ and $Sq(0,2)$.)

```

sage: SteenrodAlgebra(profile=[1,2,1]).gens()
Family (Sq(1), Sq(0,1), Sq(0,2), Sq(0,0,1))
sage: SteenrodAlgebra(p=5, profile=[[2,1], [2,2,2]]).gens()
Family (Q_0, P(1), P(5))
sage: SteenrodAlgebra(profile=lambda n: n).gens()
Lazy family (<bound method SteenrodAlgebra_generic.gen of sub-Hopf algebra
of mod 2 Steenrod algebra, milnor basis, profile function [1, 2, 3, ...,
98, 99, +Infinity, +Infinity, +Infinity, ...]>(i))_{i in Non negative_
↳integers}

```

```

>>> from sage.all import *
>>> SteenrodAlgebra(profile=[Integer(1),Integer(2),Integer(1)]).gens()
Family (Sq(1), Sq(0,1), Sq(0,2), Sq(0,0,1))
>>> SteenrodAlgebra(p=Integer(5), profile=[[Integer(2),Integer(1)],
↳[Integer(2),Integer(2),Integer(2)]]).gens()
Family (Q_0, P(1), P(5))
>>> SteenrodAlgebra(profile=lambda n: n).gens()
Lazy family (<bound method SteenrodAlgebra_generic.gen of sub-Hopf algebra
of mod 2 Steenrod algebra, milnor basis, profile function [1, 2, 3, ...,
98, 99, +Infinity, +Infinity, +Infinity, ...]>(i))_{i in Non negative_
↳integers}

```

You may also use `algebra_generators` instead of `gens`:

```

sage: SteenrodAlgebra(p=5, profile=[[2,1], [2,2,2]]).algebra_generators()
Family (Q_0, P(1), P(5))

```

```

>>> from sage.all import *
>>> SteenrodAlgebra(p=Integer(5), profile=[[Integer(2),Integer(1)],
↳[Integer(2),Integer(2),Integer(2)]]).algebra_generators()
Family (Q_0, P(1), P(5))

```

`an_element()`

An element of this Steenrod algebra.

The element depends on the basis and whether there is a nontrivial profile function. (This is used by the automatic test suite, so having different elements in different bases may help in discovering bugs.)

EXAMPLES:

```
sage: SteenrodAlgebra().an_element()
Sq(2,1)
sage: SteenrodAlgebra(basis='adem').an_element()
Sq^4 Sq^2 Sq^1
sage: SteenrodAlgebra(p=5).an_element()
4 Q_1 Q_3 P(2,1)
sage: SteenrodAlgebra(basis='pst').an_element()
P^3_1
sage: SteenrodAlgebra(basis='pst', profile=[3,2,1]).an_element()
P^0_1
```

```
>>> from sage.all import *
>>> SteenrodAlgebra().an_element()
Sq(2,1)
>>> SteenrodAlgebra(basis='adem').an_element()
Sq^4 Sq^2 Sq^1
>>> SteenrodAlgebra(p=Integer(5)).an_element()
4 Q_1 Q_3 P(2,1)
>>> SteenrodAlgebra(basis='pst').an_element()
P^3_1
>>> SteenrodAlgebra(basis='pst', profile=[Integer(3), Integer(2), Integer(1)]).
↪an_element()
P^0_1
```

antipode_on_basis(*t*)

The antipode of a basis element of this algebra.

INPUT:

- *t* – tuple, the index of a basis element of *self*

OUTPUT:

the antipode of the corresponding basis element, as an element of *self*.

ALGORITHM: according to a result of Milnor's, the antipode of $Sq(n)$ is the sum of all of the Milnor basis elements in dimension n . So: convert the element to the Serre-Cartan basis, thus writing it as a sum of products of elements $Sq(n)$, and use Milnor's formula for the antipode of $Sq(n)$, together with the fact that the antipode is an antihomomorphism: if we call the antipode c , then $c(ab) = c(b)c(a)$.

At odd primes, a similar method is used: the antipode of $P(n)$ is the sum of the Milnor P basis elements in dimension $n * 2(p - 1)$, multiplied by $(-1)^n$, and the antipode of $\beta = Q_0$ is $-Q_0$. So convert to the Serre-Cartan basis, as in the $p = 2$ case. Note that in the odd prime case, there is a sign in the antihomomorphism formula: $c(ab) = (-1)^{\deg a \deg b} c(b)c(a)$.

EXAMPLES:

```
sage: A = SteenrodAlgebra()
sage: A.antipode_on_basis((4,))
Sq(1,1) + Sq(4)
sage: A.Sq(4).antipode()
Sq(1,1) + Sq(4)
```

(continues on next page)

(continued from previous page)

```

sage: Adem = SteenrodAlgebra(basis='adem')
sage: Adem.Sq(4).antipode()
Sq^3 Sq^1 + Sq^4
sage: SteenrodAlgebra(basis='pst').Sq(3).antipode()
P^0_1 P^1_1 + P^0_2
sage: a = SteenrodAlgebra(basis='wall_long').Sq(10)
sage: a.antipode()
Sq^1 Sq^2 Sq^4 Sq^1 Sq^2 + Sq^2 Sq^4 Sq^1 Sq^2 Sq^1 + Sq^8 Sq^2
sage: a.antipode().antipode() == a
True

sage: SteenrodAlgebra(p=3).P(6).antipode()
P(2,1) + P(6)
sage: SteenrodAlgebra(p=3).P(6).antipode().antipode()
P(6)

```

```

>>> from sage.all import *
>>> A = SteenrodAlgebra()
>>> A.antipode_on_basis((Integer(4),))
Sq(1,1) + Sq(4)
>>> A.Sq(Integer(4)).antipode()
Sq(1,1) + Sq(4)
>>> Adem = SteenrodAlgebra(basis='adem')
>>> Adem.Sq(Integer(4)).antipode()
Sq^3 Sq^1 + Sq^4
>>> SteenrodAlgebra(basis='pst').Sq(Integer(3)).antipode()
P^0_1 P^1_1 + P^0_2
>>> a = SteenrodAlgebra(basis='wall_long').Sq(Integer(10))
>>> a.antipode()
Sq^1 Sq^2 Sq^4 Sq^1 Sq^2 + Sq^2 Sq^4 Sq^1 Sq^2 Sq^1 + Sq^8 Sq^2
>>> a.antipode().antipode() == a
True

>>> SteenrodAlgebra(p=Integer(3)).P(Integer(6)).antipode()
P(2,1) + P(6)
>>> SteenrodAlgebra(p=Integer(3)).P(Integer(6)).antipode().antipode()
P(6)

```

basis ($d=None$)Return basis for `self`, either the whole basis or the basis in degree d .

INPUT:

- d – integer or None (default: None)

OUTPUT:

If d is None, then return a basis of the algebra. Otherwise, return the basis in degree d .

EXAMPLES:

```

sage: A3 = SteenrodAlgebra(3)
sage: A3.basis(13)
Family (Q_1 P(2), Q_0 P(3))

```

(continues on next page)

(continued from previous page)

```

sage: SteenrodAlgebra(2, 'adem').basis(12)
Family (Sq^12, Sq^11 Sq^1, Sq^9 Sq^2 Sq^1, Sq^8 Sq^3 Sq^1, Sq^10 Sq^2, Sq^9
↪Sq^3, Sq^8 Sq^4)

sage: A = SteenrodAlgebra(profile=[1,2,1])
sage: A.basis(2)
Family ()
sage: A.basis(3)
Family (Sq(0,1),)
sage: SteenrodAlgebra().basis(3)
Family (Sq(0,1), Sq(3))
sage: A_pst = SteenrodAlgebra(profile=[1,2,1], basis='pst')
sage: A_pst.basis(3)
Family (P^0_2,)

sage: A7 = SteenrodAlgebra(p=7)
sage: B = SteenrodAlgebra(p=7, profile=([1,2,1], [1]))
sage: A7.basis(84)
Family (P(7),)
sage: B.basis(84)
Family ()
sage: C = SteenrodAlgebra(p=7, profile=([1], [2,2]))
sage: A7.Q(0,1) in C.basis(14)
True
sage: A7.Q(2) in A7.basis(97)
True
sage: A7.Q(2) in C.basis(97)
False

```

```

>>> from sage.all import *
>>> A3 = SteenrodAlgebra(Integer(3))
>>> A3.basis(Integer(13))
Family (Q_1 P(2), Q_0 P(3))
>>> SteenrodAlgebra(Integer(2), 'adem').basis(Integer(12))
Family (Sq^12, Sq^11 Sq^1, Sq^9 Sq^2 Sq^1, Sq^8 Sq^3 Sq^1, Sq^10 Sq^2, Sq^9
↪Sq^3, Sq^8 Sq^4)

>>> A = SteenrodAlgebra(profile=[Integer(1),Integer(2),Integer(1)])
>>> A.basis(Integer(2))
Family ()
>>> A.basis(Integer(3))
Family (Sq(0,1),)
>>> SteenrodAlgebra().basis(Integer(3))
Family (Sq(0,1), Sq(3))
>>> A_pst = SteenrodAlgebra(profile=[Integer(1),Integer(2),Integer(1)], basis=
↪'pst')
>>> A_pst.basis(Integer(3))
Family (P^0_2,)

>>> A7 = SteenrodAlgebra(p=Integer(7))
>>> B = SteenrodAlgebra(p=Integer(7), profile=([Integer(1),Integer(2),
↪Integer(1)], [Integer(1)]))

```

(continues on next page)

(continued from previous page)

```

>>> A7.basis(Integer(84))
Family (P(7),)
>>> B.basis(Integer(84))
Family ()
>>> C = SteenrodAlgebra(p=Integer(7), profile=([Integer(1)], [Integer(2),
↳Integer(2)]))
>>> A7.Q(Integer(0),Integer(1)) in C.basis(Integer(14))
True
>>> A7.Q(Integer(2)) in A7.basis(Integer(97))
True
>>> A7.Q(Integer(2)) in C.basis(Integer(97))
False

```

With no arguments, return the basis of the whole algebra. This does not print in a very helpful way, unfortunately:

```

sage: A7.basis()
Lazy family (Term map from basis key family of mod 7 Steenrod algebra, milnor_
↳basis
to mod 7 Steenrod algebra, milnor basis(i))_{i in basis key family
of mod 7 Steenrod algebra, milnor basis}
sage: for (idx,a) in zip((1,..,9),A7.basis()):
.....:     print("{} {}".format(idx, a))
1 1
2 Q_0
3 P(1)
4 Q_1
5 Q_0 P(1)
6 Q_0 Q_1
7 P(2)
8 Q_1 P(1)
9 Q_0 P(2)
sage: D = SteenrodAlgebra(p=3, profile=([1], [2,2]))
sage: sorted(D.basis())
[1, P(1), P(2), Q_0, Q_0 P(1), Q_0 P(2), Q_0 Q_1,
Q_0 Q_1 P(1), Q_0 Q_1 P(2), Q_1, Q_1 P(1), Q_1 P(2)]

```

```

>>> from sage.all import *
>>> A7.basis()
Lazy family (Term map from basis key family of mod 7 Steenrod algebra, milnor_
↳basis
to mod 7 Steenrod algebra, milnor basis(i))_{i in basis key family
of mod 7 Steenrod algebra, milnor basis}
>>> for (idx,a) in zip((ellipsis_iter(Integer(1),Ellipsis,Integer(9))),A7.
↳basis()):
...     print("{} {}".format(idx, a))
1 1
2 Q_0
3 P(1)
4 Q_1
5 Q_0 P(1)
6 Q_0 Q_1

```

(continues on next page)

(continued from previous page)

```

7 P(2)
8 Q_1 P(1)
9 Q_0 P(2)
>>> D = SteenrodAlgebra(p=Integer(3), profile=([Integer(1)], [Integer(2),
↪Integer(2)]))
>>> sorted(D.basis())
[1, P(1), P(2), Q_0, Q_0 P(1), Q_0 P(2), Q_0 Q_1,
Q_0 Q_1 P(1), Q_0 Q_1 P(2), Q_1, Q_1 P(1), Q_1 P(2)]

```

basis_name()

The basis name associated to self.

EXAMPLES:

```

sage: SteenrodAlgebra(p=2, profile=[1,1]).basis_name()
'milnor'
sage: SteenrodAlgebra(basis='serre-cartan').basis_name()
'serre-cartan'
sage: SteenrodAlgebra(basis='adem').basis_name()
'serre-cartan'

```

```

>>> from sage.all import *
>>> SteenrodAlgebra(p=Integer(2), profile=[Integer(1),Integer(1)]).basis_
↪name()
'milnor'
>>> SteenrodAlgebra(basis='serre-cartan').basis_name()
'serre-cartan'
>>> SteenrodAlgebra(basis='adem').basis_name()
'serre-cartan'

```

coproduct (*x*, *algorithm*='milnor')

Return the coproduct of an element *x* of this algebra.

INPUT:

- *x* – element of self
- *algorithm* – None or a string, either 'milnor' or 'serre-cartan' (or anything which will be converted to one of these by the function `get_basis_name`. If None, default to 'serre-cartan' if current basis is 'serre-cartan'; otherwise use 'milnor'.

This calls `coproduct_on_basis()` on the summands of *x* and extends linearly.

EXAMPLES:

```

sage: SteenrodAlgebra().Sq(3).coproduct()
1 # Sq(3) + Sq(1) # Sq(2) + Sq(2) # Sq(1) + Sq(3) # 1

```

```

>>> from sage.all import *
>>> SteenrodAlgebra().Sq(Integer(3)).coproduct()
1 # Sq(3) + Sq(1) # Sq(2) + Sq(2) # Sq(1) + Sq(3) # 1

```

The element `Sq(0, 1)` is primitive:

```

sage: SteenrodAlgebra(basis='adem').Sq(0,1).coproduct()
1 # Sq^2 Sq^1 + 1 # Sq^3 + Sq^2 Sq^1 # 1 + Sq^3 # 1
sage: SteenrodAlgebra(basis='pst').Sq(0,1).coproduct()
1 # P^0_2 + P^0_2 # 1

sage: SteenrodAlgebra(p=3).P(4).coproduct()
1 # P(4) + P(1) # P(3) + P(2) # P(2) + P(3) # P(1) + P(4) # 1
sage: SteenrodAlgebra(p=3).P(4).coproduct(algorithm='serre-cartan')
1 # P(4) + P(1) # P(3) + P(2) # P(2) + P(3) # P(1) + P(4) # 1
sage: SteenrodAlgebra(p=3, basis='serre-cartan').P(4).coproduct()
1 # P^4 + P^1 # P^3 + P^2 # P^2 + P^3 # P^1 + P^4 # 1
sage: SteenrodAlgebra(p=11, profile=(), (2,1,2)).Q(0,2).coproduct()
1 # Q_0 Q_2 + Q_0 # Q_2 + Q_0 Q_2 # 1 + 10*Q_2 # Q_0

```

```

>>> from sage.all import *
>>> SteenrodAlgebra(basis='adem').Sq(Integer(0),Integer(1)).coproduct()
1 # Sq^2 Sq^1 + 1 # Sq^3 + Sq^2 Sq^1 # 1 + Sq^3 # 1
>>> SteenrodAlgebra(basis='pst').Sq(Integer(0),Integer(1)).coproduct()
1 # P^0_2 + P^0_2 # 1

>>> SteenrodAlgebra(p=Integer(3)).P(Integer(4)).coproduct()
1 # P(4) + P(1) # P(3) + P(2) # P(2) + P(3) # P(1) + P(4) # 1
>>> SteenrodAlgebra(p=Integer(3)).P(Integer(4)).coproduct(algorithm='serre-
↪cartan')
1 # P(4) + P(1) # P(3) + P(2) # P(2) + P(3) # P(1) + P(4) # 1
>>> SteenrodAlgebra(p=Integer(3), basis='serre-cartan').P(Integer(4)).
↪coproduct()
1 # P^4 + P^1 # P^3 + P^2 # P^2 + P^3 # P^1 + P^4 # 1
>>> SteenrodAlgebra(p=Integer(11), profile=(), (Integer(2),Integer(1),
↪Integer(2))).Q(Integer(0),Integer(2)).coproduct()
1 # Q_0 Q_2 + Q_0 # Q_2 + Q_0 Q_2 # 1 + 10*Q_2 # Q_0

```

coproduct_on_basis (*t*, *algorithm=None*)

The coproduct of a basis element of this algebra.

INPUT:

- *t* – tuple, the index of a basis element of self
- *algorithm* – None or a string, either ‘milnor’ or ‘serre-cartan’ (or anything which will be converted to one of these by the function `get_basis_name`. If None, default to ‘milnor’ unless current basis is ‘serre-cartan’, in which case use ‘serre-cartan’.

ALGORITHM: The coproduct on a Milnor basis element $P(n_1, n_2, \dots)$ is $\sum P(i_1, i_2, \dots) \otimes P(j_1, j_2, \dots)$, summed over all $i_k + j_k = n_k$ for each k . At odd primes, each element Q_n is primitive: its coproduct is $Q_n \otimes 1 + 1 \otimes Q_n$.

One can deduce a coproduct formula for the Serre-Cartan basis from this: the coproduct on each P^n is $\sum P^i \otimes P^{n-i}$ and at odd primes β is primitive. Since the coproduct is an algebra map, one can then compute the coproduct on any Serre-Cartan basis element.

Which of these methods is used is controlled by whether *algorithm* is ‘milnor’ or ‘serre-cartan’.

OUTPUT:

the coproduct of the corresponding basis element, as an element of `self` tensor `self`.

EXAMPLES:

```
sage: A = SteenrodAlgebra()
sage: A.coproduct_on_basis((3,))
1 # Sq(3) + Sq(1) # Sq(2) + Sq(2) # Sq(1) + Sq(3) # 1
```

```
>>> from sage.all import *
>>> A = SteenrodAlgebra()
>>> A.coproduct_on_basis((Integer(3),))
1 # Sq(3) + Sq(1) # Sq(2) + Sq(2) # Sq(1) + Sq(3) # 1
```

counit_on_basis(*t*)

The counit sends all elements of positive degree to zero.

INPUT:

- *t* – tuple, the index of a basis element of *self*

EXAMPLES:

```
sage: A2 = SteenrodAlgebra(p=2)
sage: A2.counit_on_basis(())
1
sage: A2.counit_on_basis((0,0,1))
0
sage: parent(A2.counit_on_basis((0,0,1)))
Finite Field of size 2
sage: A3 = SteenrodAlgebra(p=3)
sage: A3.counit_on_basis(((1,2,3), (1,1,1)))
0
sage: A3.counit_on_basis(((), ()))
1
sage: A3.counit(A3.P(10,5))
0
sage: A3.counit(A3.P(0))
1
```

```
>>> from sage.all import *
>>> A2 = SteenrodAlgebra(p=Integer(2))
>>> A2.counit_on_basis(())
1
>>> A2.counit_on_basis((Integer(0), Integer(0), Integer(1)))
0
>>> parent(A2.counit_on_basis((Integer(0), Integer(0), Integer(1))))
Finite Field of size 2
>>> A3 = SteenrodAlgebra(p=Integer(3))
>>> A3.counit_on_basis(((Integer(1), Integer(2), Integer(3)), (Integer(1),
↪ Integer(1), Integer(1))))
0
>>> A3.counit_on_basis(((), ()))
1
>>> A3.counit(A3.P(Integer(10), Integer(5)))
0
>>> A3.counit(A3.P(Integer(0)))
1
```

degree_on_basis(*t*)

The degree of the monomial specified by the tuple *t*.

INPUT:

- *t* – tuple, representing basis element in the current basis

OUTPUT: integer, the degree of the corresponding element

The degree of $Sq(i_1, i_2, i_3, \dots)$ is

$$i_1 + 3i_2 + 7i_3 + \dots + (2^k - 1)i_k + \dots$$

At an odd prime p , the degree of Q_k is $2p^k - 1$ and the degree of $\mathcal{P}(i_1, i_2, \dots)$ is

$$\sum_{k \geq 0} 2(p^k - 1)i_k.$$

ALGORITHM: Each basis element is represented in terms relevant to the particular basis: ‘milnor’ basis elements (at the prime 2) are given by tuples $(a, b, c, \dots)$ corresponding to the element $Sq(a, b, c, \dots)$, while ‘pst’ basis elements are given by tuples of pairs $((a, b), (c, d), \dots)$, corresponding to the product $P_b^a P_d^c \dots$. The other bases have similar descriptions. The degree of each basis element is computed from this data, rather than converting the element to the Milnor basis, for example, and then computing the degree.

EXAMPLES:

```
sage: SteenrodAlgebra().degree_on_basis((0,0,1))
7
sage: Sq(7).degree()
7

sage: A11 = SteenrodAlgebra(p=11)
sage: A11.degree_on_basis(((), (1,1)))
260
sage: A11.degree_on_basis(((2,), ()))
241
```

```
>>> from sage.all import *
>>> SteenrodAlgebra().degree_on_basis((Integer(0),Integer(0),Integer(1)))
7
>>> Sq(Integer(7)).degree()
7

>>> A11 = SteenrodAlgebra(p=Integer(11))
>>> A11.degree_on_basis(((), (Integer(1),Integer(1))))
260
>>> A11.degree_on_basis(((Integer(2),), ()))
241
```

dimension()

The dimension of this algebra as a vector space over $\mathbf{F}_p$.

If the algebra is infinite, return `+Infinity`. Otherwise, the profile function must be finite. In this case, at the prime 2, its dimension is 2^s , where s is the sum of the entries in the profile function. At odd primes, the dimension is $p^s * 2^t$ where s is the sum of the e component of the profile function and t is the number of 2's in the k component of the profile function.

EXAMPLES:

```

sage: SteenrodAlgebra(p=7).dimension()
+Infinity
sage: SteenrodAlgebra(profile=[3,2,1]).dimension()
64
sage: SteenrodAlgebra(p=3, profile=([1,1], [])).dimension()
9
sage: SteenrodAlgebra(p=5, profile=([1], [2,2])).dimension()
20

```

```

>>> from sage.all import *
>>> SteenrodAlgebra(p=Integer(7)).dimension()
+Infinity
>>> SteenrodAlgebra(profile=[Integer(3),Integer(2),Integer(1)]).dimension()
64
>>> SteenrodAlgebra(p=Integer(3), profile=([Integer(1),Integer(1)], [])).
↪dimension()
9
>>> SteenrodAlgebra(p=Integer(5), profile=([Integer(1)], [Integer(2),
↪Integer(2)])).dimension()
20

```

gen ($i=0$)

The i -th generator of this algebra.

INPUT:

- i – nonnegative integer

OUTPUT: the i -th generator of this algebra

For the full Steenrod algebra, the i -th generator is $Sq(2^i)$ at the prime 2; when p is odd, the 0-th generator is $\beta = Q(0)$, and for $i > 0$, the i -th generator is $P(p^{i-1})$.

For sub-Hopf algebras of the Steenrod algebra, it is not always clear what a minimal generating set is. The sub-Hopf algebra $A(n)$ is minimally generated by the elements Sq^{2^i} for $0 \leq i \leq n$ at the prime 2. At odd primes, $A(n)$ is minimally generated by Q_0 along with P^{p^i} for $0 \leq i \leq n-1$. So if this algebra is $A(n)$, return the appropriate generator.

For other sub-Hopf algebras: they are generated (but not necessarily minimally) by the P_t^s 's (and Q_n 's, if p is odd) that they contain. So order the P_t^s 's (and Q_n 's) in the algebra by degree and return the i -th one.

EXAMPLES:

```

sage: A = SteenrodAlgebra(2)
sage: A.gen(4)
Sq(16)
sage: A.gen(200)
Sq(1606938044258990275541962092341162602522202993782792835301376)
sage: SteenrodAlgebra(2, basis='adem').gen(2)
Sq^4
sage: SteenrodAlgebra(2, basis='pst').gen(2)
P^2_1
sage: B = SteenrodAlgebra(5)
sage: B.gen(0)
Q_0

```

(continues on next page)

(continued from previous page)

```

sage: B.gen(2)
P(5)

sage: SteenrodAlgebra(profile=[2,1]).gen(1)
Sq(2)
sage: SteenrodAlgebra(profile=[1,2,1]).gen(1)
Sq(0,1)
sage: SteenrodAlgebra(profile=[1,2,1]).gen(5)
Traceback (most recent call last):
...
ValueError: This algebra only has 4 generators, so call gen(i) with 0 <= i < 4

sage: D = SteenrodAlgebra(profile=lambda n: n)
sage: [D.gen(n) for n in range(5)]
[Sq(1), Sq(0,1), Sq(0,2), Sq(0,0,1), Sq(0,0,2)]
sage: D3 = SteenrodAlgebra(p=3, profile=(lambda n: n, lambda n: 2))
sage: [D3.gen(n) for n in range(9)]
[Q_0, P(1), Q_1, P(0,1), Q_2, P(0,3), P(0,0,1), Q_3, P(0,0,3)]
sage: D3 = SteenrodAlgebra(p=3, profile=(lambda n: n, lambda n: 1 if n<1 else_
↳2))
sage: [D3.gen(n) for n in range(9)]
[P(1), Q_1, P(0,1), Q_2, P(0,3), P(0,0,1), Q_3, P(0,0,3), P(0,0,0,1)]
sage: SteenrodAlgebra(p=5, profile=[[2,1], [2,2,2]], basis='pst').gen(2)
P^1_1

```

```

>>> from sage.all import *
>>> A = SteenrodAlgebra(Integer(2))
>>> A.gen(Integer(4))
Sq(16)
>>> A.gen(Integer(200))
Sq(1606938044258990275541962092341162602522202993782792835301376)
>>> SteenrodAlgebra(Integer(2), basis='adem').gen(Integer(2))
Sq^4
>>> SteenrodAlgebra(Integer(2), basis='pst').gen(Integer(2))
P^2_1
>>> B = SteenrodAlgebra(Integer(5))
>>> B.gen(Integer(0))
Q_0
>>> B.gen(Integer(2))
P(5)

>>> SteenrodAlgebra(profile=[Integer(2),Integer(1)]).gen(Integer(1))
Sq(2)
>>> SteenrodAlgebra(profile=[Integer(1),Integer(2),Integer(1)]).
↳gen(Integer(1))
Sq(0,1)
>>> SteenrodAlgebra(profile=[Integer(1),Integer(2),Integer(1)]).
↳gen(Integer(5))
Traceback (most recent call last):
...
ValueError: This algebra only has 4 generators, so call gen(i) with 0 <= i < 4

```

(continues on next page)

(continued from previous page)

```

>>> D = SteenrodAlgebra(profile=lambda n: n)
>>> [D.gen(n) for n in range(Integer(5))]
[Sq(1), Sq(0,1), Sq(0,2), Sq(0,0,1), Sq(0,0,2)]
>>> D3 = SteenrodAlgebra(p=Integer(3), profile=(lambda n: n, lambda n:
↳ Integer(2)))
>>> [D3.gen(n) for n in range(Integer(9))]
[Q_0, P(1), Q_1, P(0,1), Q_2, P(0,3), P(0,0,1), Q_3, P(0,0,3)]
>>> D3 = SteenrodAlgebra(p=Integer(3), profile=(lambda n: n, lambda n:
↳ Integer(1) if n<Integer(1) else Integer(2)))
>>> [D3.gen(n) for n in range(Integer(9))]
[P(1), Q_1, P(0,1), Q_2, P(0,3), P(0,0,1), Q_3, P(0,0,3), P(0,0,0,1)]
>>> SteenrodAlgebra(p=Integer(5), profile=[[Integer(2), Integer(1)],
↳ [Integer(2), Integer(2), Integer(2)]], basis='pst').gen(Integer(2))
P^1_1

```

gens()

Family of generators for this algebra.

OUTPUT: family of elements of this algebra

At the prime 2, the Steenrod algebra is generated by the elements Sq^{2^i} for $i \geq 0$. At odd primes, it is generated by the elements Q_0 and P^{p^i} for $i \geq 0$. So if this algebra is the entire Steenrod algebra, return an infinite family made up of these elements.

For sub-Hopf algebras of the Steenrod algebra, it is not always clear what a minimal generating set is. The sub-Hopf algebra $A(n)$ is minimally generated by the elements Sq^{2^i} for $0 \leq i \leq n$ at the prime 2. At odd primes, $A(n)$ is minimally generated by Q_0 along with P^{p^i} for $0 \leq i \leq n-1$. So if this algebra is $A(n)$, return the appropriate list of generators.

For other sub-Hopf algebras: return a non-minimal generating set: the family of P_t^s 's and Q_n 's contained in the algebra.

EXAMPLES:

```

sage: A3 = SteenrodAlgebra(3, 'adem')
sage: A3.gens()
Lazy family (<bound method SteenrodAlgebra_generic.gen of mod 3 Steenrod_
↳ algebra,
  serre-cartan basis>(i))_{i in Non negative integers}
sage: A3.gens()[0]
beta
sage: A3.gens()[1]
P^1
sage: A3.gens()[2]
P^3
sage: SteenrodAlgebra(profile=[3,2,1]).gens()
Family (Sq(1), Sq(2), Sq(4))

```

```

>>> from sage.all import *
>>> A3 = SteenrodAlgebra(Integer(3), 'adem')
>>> A3.gens()
Lazy family (<bound method SteenrodAlgebra_generic.gen of mod 3 Steenrod_
↳ algebra,
  serre-cartan basis>(i))_{i in Non negative integers}

```

(continues on next page)

(continued from previous page)

```

>>> A3.gens() [Integer(0)]
beta
>>> A3.gens() [Integer(1)]
P^1
>>> A3.gens() [Integer(2)]
P^3
>>> SteenrodAlgebra(profile=[Integer(3), Integer(2), Integer(1)]).gens()
Family (Sq(1), Sq(2), Sq(4))

```

In the following case, return a non-minimal generating set. (It is not minimal because $Sq(0,0,1)$ is the commutator of $Sq(1)$ and $Sq(0,2)$.)

```

sage: SteenrodAlgebra(profile=[1,2,1]).gens()
Family (Sq(1), Sq(0,1), Sq(0,2), Sq(0,0,1))
sage: SteenrodAlgebra(p=5, profile=[[2,1], [2,2,2]]).gens()
Family (Q_0, P(1), P(5))
sage: SteenrodAlgebra(profile=lambda n: n).gens()
Lazy family (<bound method SteenrodAlgebra_generic.gen of sub-Hopf algebra
of mod 2 Steenrod algebra, milnor basis, profile function [1, 2, 3, ...,
98, 99, +Infinity, +Infinity, +Infinity, ...]>(i))_{i in Non negative
↳integers}

```

```

>>> from sage.all import *
>>> SteenrodAlgebra(profile=[Integer(1), Integer(2), Integer(1)]).gens()
Family (Sq(1), Sq(0,1), Sq(0,2), Sq(0,0,1))
>>> SteenrodAlgebra(p=Integer(5), profile=[[Integer(2), Integer(1)],
↳[Integer(2), Integer(2), Integer(2)]]).gens()
Family (Q_0, P(1), P(5))
>>> SteenrodAlgebra(profile=lambda n: n).gens()
Lazy family (<bound method SteenrodAlgebra_generic.gen of sub-Hopf algebra
of mod 2 Steenrod algebra, milnor basis, profile function [1, 2, 3, ...,
98, 99, +Infinity, +Infinity, +Infinity, ...]>(i))_{i in Non negative
↳integers}

```

You may also use `algebra_generators` instead of `gens`:

```

sage: SteenrodAlgebra(p=5, profile=[[2,1], [2,2,2]]).algebra_generators()
Family (Q_0, P(1), P(5))

```

```

>>> from sage.all import *
>>> SteenrodAlgebra(p=Integer(5), profile=[[Integer(2), Integer(1)],
↳[Integer(2), Integer(2), Integer(2)]]).algebra_generators()
Family (Q_0, P(1), P(5))

```

homogeneous_component (*n*)

Return the *n*-th homogeneous piece of the Steenrod algebra.

INPUT:

- *n* – integer

OUTPUT: a vector space spanned by the basis for this algebra in dimension *n*

EXAMPLES:

```
sage: A = SteenrodAlgebra()
sage: A.homogeneous_component(4)
Vector space spanned by (Sq(1,1), Sq(4)) over Finite Field of size 2
sage: SteenrodAlgebra(profile=[2,1,0]).homogeneous_component(4)
Vector space spanned by (Sq(1,1),) over Finite Field of size 2
```

```
>>> from sage.all import *
>>> A = SteenrodAlgebra()
>>> A.homogeneous_component(Integer(4))
Vector space spanned by (Sq(1,1), Sq(4)) over Finite Field of size 2
>>> SteenrodAlgebra(profile=[Integer(2),Integer(1),Integer(0)]).homogeneous_
↪component(Integer(4))
Vector space spanned by (Sq(1,1),) over Finite Field of size 2
```

The notation $A[n]$ may also be used:

```
sage: A[5]
Vector space spanned by (Sq(2,1), Sq(5)) over Finite Field of size 2
sage: SteenrodAlgebra(basis='wall')[4]
Vector space spanned by ( $Q^1_0 Q^0_0$ ,  $Q^2_2$ ) over Finite Field of size 2
sage: SteenrodAlgebra(p=5)[17]
Vector space spanned by ( $Q_1 P(1)$ ,  $Q_0 P(2)$ ) over Finite Field of size 5
```

```
>>> from sage.all import *
>>> A[Integer(5)]
Vector space spanned by (Sq(2,1), Sq(5)) over Finite Field of size 2
>>> SteenrodAlgebra(basis='wall')[Integer(4)]
Vector space spanned by ( $Q^1_0 Q^0_0$ ,  $Q^2_2$ ) over Finite Field of size 2
>>> SteenrodAlgebra(p=Integer(5))[Integer(17)]
Vector space spanned by ( $Q_1 P(1)$ ,  $Q_0 P(2)$ ) over Finite Field of size 5
```

Note that $A[n]$ is just a vector space, not a Hopf algebra, so its elements don't have products, coproducts, or antipodes defined on them. If you want to use operations like this on elements of some $A[n]$, then convert them back to elements of A :

```
sage: sorted(A[5].basis())
[milnor[(2, 1)], milnor[(5,)]]
sage: a = list(A[5].basis())[1]
sage: a # not in A, doesn't print like an element of A
milnor[(5,)]
sage: A(a) # in A
Sq(5)
sage: A(a) * A(a)
Sq(7,1)
sage: a * A(a) # only need to convert one factor
Sq(7,1)
sage: a.antipode() # not defined
Traceback (most recent call last):
...
AttributeError: 'CombinatorialFreeModule_with_category.element_class' object
↪has no attribute 'antipode'...
sage: A(a).antipode() # convert to elt of A, then compute antipode
Sq(2,1) + Sq(5)
```

(continues on next page)

(continued from previous page)

```
sage: G = SteenrodAlgebra(p=5, profile=[[2,1], [2,2,2]], basis='pst')
```

```
>>> from sage.all import *
>>> sorted(A[Integer(5)].basis())
[milnor[(2, 1)], milnor[(5,)]]
>>> a = list(A[Integer(5)].basis())[Integer(1)]
>>> a # not in A, doesn't print like an element of A
milnor[(5,)]
>>> A(a) # in A
Sq(5)
>>> A(a) * A(a)
Sq(7,1)
>>> a * A(a) # only need to convert one factor
Sq(7,1)
>>> a.antipode() # not defined
Traceback (most recent call last):
...
AttributeError: 'CombinatorialFreeModule_with_category.element_class' object
↳ has no attribute 'antipode'...
>>> A(a).antipode() # convert to elt of A, then compute antipode
Sq(2,1) + Sq(5)

>>> G = SteenrodAlgebra(p=Integer(5), profile=[[Integer(2),Integer(1)],
↳ [Integer(2),Integer(2),Integer(2)]], basis='pst')
```

`is_commutative()`

Return True if self is graded commutative, as determined by the profile function.

In particular, a sub-Hopf algebra of the mod 2 Steenrod algebra is commutative if and only if there is an integer $n > 0$ so that its profile function e satisfies

- $e(i) = 0$ for $i < n$,
- $e(i) \leq n$ for $i \geq n$.

When p is odd, there must be an integer $n \geq 0$ so that the profile functions e and k satisfy

- $e(i) = 0$ for $i < n$,
- $e(i) \leq n$ for $i \geq n$.
- $k(i) = 1$ for $i < n$.

EXAMPLES:

```
sage: A = SteenrodAlgebra(p=3)
sage: A.is_commutative()
False
sage: SteenrodAlgebra(profile=[2,1]).is_commutative()
False
sage: SteenrodAlgebra(profile=[0,2,2,1]).is_commutative()
True
```

```

>>> from sage.all import *
>>> A = SteenrodAlgebra(p=Integer(3))
>>> A.is_commutative()
False
>>> SteenrodAlgebra(profile=[Integer(2),Integer(1)]).is_commutative()
False
>>> SteenrodAlgebra(profile=[Integer(0),Integer(2),Integer(2),Integer(1)]).is_
↳commutative()
True

```

Note that if the profile function is specified by a function, then by default it has infinite truncation type: the profile function is assumed to be infinite after the 100th term.

```

sage: SteenrodAlgebra(profile=lambda n: 1).is_commutative()
False
sage: SteenrodAlgebra(profile=lambda n: 1, truncation_type=0).is_commutative()
True

sage: SteenrodAlgebra(p=5, profile=([0,2,2,1], [])).is_commutative()
True
sage: SteenrodAlgebra(p=5, profile=([0,2,2,1], [1,1,2])).is_commutative()
True
sage: SteenrodAlgebra(p=5, profile=([0,2,1], [1,2,2,2])).is_commutative()
False

```

```

>>> from sage.all import *
>>> SteenrodAlgebra(profile=lambda n: Integer(1)).is_commutative()
False
>>> SteenrodAlgebra(profile=lambda n: Integer(1), truncation_type=Integer(0)).
↳is_commutative()
True

>>> SteenrodAlgebra(p=Integer(5), profile=([Integer(0),Integer(2),Integer(2),
↳Integer(1)], [])).is_commutative()
True
>>> SteenrodAlgebra(p=Integer(5), profile=([Integer(0),Integer(2),Integer(2),
↳Integer(1)], [Integer(1),Integer(1),Integer(2)])).is_commutative()
True
>>> SteenrodAlgebra(p=Integer(5), profile=([Integer(0),Integer(2),Integer(1)],
↳[Integer(1),Integer(2),Integer(2),Integer(2)])).is_commutative()
False

```

`is_division_algebra()`

The only way this algebra can be a division algebra is if it is the ground field $\mathbf{F}_p$.

EXAMPLES:

```

sage: SteenrodAlgebra(11).is_division_algebra()
False
sage: SteenrodAlgebra(profile=lambda n: 0, truncation_type=0).is_division_
↳algebra()
True

```

```
>>> from sage.all import *
>>> SteenrodAlgebra(Integer(11)).is_division_algebra()
False
>>> SteenrodAlgebra(profile=lambda n: Integer(0), truncation_type=Integer(0)).
↳ is_division_algebra()
True
```

is_field(*proof=True*)

The only way this algebra can be a field is if it is the ground field $\mathbf{F}_p$.

EXAMPLES:

```
sage: SteenrodAlgebra(11).is_field()
False
sage: SteenrodAlgebra(profile=lambda n: 0, truncation_type=0).is_field()
True
```

```
>>> from sage.all import *
>>> SteenrodAlgebra(Integer(11)).is_field()
False
>>> SteenrodAlgebra(profile=lambda n: Integer(0), truncation_type=Integer(0)).
↳ is_field()
True
```

is_finite()

Return True if this algebra is finite-dimensional.

Therefore true if the profile function is finite, and in particular the `truncation_type` must be finite.

EXAMPLES:

```
sage: A = SteenrodAlgebra(p=3)
sage: A.is_finite()
False
sage: SteenrodAlgebra(profile=[3,2,1]).is_finite()
True
sage: SteenrodAlgebra(profile=lambda n: n).is_finite()
False
```

```
>>> from sage.all import *
>>> A = SteenrodAlgebra(p=Integer(3))
>>> A.is_finite()
False
>>> SteenrodAlgebra(profile=[Integer(3), Integer(2), Integer(1)]).is_finite()
True
>>> SteenrodAlgebra(profile=lambda n: n).is_finite()
False
```

is_generic()

The algebra is generic if it is based on the odd-primary relations, i.e. if its dual is a quotient of

$$A_* = \mathbf{F}_p[\xi_1, \xi_2, \xi_3, \dots] \otimes \Lambda(\tau_0, \tau_1, \dots)$$

Sage also allows this for $p = 2$. Only the usual Steenrod algebra at the prime 2 and its sub algebras are non-generic.

EXAMPLES:

```
sage: SteenrodAlgebra(3).is_generic()
True
sage: SteenrodAlgebra(2).is_generic()
False
sage: SteenrodAlgebra(2, generic=True).is_generic()
True
```

```
>>> from sage.all import *
>>> SteenrodAlgebra(Integer(3)).is_generic()
True
>>> SteenrodAlgebra(Integer(2)).is_generic()
False
>>> SteenrodAlgebra(Integer(2), generic=True).is_generic()
True
```

is_integral_domain(*proof=True*)

The only way this algebra can be an integral domain is if it is the ground field $\mathbb{F}_p$.

EXAMPLES:

```
sage: SteenrodAlgebra(11).is_integral_domain()
False
sage: SteenrodAlgebra(profile=lambda n: 0, truncation_type=0).is_integral_
↳domain()
True
```

```
>>> from sage.all import *
>>> SteenrodAlgebra(Integer(11)).is_integral_domain()
False
>>> SteenrodAlgebra(profile=lambda n: Integer(0), truncation_type=Integer(0)).
↳is_integral_domain()
True
```

is_noetherian()

This algebra is Noetherian if and only if it is finite.

EXAMPLES:

```
sage: SteenrodAlgebra(3).is_noetherian()
False
sage: SteenrodAlgebra(profile=[1,2,1]).is_noetherian()
True
sage: SteenrodAlgebra(profile=lambda n: n+2).is_noetherian()
False
```

```
>>> from sage.all import *
>>> SteenrodAlgebra(Integer(3)).is_noetherian()
False
>>> SteenrodAlgebra(profile=[Integer(1), Integer(2), Integer(1)]).is_
↳noetherian()
True
```

(continues on next page)

(continued from previous page)

```
>>> SteenrodAlgebra(profile=lambda n: n+Integer(2)).is_noetherian()
False
```

milnor()

Convert an element of this algebra to the Milnor basis.

INPUT:

- x – an element of this algebra

OUTPUT: x converted to the Milnor basisALGORITHM: use the method `_milnor_on_basis` and linearity.

EXAMPLES:

```
sage: Adem = SteenrodAlgebra(basis='adem')
sage: a = Adem.Sq(2) * Adem.Sq(1)
sage: Adem.milnor(a)
Sq(0,1) + Sq(3)
```

```
>>> from sage.all import *
>>> Adem = SteenrodAlgebra(basis='adem')
>>> a = Adem.Sq(Integer(2)) * Adem.Sq(Integer(1))
>>> Adem.milnor(a)
Sq(0,1) + Sq(3)
```

ngens()Number of generators of `self`.

OUTPUT: number or Infinity

The Steenrod algebra is infinitely generated. A sub-Hopf algebra may be finitely or infinitely generated; in general, it is not clear what a minimal generating set is, nor the cardinality of that set. So: if the algebra is infinite-dimensional, this returns Infinity. If the algebra is finite-dimensional and is equal to one of the sub-Hopf algebras $A(n)$, then their minimal generating set is known, and this returns the cardinality of that set. Otherwise, any sub-Hopf algebra is (not necessarily minimally) generated by the P_t^s 's that it contains (along with the Q_n 's it contains, at odd primes), so this returns the number of P_t^s 's and Q_n 's in the algebra.

EXAMPLES:

```
sage: A = SteenrodAlgebra(3)
sage: A.ngens()
+Infinity
sage: SteenrodAlgebra(profile=lambda n: n).ngens()
+Infinity
sage: SteenrodAlgebra(profile=[3,2,1]).ngens() # A(2)
3
sage: SteenrodAlgebra(profile=[3,2,1], basis='pst').ngens()
3
sage: SteenrodAlgebra(p=3, profile=[[3,2,1], [2,2,2,2]]).ngens() # A(3) at p=3
4
sage: SteenrodAlgebra(profile=[1,2,1,1]).ngens()
5
```

```

>>> from sage.all import *
>>> A = SteenrodAlgebra(Integer(3))
>>> A.ngens()
+Infinity
>>> SteenrodAlgebra(profile=lambda n: n).ngens()
+Infinity
>>> SteenrodAlgebra(profile=[Integer(3), Integer(2), Integer(1)]).ngens() # A(2)
3
>>> SteenrodAlgebra(profile=[Integer(3), Integer(2), Integer(1)], basis='pst').
↪ngens()
3
>>> SteenrodAlgebra(p=Integer(3), profile=[[Integer(3), Integer(2), Integer(1)],
↪ [Integer(2), Integer(2), Integer(2), Integer(2)]]).ngens() # A(3) at p=3
4
>>> SteenrodAlgebra(profile=[Integer(1), Integer(2), Integer(1), Integer(1)]).
↪ngens()
5

```

one_basis()

The index of the element 1 in the basis for the Steenrod algebra.

EXAMPLES:

```

sage: SteenrodAlgebra(p=2).one_basis()
()
sage: SteenrodAlgebra(p=7).one_basis()
((), ())

```

```

>>> from sage.all import *
>>> SteenrodAlgebra(p=Integer(2)).one_basis()
()
>>> SteenrodAlgebra(p=Integer(7)).one_basis()
((), ())

```

order()

The order of this algebra.

This is computed by computing its vector space dimension d and then returning p^d .

EXAMPLES:

```

sage: SteenrodAlgebra(p=7).order()
+Infinity
sage: SteenrodAlgebra(profile=[2,1]).dimension()
8
sage: SteenrodAlgebra(profile=[2,1]).order()
256
sage: SteenrodAlgebra(p=3, profile=([1], [])).dimension()
3
sage: SteenrodAlgebra(p=3, profile=([1], [])).order()
27
sage: SteenrodAlgebra(p=5, profile=([], [2, 2])).dimension()
4

```

(continues on next page)

(continued from previous page)

```
sage: SteenrodAlgebra(p=5, profile=([], [2, 2])).order() == 5**4
True
```

```
>>> from sage.all import *
>>> SteenrodAlgebra(p=Integer(7)).order()
+Infinity
>>> SteenrodAlgebra(profile=[Integer(2), Integer(1)]).dimension()
8
>>> SteenrodAlgebra(profile=[Integer(2), Integer(1)]).order()
256
>>> SteenrodAlgebra(p=Integer(3), profile=[Integer(1)], []).dimension()
3
>>> SteenrodAlgebra(p=Integer(3), profile=[Integer(1)], []).order()
27
>>> SteenrodAlgebra(p=Integer(5), profile=([], [Integer(2), Integer(2)]))
↳dimension()
4
>>> SteenrodAlgebra(p=Integer(5), profile=([], [Integer(2), Integer(2)]))
↳order() == Integer(5)**Integer(4)
True
```

prime()

The prime associated to self.

EXAMPLES:

```
sage: SteenrodAlgebra(p=2, profile=[1, 1]).prime()
2
sage: SteenrodAlgebra(p=7).prime()
7
```

```
>>> from sage.all import *
>>> SteenrodAlgebra(p=Integer(2), profile=[Integer(1), Integer(1)]).prime()
2
>>> SteenrodAlgebra(p=Integer(7)).prime()
7
```

product_on_basis(t1, t2)

The product of two basis elements of this algebra.

INPUT:

- t_1, t_2 – tuples, the indices of two basis elements of self

OUTPUT:

the product of the two corresponding basis elements, as an element of self

ALGORITHM: If the two elements are represented in the Milnor basis, use Milnor multiplication as implemented in `sage.algebras.steenrod.steenrod_algebra_mult`. If the two elements are represented in the Serre-Cartan basis, then multiply them using Adem relations (also implemented in `sage.algebras.steenrod.steenrod_algebra_mult`). This provides a good way of checking work – multiply Milnor elements, then convert them to Adem elements and multiply those, and see if the answers correspond.

If the two elements are represented in some other basis, then convert them both to the Milnor basis and multiply.

EXAMPLES:

```
sage: Milnor = SteenrodAlgebra()
sage: Milnor.product_on_basis((2,), (2,))
Sq(1,1)
sage: Adem = SteenrodAlgebra(basis='adem')
sage: Adem.Sq(2) * Adem.Sq(2) # indirect doctest
Sq^3 Sq^1
```

```
>>> from sage.all import *
>>> Milnor = SteenrodAlgebra()
>>> Milnor.product_on_basis((Integer(2),), (Integer(2),))
Sq(1,1)
>>> Adem = SteenrodAlgebra(basis='adem')
>>> Adem.Sq(Integer(2)) * Adem.Sq(Integer(2)) # indirect doctest
Sq^3 Sq^1
```

When multiplying elements from different bases, the left-hand factor determines the form of the output:

```
sage: Adem.Sq(2) * Milnor.Sq(2)
Sq^3 Sq^1
sage: Milnor.Sq(2) * Adem.Sq(2)
Sq(1,1)
```

```
>>> from sage.all import *
>>> Adem.Sq(Integer(2)) * Milnor.Sq(Integer(2))
Sq^3 Sq^1
>>> Milnor.Sq(Integer(2)) * Adem.Sq(Integer(2))
Sq(1,1)
```

profile (*i*, *component*=0)

Profile function for this algebra.

INPUT:

- *i* – integer
- *component* – either 0 or 1 (default: 0)

OUTPUT: integer or ∞

See the documentation for `sage.algebras.steenrod.steenrod_algebra` and `SteenrodAlgebra()` for information on profile functions.

This applies the profile function to the integer *i*. Thus when $p = 2$, *i* must be a positive integer. When p is odd, there are two profile functions, *e* and *k* (in the notation of the aforementioned documentation), corresponding, respectively to *component*=0 and *component*=1. So when p is odd and *component* is 0, *i* must be positive, while when *component* is 1, *i* must be nonnegative.

EXAMPLES:

```
sage: SteenrodAlgebra().profile(3)
+Infinity
sage: SteenrodAlgebra(profile=[3,2,1]).profile(1)
3
sage: SteenrodAlgebra(profile=[3,2,1]).profile(2)
2
```

```

>>> from sage.all import *
>>> SteenrodAlgebra().profile(Integer(3))
+Infinity
>>> SteenrodAlgebra(profile=[Integer(3), Integer(2), Integer(1)]).
↳profile(Integer(1))
3
>>> SteenrodAlgebra(profile=[Integer(3), Integer(2), Integer(1)]).
↳profile(Integer(2))
2

```

When the profile is specified by a list, the default behavior is to return zero values outside the range of the list. This can be overridden if the algebra is created with an infinite `truncation_type`:

```

sage: SteenrodAlgebra(profile=[3,2,1]).profile(9)
0
sage: SteenrodAlgebra(profile=[3,2,1], truncation_type=Infinity).profile(9)
+Infinity

sage: B = SteenrodAlgebra(p=3, profile=(lambda n: n, lambda n: 1))
sage: B.profile(3)
3
sage: B.profile(3, component=1)
1

sage: EA = SteenrodAlgebra(generic=True, profile=(lambda n: n, lambda n: 1))
sage: EA.profile(4)
4
sage: EA.profile(2, component=1)
1

```

```

>>> from sage.all import *
>>> SteenrodAlgebra(profile=[Integer(3), Integer(2), Integer(1)]).
↳profile(Integer(9))
0
>>> SteenrodAlgebra(profile=[Integer(3), Integer(2), Integer(1)], truncation_
↳type=Infinity).profile(Integer(9))
+Infinity

>>> B = SteenrodAlgebra(p=Integer(3), profile=(lambda n: n, lambda n:
↳Integer(1)))
>>> B.profile(Integer(3))
3
>>> B.profile(Integer(3), component=Integer(1))
1

>>> EA = SteenrodAlgebra(generic=True, profile=(lambda n: n, lambda n:
↳Integer(1)))
>>> EA.profile(Integer(4))
4
>>> EA.profile(Integer(2), component=Integer(1))
1

```

$\text{pst}(s, t)$

The Margolis element P_t^s .

INPUT:

- s – nonnegative integer
- t – positive integer
- p – positive prime number

OUTPUT: element of the Steenrod algebra

This returns the Margolis element P_t^s of the mod p Steenrod algebra: the element equal to $P(0, 0, \dots, 0, p^s)$, where the p^s is in position t .

EXAMPLES:

```
sage: A2 = SteenrodAlgebra(2)
sage: A2.pst(3,5)
Sq(0,0,0,0,8)
sage: A2.pst(1,2) == Sq(4)*Sq(2) + Sq(2)*Sq(4)
True
sage: SteenrodAlgebra(5).pst(3,5)
P(0,0,0,0,125)
```

```
>>> from sage.all import *
>>> A2 = SteenrodAlgebra(Integer(2))
>>> A2.pst(Integer(3), Integer(5))
Sq(0,0,0,0,8)
>>> A2.pst(Integer(1), Integer(2)) == Sq(Integer(4))*Sq(Integer(2)) +
↳ Sq(Integer(2))*Sq(Integer(4))
True
>>> SteenrodAlgebra(Integer(5)).pst(Integer(3), Integer(5))
P(0,0,0,0,125)
```

top_class()

Highest dimensional basis element. This is only defined if the algebra is finite.

EXAMPLES:

```
sage: SteenrodAlgebra(2, profile=(3,2,1)).top_class()
Sq(7,3,1)
sage: SteenrodAlgebra(3, profile=((2,2,1), (1,2,2,2,2))).top_class()
Q_1 Q_2 Q_3 Q_4 P(8,8,2)
```

```
>>> from sage.all import *
>>> SteenrodAlgebra(Integer(2), profile=(Integer(3), Integer(2), Integer(1))).
↳ top_class()
Sq(7,3,1)
>>> SteenrodAlgebra(Integer(3), profile=((Integer(2), Integer(2), Integer(1)),
↳ (Integer(1), Integer(2), Integer(2), Integer(2), Integer(2)))).top_class()
Q_1 Q_2 Q_3 Q_4 P(8,8,2)
```

```
class sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_mod_two(p=2, basis='milnor',
**kws)
```

Bases: *SteenrodAlgebra_generic*

The mod 2 Steenrod algebra.

Users should not call this, but use the function `SteenrodAlgebra()` instead. See that function for extensive documentation. (This differs from `SteenrodAlgebra_generic` only in that it has a method `Sq()` for defining elements.)

Sq(**nums*)

Milnor element $Sq(a, b, c, \dots)$.

INPUT:

- $a, b, c, \dots$ – nonnegative integers

OUTPUT: element of the Steenrod algebra

This returns the Milnor basis element $Sq(a, b, c, \dots)$.

EXAMPLES:

```
sage: A = SteenrodAlgebra(2)
sage: A.Sq(5)
Sq(5)
sage: A.Sq(5, 0, 2)
Sq(5, 0, 2)
```

```
>>> from sage.all import *
>>> A = SteenrodAlgebra(Integer(2))
>>> A.Sq(Integer(5))
Sq(5)
>>> A.Sq(Integer(5), Integer(0), Integer(2))
Sq(5, 0, 2)
```

Entries must be nonnegative integers; otherwise, an error results.

5.25 Steenrod algebra bases

AUTHORS:

- John H. Palmieri (2008-07-30): version 0.9
- John H. Palmieri (2010-06-30): version 1.0
- Simon King (2011-10-25): Fix the use of cached functions

This package defines functions for computing various bases of the Steenrod algebra, and for converting between the Milnor basis and any other basis.

This packages implements a number of different bases, at least at the prime 2. The Milnor and Serre-Cartan bases are the most familiar and most standard ones, and all of the others are defined in terms of one of these. The bases are described in the documentation for the function `steenrod_algebra_basis()`; also see the papers by Monks [Mon1998] and Wood [Woo1998] for more information about them. For commutator bases, see the preprint by Palmieri and Zhang [PZ2008].

- 'milnor' – Milnor basis
- 'serre-cartan' or 'adem' or 'admissible' – Serre-Cartan basis

Most of the rest of the bases are only defined when $p = 2$. The only exceptions are the P_t^s -bases and the commutator bases, which are defined at all primes.

- 'wood_y' – Wood's Y basis
- 'wood_z' – Wood's Z basis

- 'wall', 'wall_long' – Wall's basis
- 'arnon_a', 'arnon_a_long' – Arnon's A basis
- 'arnon_c' – Arnon's C basis
- 'pst', 'pst_rlex', 'pst_llex', 'pst_deg', 'pst_revz' – various P_t^s -bases
- 'comm', 'comm_rlex', 'comm_llex', 'comm_deg', 'comm_revz',
or these with '_long' appended – various commutator bases

The main functions provided here are

- `steenrod_algebra_basis()`. This computes a tuple representing basis elements for the Steenrod algebra in a given degree, at a given prime, with respect to a given basis. It is a cached function.
- `convert_to_milnor_matrix()`. This returns the change-of-basis matrix, in a given degree, from any basis to the Milnor basis. It is a cached function.
- `convert_from_milnor_matrix()`. This returns the inverse of the previous matrix.

INTERNAL DOCUMENTATION:

If you want to implement a new basis for the Steenrod algebra:

In the file `steenrod_algebra.py`:

For the class `SteenrodAlgebra_generic`, add functionality to the methods:

- `_repr_term`
- `degree_on_basis`
- `_milnor_on_basis`
- `an_element`

In the file `steenrod_algebra_misc.py`:

- add functionality to `get_basis_name`: this should accept as input various synonyms for the basis, and its output should be a canonical name for the basis.
- add a function `BASIS_mono_to_string` like `milnor_mono_to_string` or one of the other similar functions.

In this file `steenrod_algebra_bases.py`:

- add appropriate lines to `steenrod_algebra_basis()`.
- add a function to compute the basis in a given dimension (to be called by `steenrod_algebra_basis()`).
- modify `steenrod_basis_error_check()` so it checks the new basis.

If the basis has an intrinsic way of defining a product, implement it in the file `steenrod_algebra_mult.py` and also in the `product_on_basis` method for `SteenrodAlgebra_generic` in `steenrod_algebra.py`.

`sage.algebras.steenrod.steenrod_algebra_bases.arnonC_basis(bound=1)`

Arnon's C basis in dimension n .

INPUT:

- n – nonnegative integer
- bound – positive integer (optional)

OUTPUT: tuple of basis elements in dimension n

The elements of Arnon's C basis are monomials of the form $\text{Sq}^{t_1} \dots \text{Sq}^{t_m}$ where for each i , we have $t_i \leq 2t_{i+1}$ and $2^i \mid t_{m-i}$.

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_bases import arnonC_basis
sage: arnonC_basis(7)
((7,), (2, 5), (4, 3), (4, 2, 1))
```

```
>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_bases import arnonC_basis
>>> arnonC_basis(Integer(7))
((7,), (2, 5), (4, 3), (4, 2, 1))
```

If optional argument `bound` is present, include only those monomials whose first term is at least as large as `bound`:

```
sage: arnonC_basis(7, 3)
((7,), (4, 3), (4, 2, 1))
```

```
>>> from sage.all import *
>>> arnonC_basis(Integer(7), Integer(3))
((7,), (4, 3), (4, 2, 1))
```

`sage.algebras.steenrod.steenrod_algebra_bases.atomic_basis(n, basis, **kws)`

Basis for dimension n made of elements in ‘atomic’ degrees: degrees of the form $2^i(2^j - 1)$.

This works at the prime 2 only.

INPUT:

- `n` – nonnegative integer
- `basis` – string, the name of the basis
- `profile` – profile function (default: `None`). Together with `truncation_type`, specify the profile function to be used; `None` means the profile function for the entire Steenrod algebra. See [sage.algebras.steenrod.steenrod_algebra](#) and [SteenrodAlgebra\(\)](#) for information on profile functions.
- `truncation_type` – truncation type, either 0 or `Infinity` (default: `Infinity` if no profile function is specified, 0 otherwise).

OUTPUT: tuple of basis elements in dimension n

The atomic bases include Wood’s Y and Z bases, Wall’s basis, Arnon’s A basis, the P_t^s -bases, and the commutator bases. (All of these bases are constructed similarly, hence their constructions have been consolidated into a single function. Also, see the documentation for [steenrod_algebra_basis\(\)](#) for descriptions of them.) For P_t^s -bases, you may also specify a profile function and truncation type; profile functions are ignored for the other bases.

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_bases import atomic_basis
sage: atomic_basis(6, 'woody')
((1, 0), (0, 1), (0, 0)), ((2, 0), (1, 0)), ((1, 1),))
sage: atomic_basis(8, 'woodz')
((2, 0), (0, 1), (0, 0)), ((0, 2), (0, 0)), ((1, 1), (1, 0)), ((3, 0),))
sage: atomic_basis(6, 'woodz') == atomic_basis(6, 'woody')
True
sage: atomic_basis(9, 'woodz') == atomic_basis(9, 'woody')
False
```

```

>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_bases import atomic_basis
>>> atomic_basis(Integer(6), 'woody')
((1, 0), (0, 1), (0, 0)), ((2, 0), (1, 0)), ((1, 1),)
>>> atomic_basis(Integer(8), 'woodz')
((2, 0), (0, 1), (0, 0)), ((0, 2), (0, 0)), ((1, 1), (1, 0)), ((3, 0),)
>>> atomic_basis(Integer(6), 'woodz') == atomic_basis(Integer(6), 'woody')
True
>>> atomic_basis(Integer(9), 'woodz') == atomic_basis(Integer(9), 'woody')
False

```

Wall's basis:

```

sage: atomic_basis(8, 'wall')
((2, 2), (1, 0), (0, 0)), ((2, 0), (0, 0)), ((2, 1), (1, 1)), ((3, 3),)

```

```

>>> from sage.all import *
>>> atomic_basis(Integer(8), 'wall')
((2, 2), (1, 0), (0, 0)), ((2, 0), (0, 0)), ((2, 1), (1, 1)), ((3, 3),)

```

Arnon's A basis:

```

sage: atomic_basis(7, 'arnona')
((0, 0), (1, 1), (2, 2)), ((0, 0), (2, 1)), ((1, 0), (2, 2)), ((2, 0),)

```

```

>>> from sage.all import *
>>> atomic_basis(Integer(7), 'arnona')
((0, 0), (1, 1), (2, 2)), ((0, 0), (2, 1)), ((1, 0), (2, 2)), ((2, 0),)

```

P_t^s -bases:

```

sage: atomic_basis(7, 'pst_rlex')
((0, 1), (1, 1), (2, 1)), ((0, 1), (1, 2)), ((2, 1), (0, 2)), ((0, 3),)
sage: atomic_basis(7, 'pst_llex')
((0, 1), (1, 1), (2, 1)), ((0, 1), (1, 2)), ((0, 2), (2, 1)), ((0, 3),)
sage: atomic_basis(7, 'pst_deg')
((0, 1), (1, 1), (2, 1)), ((0, 1), (1, 2)), ((0, 2), (2, 1)), ((0, 3),)
sage: atomic_basis(7, 'pst_revz')
((0, 1), (1, 1), (2, 1)), ((0, 1), (1, 2)), ((0, 2), (2, 1)), ((0, 3),)

```

```

>>> from sage.all import *
>>> atomic_basis(Integer(7), 'pst_rlex')
((0, 1), (1, 1), (2, 1)), ((0, 1), (1, 2)), ((2, 1), (0, 2)), ((0, 3),)
>>> atomic_basis(Integer(7), 'pst_llex')
((0, 1), (1, 1), (2, 1)), ((0, 1), (1, 2)), ((0, 2), (2, 1)), ((0, 3),)
>>> atomic_basis(Integer(7), 'pst_deg')
((0, 1), (1, 1), (2, 1)), ((0, 1), (1, 2)), ((0, 2), (2, 1)), ((0, 3),)
>>> atomic_basis(Integer(7), 'pst_revz')
((0, 1), (1, 1), (2, 1)), ((0, 1), (1, 2)), ((0, 2), (2, 1)), ((0, 3),)

```

Commutator bases:


```

sage: atomic_basis(7, 'comm_rlex')
((0, 1), (1, 1), (2, 1)), ((0, 1), (1, 2)), ((2, 1), (0, 2)), ((0, 3),)
sage: atomic_basis(7, 'comm_llex')
((0, 1), (1, 1), (2, 1)), ((0, 1), (1, 2)), ((0, 2), (2, 1)), ((0, 3),)
sage: atomic_basis(7, 'comm_deg')
((0, 1), (1, 1), (2, 1)), ((0, 1), (1, 2)), ((0, 2), (2, 1)), ((0, 3),)
sage: atomic_basis(7, 'comm_revz')
((0, 1), (1, 1), (2, 1)), ((0, 1), (1, 2)), ((0, 2), (2, 1)), ((0, 3),)

```

```

>>> from sage.all import *
>>> atomic_basis(Integer(7), 'comm_rlex')
((0, 1), (1, 1), (2, 1)), ((0, 1), (1, 2)), ((2, 1), (0, 2)), ((0, 3),)
>>> atomic_basis(Integer(7), 'comm_llex')
((0, 1), (1, 1), (2, 1)), ((0, 1), (1, 2)), ((0, 2), (2, 1)), ((0, 3),)
>>> atomic_basis(Integer(7), 'comm_deg')
((0, 1), (1, 1), (2, 1)), ((0, 1), (1, 2)), ((0, 2), (2, 1)), ((0, 3),)
>>> atomic_basis(Integer(7), 'comm_revz')
((0, 1), (1, 1), (2, 1)), ((0, 1), (1, 2)), ((0, 2), (2, 1)), ((0, 3),)

```

`sage.algebras.steenrod.steenrod_algebra_bases.atomic_basis_odd(n, basis, p, **kws)`

P_t^s -bases and commutator basis in dimension n at odd primes.

This function is called `atomic_basis_odd` in analogy with `atomic_basis()`.

INPUT:

- *n* – nonnegative integer
- *basis* – string, the name of the basis
- *p* – positive prime number
- *profile* – profile function (default: None). Together with *truncation_type*, specify the profile function to be used; None means the profile function for the entire Steenrod algebra. See `sage.algebras.steenrod.steenrod_algebra` and `SteenrodAlgebra()` for information on profile functions.
- *truncation_type* – truncation type, either 0 or Infinity (default: Infinity if no profile function is specified, 0 otherwise).

OUTPUT: tuple of basis elements in dimension n

The only possible difference in the implementations for P_t^s bases and commutator bases is that the former make sense, and require filtering, if there is a nontrivial profile function. This function is called by `steenrod_algebra_basis()`, and it will not be called for commutator bases if there is a profile function, so we treat the two bases exactly the same.

EXAMPLES:

```

sage: from sage.algebras.steenrod.steenrod_algebra_bases import atomic_basis_odd
sage: atomic_basis_odd(8, 'pst_rlex', 3)
(((), ((0, 1), 2)),)

sage: atomic_basis_odd(18, 'pst_rlex', 3)
(((0, 2), ()), ((0, 1), ((1, 1), 1)))
sage: atomic_basis_odd(18, 'pst_rlex', 3, profile=(((), (2, 2, 2)))
(((0, 2), ()),)

```

```

>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_bases import atomic_basis_odd
>>> atomic_basis_odd(Integer(8), 'pst_rlex', Integer(3))
(((), ((0, 1), 2)),)

>>> atomic_basis_odd(Integer(18), 'pst_rlex', Integer(3))
(((0, 2), ()), ((0, 1), ((1, 1), 1)))
>>> atomic_basis_odd(Integer(18), 'pst_rlex', Integer(3), profile=(),
↳ (Integer(2), Integer(2), Integer(2)))
(((0, 2), ()),)

```

`sage.algebras.steenrod.steenrod_algebra_bases.convert_from_milnor_matrix(n, basis, p=2, generic='auto')`

Change-of-basis matrix, Milnor to basis, in dimension n .

INPUT:

- n – nonnegative integer, the dimension
- *basis* – string, the basis to which to convert
- p – positive prime number (default: 2)

OUTPUT:

matrix – change-of-basis matrix, a square matrix over $\mathbf{F}_p$

Note

This is called internally. It is not intended for casual users, so no error checking is made on the integer n , the basis name, or the prime.

EXAMPLES:

```

sage: # needs sage.modules
sage: from sage.algebras.steenrod.steenrod_algebra_bases import convert_from_
↳ milnor_matrix, convert_to_milnor_matrix
sage: convert_from_milnor_matrix(12, 'wall')
[1 0 0 1 0 0 0]
[0 0 1 1 0 0 0]
[0 0 0 1 0 1 1]
[0 0 0 1 0 0 0]
[1 0 1 0 1 0 0]
[1 1 1 0 0 0 0]
[1 0 1 0 1 0 1]
sage: convert_from_milnor_matrix(38, 'serre_cartan')
72 x 72 dense matrix over Finite Field of size 2 (use the '.str()' method to see
↳ the entries)
sage: x = convert_to_milnor_matrix(20, 'wood_y')
sage: y = convert_from_milnor_matrix(20, 'wood_y')
sage: x*y
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]

```

(continues on next page)

(continued from previous page)

```

[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]

```

```

>>> from sage.all import *
>>> # needs sage.modules
>>> from sage.algebras.steenrod.steenrod_algebra_bases import convert_from_milnor_
↳matrix, convert_to_milnor_matrix
>>> convert_from_milnor_matrix(Integer(12), 'wall')
[1 0 0 1 0 0 0]
[0 0 1 1 0 0 0]
[0 0 0 1 0 1 1]
[0 0 0 1 0 0 0]
[1 0 1 0 1 0 0]
[1 1 1 0 0 0 0]
[1 0 1 0 1 0 1]
>>> convert_from_milnor_matrix(Integer(38), 'serre_cartan')
72 x 72 dense matrix over Finite Field of size 2 (use the '.str()' method to see
↳the entries)
>>> x = convert_to_milnor_matrix(Integer(20), 'wood_y')
>>> y = convert_from_milnor_matrix(Integer(20), 'wood_y')
>>> x*y
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]

```

The function takes an optional argument, the prime p over which to work:

```
sage: convert_from_milnor_matrix(17, 'adem', 3)
↪needs sage.modules
[2 1 1 2]
[0 2 0 1]
[1 2 0 0]
[0 1 0 0]
```

```
>>> from sage.all import *
>>> convert_from_milnor_matrix(Integer(17), 'adem', Integer(3))
↪ # needs sage.modules
[2 1 1 2]
[0 2 0 1]
[1 2 0 0]
[0 1 0 0]
```

```
sage.algebras.steenrod.steenrod_algebra_bases.convert_to_milnor_matrix(basis, p=2,
                                                                    generic='auto')
```

Change-of-basis matrix, ‘basis’ to Milnor, in dimension n , at the prime p .

INPUT:

- n – nonnegative integer, the dimension
- *basis* – string, the basis from which to convert
- p – positive prime number (default: 2)

OUTPUT:

matrix – change-of-basis matrix, a square matrix over $\mathbf{F}_p$

EXAMPLES:

```
sage: # needs sage.modules
sage: from sage.algebras.steenrod.steenrod_algebra_bases import convert_to_milnor_
↪matrix
sage: convert_to_milnor_matrix(5, 'adem') # indirect doctest
[0 1]
[1 1]
sage: convert_to_milnor_matrix(45, 'milnor')
111 x 111 dense matrix over Finite Field of size 2 (use the '.str()' method to
↪see the entries)
sage: convert_to_milnor_matrix(12, 'wall')
[1 0 0 1 0 0 0]
[1 1 0 0 0 1 0]
[0 1 0 1 0 0 0]
[0 0 0 1 0 0 0]
[1 1 0 0 1 0 0]
[0 0 1 1 1 0 1]
[0 0 0 0 1 0 1]
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> from sage.algebras.steenrod.steenrod_algebra_bases import convert_to_milnor_
↪matrix
```

(continues on next page)

(continued from previous page)

```

>>> convert_to_milnor_matrix(Integer(5), 'adem') # indirect doctest
[0 1]
[1 1]
>>> convert_to_milnor_matrix(Integer(45), 'milnor')
111 x 111 dense matrix over Finite Field of size 2 (use the '.str()' method to
→see the entries)
>>> convert_to_milnor_matrix(Integer(12), 'wall')
[1 0 0 1 0 0 0]
[1 1 0 0 0 1 0]
[0 1 0 1 0 0 0]
[0 0 0 1 0 0 0]
[1 1 0 0 1 0 0]
[0 0 1 1 1 0 1]
[0 0 0 0 1 0 1]

```

The function takes an optional argument, the prime p over which to work:

```

sage: # needs sage.modules
sage: convert_to_milnor_matrix(17, 'adem', 3)
[0 0 1 1]
[0 0 0 1]
[1 1 1 1]
[0 1 0 1]
sage: convert_to_milnor_matrix(48, 'adem', 5)
[0 1]
[1 1]
sage: convert_to_milnor_matrix(36, 'adem', 3)
[0 0 1]
[0 1 0]
[1 2 0]

```

```

>>> from sage.all import *
>>> # needs sage.modules
>>> convert_to_milnor_matrix(Integer(17), 'adem', Integer(3))
[0 0 1 1]
[0 0 0 1]
[1 1 1 1]
[0 1 0 1]
>>> convert_to_milnor_matrix(Integer(48), 'adem', Integer(5))
[0 1]
[1 1]
>>> convert_to_milnor_matrix(Integer(36), 'adem', Integer(3))
[0 0 1]
[0 1 0]
[1 2 0]

```

`sage.algebras.steenrod.steenrod_algebra_bases.milnor_basis( $n, p=2, **kws$ )`

Milnor basis in dimension n with profile function `profile`.

INPUT:

- n – nonnegative integer
- p – positive prime number (default: 2)

- `profile` – profile function (default: `None`). Together with `truncation_type`, specify the profile function to be used; `None` means the profile function for the entire Steenrod algebra. See `sage.algebras.steenrod.steenrod_algebra` and `SteenrodAlgebra` for information on profile functions.
- `truncation_type` – truncation type, either 0 or `Infinity` (default: `Infinity` if no profile function is specified, 0 otherwise)

OUTPUT: tuple of mod p Milnor basis elements in dimension n

At the prime 2, the Milnor basis consists of symbols of the form $Sq(m_1, m_2, \dots, m_t)$, where each m_i is a nonnegative integer and if $t > 1$, then $m_t \neq 0$. At odd primes, it consists of symbols of the form $Q_{e_1} Q_{e_2} \dots P(m_1, m_2, \dots, m_t)$, where $0 \leq e_1 < e_2 < \dots$, each m_i is a nonnegative integer, and if $t > 1$, then $m_t \neq 0$.

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_bases import milnor_basis
sage: milnor_basis(7)
((0, 0, 1), (1, 2), (4, 1), (7,))
sage: milnor_basis(7, 2)
((0, 0, 1), (1, 2), (4, 1), (7,))
sage: milnor_basis(4, 2)
((1, 1), (4,))
sage: milnor_basis(4, 2, profile=[2,1])
((1, 1),)
sage: milnor_basis(4, 2, profile=(), truncation_type=0)
()
sage: milnor_basis(4, 2, profile=(), truncation_type=Infinity)
((1, 1), (4,))
sage: milnor_basis(9, 3)
(((1,), (1,)), ((0,), (2,)))
sage: milnor_basis(17, 3)
(((2,), ()), ((1,), (3,)), ((0,), (0, 1)), ((0,), (4,)))
sage: milnor_basis(48, p=5)
((( ), (0, 1)), (( ), (6,)))
sage: len(milnor_basis(100,3))
13
sage: len(milnor_basis(200,7))
0
sage: len(milnor_basis(240,7))
3
sage: len(milnor_basis(240,7, profile=(), truncation_type=Infinity))
3
sage: len(milnor_basis(240,7, profile=(), truncation_type=0))
0
```

```
>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_bases import milnor_basis
>>> milnor_basis(Integer(7))
((0, 0, 1), (1, 2), (4, 1), (7,))
>>> milnor_basis(Integer(7), Integer(2))
((0, 0, 1), (1, 2), (4, 1), (7,))
>>> milnor_basis(Integer(4), Integer(2))
((1, 1), (4,))
>>> milnor_basis(Integer(4), Integer(2), profile=[Integer(2), Integer(1)])
```

(continues on next page)

(continued from previous page)

```

((1, 1),)
>>> milnor_basis(Integer(4), Integer(2), profile=(), truncation_type=Integer(0))
()
>>> milnor_basis(Integer(4), Integer(2), profile=(), truncation_type=Infinity)
((1, 1), (4,))
>>> milnor_basis(Integer(9), Integer(3))
(((1,), (1,)), ((0,), (2,)))
>>> milnor_basis(Integer(17), Integer(3))
(((2,), ()), ((1,), (3,)), ((0,), (0, 1)), ((0,), (4,)))
>>> milnor_basis(Integer(48), p=Integer(5))
(((0,), (0, 1)), ((0,), (6,)))
>>> len(milnor_basis(Integer(100), Integer(3)))
13
>>> len(milnor_basis(Integer(200), Integer(7)))
0
>>> len(milnor_basis(Integer(240), Integer(7)))
3
>>> len(milnor_basis(Integer(240), Integer(7), profile=(), truncation_
↪type=Infinity))
3
>>> len(milnor_basis(Integer(240), Integer(7), profile=(), truncation_
↪type=Integer(0)))
0

```

sage.algebras.steenrod.steenrod_algebra_bases.**restricted_partitions**(*n*, *l*, *no_repeats=False*)

Iterator over ‘restricted’ partitions of *n*: partitions with parts taken from list *l*.

INPUT:

- *n* – nonnegative integer
- *l* – list of positive integers
- *no_repeats* – boolean (default: False); if True, only return partitions with no repeated parts

OUTPUT: iterator of lists

One could also use `Partitions(n, parts_in=l)`, but this function may be faster. Also, while `Partitions(n, parts_in=l, max_slope=-1)` should in theory return the partitions of *n* with parts in *l* with no repetitions, the `max_slope=-1` argument is ignored, so it does not work. (At the moment, the `no_repeats=True` case is the only one used in the code.)

Todo

This should be re-implemented in a non-recursive way.

EXAMPLES:

```

sage: from sage.algebras.steenrod.steenrod_algebra_bases import restricted_
↪partitions
sage: list(restricted_partitions(10, [7,5,1]))
[[7, 1, 1, 1], [5, 5], [5, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1]]
sage: list(restricted_partitions(10, [6,5,4,3,2,1], no_repeats=True))
[[6, 4], [6, 3, 1], [5, 4, 1], [5, 3, 2], [4, 3, 2, 1]]

```

(continues on next page)

(continued from previous page)

```
sage: list(restricted_partitions(10, [6,4,2]))
[[6, 4], [6, 2, 2], [4, 4, 2], [4, 2, 2, 2], [2, 2, 2, 2, 2]]
sage: list(restricted_partitions(10, [6,4,2], no_repeats=True))
[[6, 4]]
```

```
>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_bases import restricted_
↳ partitions
>>> list(restricted_partitions(Integer(10), [Integer(7), Integer(5), Integer(1)]))
[[7, 1, 1, 1], [5, 5], [5, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]
>>> list(restricted_partitions(Integer(10), [Integer(6), Integer(5), Integer(4),
↳ Integer(3), Integer(2), Integer(1)], no_repeats=True))
[[6, 4], [6, 3, 1], [5, 4, 1], [5, 3, 2], [4, 3, 2, 1]]
>>> list(restricted_partitions(Integer(10), [Integer(6), Integer(4), Integer(2)]))
[[6, 4], [6, 2, 2], [4, 4, 2], [4, 2, 2, 2], [2, 2, 2, 2, 2]]
>>> list(restricted_partitions(Integer(10), [Integer(6), Integer(4), Integer(2)],
↳ no_repeats=True))
[[6, 4]]
```

1 may have repeated elements. If `no_repeats` is `False`, this has no effect. If `no_repeats` is `True`, and if the repeated elements appear consecutively in 1, then each element may be used only as many times as it appears in 1:

```
sage: list(restricted_partitions(10, [6,4,2,2], no_repeats=True))
[[6, 4], [6, 2, 2]]
sage: list(restricted_partitions(10, [6,4,2,2,2], no_repeats=True))
[[6, 4], [6, 2, 2], [4, 2, 2, 2]]
```

```
>>> from sage.all import *
>>> list(restricted_partitions(Integer(10), [Integer(6), Integer(4), Integer(2),
↳ Integer(2)], no_repeats=True))
[[6, 4], [6, 2, 2]]
>>> list(restricted_partitions(Integer(10), [Integer(6), Integer(4), Integer(2),
↳ Integer(2), Integer(2)], no_repeats=True))
[[6, 4], [6, 2, 2], [4, 2, 2, 2]]
```

(If the repeated elements don't appear consecutively, the results are likely meaningless, containing several partitions more than once, for example.)

In the following examples, `no_repeats` is `False`:

```
sage: list(restricted_partitions(10, [6,4,2]))
[[6, 4], [6, 2, 2], [4, 4, 2], [4, 2, 2, 2], [2, 2, 2, 2, 2]]
sage: list(restricted_partitions(10, [6,4,2,2,2]))
[[6, 4], [6, 2, 2], [4, 4, 2], [4, 2, 2, 2], [2, 2, 2, 2, 2]]
sage: list(restricted_partitions(10, [6,4,4,4,2,2,2,2,2]))
[[6, 4], [6, 2, 2], [4, 4, 2], [4, 2, 2, 2], [2, 2, 2, 2, 2]]
```

```
>>> from sage.all import *
>>> list(restricted_partitions(Integer(10), [Integer(6), Integer(4), Integer(2)]))
[[6, 4], [6, 2, 2], [4, 4, 2], [4, 2, 2, 2], [2, 2, 2, 2, 2]]
>>> list(restricted_partitions(Integer(10), [Integer(6), Integer(4), Integer(2),
↳ Integer(2), Integer(2)]))
```

(continues on next page)

(continued from previous page)

```
[[6, 4], [6, 2, 2], [4, 4, 2], [4, 2, 2, 2], [2, 2, 2, 2, 2]]
>>> list(restricted_partitions(Integer(10), [Integer(6), Integer(4), Integer(4),
↳ Integer(4), Integer(2), Integer(2), Integer(2), Integer(2), Integer(2)]))
[[6, 4], [6, 2, 2], [4, 4, 2], [4, 2, 2, 2], [2, 2, 2, 2, 2]]
```

sage.algebras.steenrod.steenrod_algebra_bases.**serre_cartan_basis**(*n*, *p*=2, *bound*=1, ***kws*)

Serre-Cartan basis in dimension *n*.

INPUT:

- *n* – nonnegative integer
- *bound* – positive integer (optional)
- *prime* – positive prime number (default: 2)

OUTPUT: tuple of mod *p* Serre-Cartan basis elements in dimension *n*

The Serre-Cartan basis consists of ‘admissible monomials in the Steenrod squares’. Thus at the prime 2, it consists of monomials $Sq^{m_1}Sq^{m_2}...Sq^{m_t}$ with $m_i \geq 2m_{i+1}$ for each *i*. At odd primes, it consists of monomials $\beta^{e_0}P^{s_1}\beta^{e_1}P^{s_2}...P^{s_k}\beta^{e_k}$ with each *e_i* either 0 or 1, $s_i \geq ps_{i+1} + e_i$ for all *i*, and $s_k \geq 1$.

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_bases import serre_cartan_basis
sage: serre_cartan_basis(7)
((7,), (6, 1), (4, 2, 1), (5, 2))
sage: serre_cartan_basis(13, 3)
((1, 3, 0), (0, 3, 1))
sage: serre_cartan_basis(50, 5)
((1, 5, 0, 1, 1), (1, 6, 1))
```

```
>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_bases import serre_cartan_basis
>>> serre_cartan_basis(Integer(7))
((7,), (6, 1), (4, 2, 1), (5, 2))
>>> serre_cartan_basis(Integer(13), Integer(3))
((1, 3, 0), (0, 3, 1))
>>> serre_cartan_basis(Integer(50), Integer(5))
((1, 5, 0, 1, 1), (1, 6, 1))
```

If optional argument *bound* is present, include only those monomials whose last term is at least *bound* (when *p* = 2), or those for which $s_k - e_k \geq bound$ (when *p* is odd).

```
sage: serre_cartan_basis(7, bound=2)
((7,), (5, 2))
sage: serre_cartan_basis(13, 3, bound=3)
((1, 3, 0),)
```

```
>>> from sage.all import *
>>> serre_cartan_basis(Integer(7), bound=Integer(2))
((7,), (5, 2))
>>> serre_cartan_basis(Integer(13), Integer(3), bound=Integer(3))
((1, 3, 0),)
```

`sage.algebras.steenrod.steenrod_algebra_bases.steenrod_algebra_basis` (*basis*='milnor', *p*=2, ***kws*)

Basis for the Steenrod algebra in degree n .

INPUT:

- *n* – nonnegative integer
- *basis* – string, which basis to use (default: 'milnor')
- *p* – positive prime number (default: 2)
- *profile* – profile function (default: None); this is just passed on to the functions `milnor_basis()` and `pst_basis()`
- *truncation_type* – truncation type, either 0 or Infinity (default: Infinity if no profile function is specified, 0 otherwise). This is just passed on to the function `milnor_basis()`.
- *generic* – boolean (default: None)

OUTPUT:

Tuple of objects representing basis elements for the Steenrod algebra in dimension n .

The choices for the string *basis* are as follows; see the documentation for `sage.algebras.steenrod.steenrod_algebra` for details on each basis:

- 'milnor' – Milnor basis
- 'serre-cartan' or 'adem' or 'admissible' – Serre-Cartan basis
- 'pst', 'pst_rlex', 'pst_lllex', 'pst_deg', 'pst_revz' – various P_t^s -bases
- 'comm', 'comm_rlex', 'comm_lllex', 'comm_deg', 'comm_revz', or any of these with '_long' appended – various commutator bases

The rest of these bases are only defined when $p = 2$.

- 'wood_y' – Wood's Y basis
- 'wood_z' – Wood's Z basis
- 'wall' or 'wall_long' – Wall's basis
- 'arnon_a' or 'arnon_a_long' – Arnon's A basis
- 'arnon_c' – Arnon's C basis

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_bases import steenrod_algebra_
      ↪basis
sage: steenrod_algebra_basis(7, 'milnor') # indirect doctest
((0, 0, 1), (1, 2), (4, 1), (7,))
sage: steenrod_algebra_basis(5) # milnor basis is the default
((2, 1), (5,))
```

```
>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_bases import steenrod_algebra_
      ↪basis
>>> steenrod_algebra_basis(Integer(7), 'milnor') # indirect doctest
((0, 0, 1), (1, 2), (4, 1), (7,))
>>> steenrod_algebra_basis(Integer(5)) # milnor basis is the default
((2, 1), (5,))
```

Bases in negative dimensions are empty:

```
sage: steenrod_algebra_basis(-2, 'wall')
()
```

```
>>> from sage.all import *
>>> steenrod_algebra_basis(-Integer(2), 'wall')
()
```

The third (optional) argument to 'steenrod_algebra_basis' is the prime p:

```
sage: steenrod_algebra_basis(9, 'milnor', p=3)
(((1,), (1,)), ((0,), (2,)))
sage: steenrod_algebra_basis(9, 'milnor', 3)
(((1,), (1,)), ((0,), (2,)))
sage: steenrod_algebra_basis(17, 'milnor', 3)
(((2,), ()), ((1,), (3,)), ((0,), (0, 1)), ((0,), (4,)))
```

```
>>> from sage.all import *
>>> steenrod_algebra_basis(Integer(9), 'milnor', p=Integer(3))
(((1,), (1,)), ((0,), (2,)))
>>> steenrod_algebra_basis(Integer(9), 'milnor', Integer(3))
(((1,), (1,)), ((0,), (2,)))
>>> steenrod_algebra_basis(Integer(17), 'milnor', Integer(3))
(((2,), ()), ((1,), (3,)), ((0,), (0, 1)), ((0,), (4,)))
```

Other bases:

```
sage: steenrod_algebra_basis(7, 'admissible')
((7,), (6, 1), (4, 2, 1), (5, 2))
sage: steenrod_algebra_basis(13, 'admissible', p=3)
((1, 3, 0), (0, 3, 1))
sage: steenrod_algebra_basis(5, 'wall')
(((2, 2), (0, 0)), ((1, 1), (1, 0)))
sage: steenrod_algebra_basis(5, 'wall_long')
(((2, 2), (0, 0)), ((1, 1), (1, 0)))
sage: steenrod_algebra_basis(5, 'pst-rlex')
(((0, 1), (2, 1)), ((1, 1), (0, 2)))
```

```
>>> from sage.all import *
>>> steenrod_algebra_basis(Integer(7), 'admissible')
((7,), (6, 1), (4, 2, 1), (5, 2))
>>> steenrod_algebra_basis(Integer(13), 'admissible', p=Integer(3))
((1, 3, 0), (0, 3, 1))
>>> steenrod_algebra_basis(Integer(5), 'wall')
(((2, 2), (0, 0)), ((1, 1), (1, 0)))
>>> steenrod_algebra_basis(Integer(5), 'wall_long')
(((2, 2), (0, 0)), ((1, 1), (1, 0)))
>>> steenrod_algebra_basis(Integer(5), 'pst-rlex')
(((0, 1), (2, 1)), ((1, 1), (0, 2)))
```

`sage.algebras.steenrod.steenrod_algebra_bases.steenrod_basis_error_check(dim, p, **kws)`

This performs crude error checking.

INPUT:

- dim – nonnegative integer
- p – positive prime number

OUTPUT: none

This checks to see if the different bases have the same length, and if the change-of-basis matrices are invertible. If something goes wrong, an error message is printed.

This function checks at the prime p as the dimension goes up from 0 to dim .

If you set the Sage verbosity level to a positive integer (using `set_verbosity(n)`), then some extra messages will be printed.

EXAMPLES:

```
sage: # long time
sage: from sage.algebras.steenrod.steenrod_algebra_bases import steenrod_basis_
      ↪error_check
sage: steenrod_basis_error_check(15, 2)
sage: steenrod_basis_error_check(15, 2, generic=True)
sage: steenrod_basis_error_check(40, 3)
sage: steenrod_basis_error_check(80, 5)
```

```
>>> from sage.all import *
>>> # long time
>>> from sage.algebras.steenrod.steenrod_algebra_bases import steenrod_basis_
      ↪error_check
>>> steenrod_basis_error_check(Integer(15), Integer(2))
>>> steenrod_basis_error_check(Integer(15), Integer(2), generic=True)
>>> steenrod_basis_error_check(Integer(40), Integer(3))
>>> steenrod_basis_error_check(Integer(80), Integer(5))
```

`sage.algebras.steenrod.steenrod_algebra_bases.xi_degrees(n, p=2, reverse=True)`

Decreasing list of degrees of the ξ_i 's, starting in degree n .

INPUT:

- n – integer
- p – prime number (default: 2)
- `reverse` – boolean (default: True)

OUTPUT: list of integers

When $p = 2$: decreasing list of the degrees of the ξ_i 's with degree at most n .

At odd primes: decreasing list of these degrees, each divided by $2(p - 1)$.

If `reverse` is False, then return an increasing list rather than a decreasing one.

EXAMPLES:

```
sage: sage.algebras.steenrod.steenrod_algebra_bases.xi_degrees(17)
[15, 7, 3, 1]
sage: sage.algebras.steenrod.steenrod_algebra_bases.xi_degrees(17, reverse=False)
[1, 3, 7, 15]
sage: sage.algebras.steenrod.steenrod_algebra_bases.xi_degrees(17, p=3)
```

(continues on next page)

(continued from previous page)

```
[13, 4, 1]
sage: sage.algebras.steenrod.steenrod_algebra_bases.xi_degrees(400, p=17)
[307, 18, 1]

>>> from sage.all import *
>>> sage.algebras.steenrod.steenrod_algebra_bases.xi_degrees(Integer(17))
[15, 7, 3, 1]
>>> sage.algebras.steenrod.steenrod_algebra_bases.xi_degrees(Integer(17), ↵
↵reverse=False)
[1, 3, 7, 15]
>>> sage.algebras.steenrod.steenrod_algebra_bases.xi_degrees(Integer(17), ↵
↵p=Integer(3))
[13, 4, 1]
>>> sage.algebras.steenrod.steenrod_algebra_bases.xi_degrees(Integer(400), ↵
↵p=Integer(17))
[307, 18, 1]
```

5.26 Miscellaneous functions for the Steenrod algebra and its elements

AUTHORS:

- John H. Palmieri (2008-07-30): initial version (as the file `steenrod_algebra_element.py`)
- John H. Palmieri (2010-06-30): initial version of `steenrod_misc.py`. Implemented profile functions. Moved most of the methods for elements to the `Element` subclass of `sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic`.

The main functions here are

- `get_basis_name()`. This function takes a string as input and attempts to interpret it as the name of a basis for the Steenrod algebra; it returns the canonical name attached to that basis. This allows for the use of synonyms when defining bases, while the resulting algebras will be identical.
- `normalize_profile()`. This function returns the canonical (and hashable) description of any profile function. See `sage.algebras.steenrod.steenrod_algebra` and `SteenrodAlgebra` for information on profile functions.
- functions named `*_mono_to_string` where `*` is a basis name (`milnor_mono_to_string()`, etc.). These convert tuples representing basis elements to strings, for `_repr_` and `_latex_` methods.

```
sage.algebras.steenrod.steenrod_algebra_misc.arnonA_long_mono_to_string(mono, latex=False,
                                                                    p=2)
```

Alternate string representation of element of Arnon's A basis.

This is used by the `_repr_` and `_latex_` methods.

INPUT:

- `mono` – tuple of pairs of nonnegative integers (m, k) with $m \geq k$
- `latex` – boolean (default: `False`); if `True`, output LaTeX string

OUTPUT:

`string` – concatenation of strings of the form $Sq(2^m)$

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_misc import arnonA_long_mono_
      ↪to_string
sage: arnonA_long_mono_to_string(((1,2),(3,0)))
'Sq^{8} Sq^{4} Sq^{2} Sq^{1}'
sage: arnonA_long_mono_to_string(((1,2),(3,0)), latex=True)
'\\text{Sq}^{8} \\text{Sq}^{4} \\text{Sq}^{2} \\text{Sq}^{1}'
```

```
>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_misc import arnonA_long_mono_to_
      ↪string
>>> arnonA_long_mono_to_string(((Integer(1),Integer(2)),(Integer(3),Integer(0))))
'Sq^{8} Sq^{4} Sq^{2} Sq^{1}'
>>> arnonA_long_mono_to_string(((Integer(1),Integer(2)),(Integer(3),Integer(0))),
      ↪latex=True)
'\\text{Sq}^{8} \\text{Sq}^{4} \\text{Sq}^{2} \\text{Sq}^{1}'
```

The empty tuple represents the unit element:

```
sage: arnonA_long_mono_to_string(())
'1'
```

```
>>> from sage.all import *
>>> arnonA_long_mono_to_string(())
'1'
```

`sage.algebras.steenrod.steenrod_algebra_misc.arnonA_mono_to_string(mono, latex=False, p=2)`

String representation of element of Arnon's A basis.

This is used by the `_repr_` and `_latex_` methods.

INPUT:

- **mono** – tuple of pairs of nonnegative integers
(m,k) with $m \geq k$
- **latex** – boolean (default: False); if True, output LaTeX string

OUTPUT: concatenation of strings of the form $X^{\{m\}}_{\{k\}}$ for each pair (m,k)

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_misc import arnonA_mono_to_
      ↪string
sage: arnonA_mono_to_string(((1,2),(3,0)))
'X^{1}_{2} X^{3}_{0}'
sage: arnonA_mono_to_string(((1,2),(3,0)), latex=True)
'X^{\{1\}}_{\{2\}} X^{\{3\}}_{\{0\}}'
```

```
>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_misc import arnonA_mono_to_string
>>> arnonA_mono_to_string(((Integer(1),Integer(2)),(Integer(3),Integer(0))))
'X^{1}_{2} X^{3}_{0}'
>>> arnonA_mono_to_string(((Integer(1),Integer(2)),(Integer(3),Integer(0))),
```

(continues on next page)

(continued from previous page)

```
↪ latex=True)
'X^{1}_{2} X^{3}_{0}'
```

The empty tuple represents the unit element:

```
sage: arnonA_mono_to_string(())
'1'
```

```
>>> from sage.all import *
>>> arnonA_mono_to_string(())
'1'
```

```
sage.algebras.steenrod.steenrod_algebra_misc.comm_long_mono_to_string(mono, p, latex=False,
                                                                    generic=False)
```

Alternate string representation of element of a commutator basis.

Okay in low dimensions, but gets unwieldy as the dimension increases.

INPUT:

- `mono` – tuple of pairs of integers (s,t) with $s \geq 0, t \geq 0$
- `latex` – boolean (default: `False`); if `True`, output LaTeX string
- `generic` – whether to format generically, or for the prime 2 (default)

OUTPUT:

string – concatenation of strings of the form $s_{2^s \dots 2^{s+t-1}}$ for each pair (s,t)

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_misc import comm_long_mono_to_
↪ string
sage: comm_long_mono_to_string(((1,2),(0,3)), 2)
's_{24} s_{124}'
sage: comm_long_mono_to_string(((1,2),(0,3)), 2, latex=True)
's_{24} s_{124}'
sage: comm_long_mono_to_string(((1, 4), ((1,2), 1), ((0,3), 2))), 5, generic=True)
'Q_{1} Q_{4} s_{5,25} s_{1,5,25}^2'
sage: comm_long_mono_to_string(((1, 4), ((1,2), 1), ((0,3), 2))), 3, latex=True, ↪
↪ generic=True)
'Q_{1} Q_{4} s_{3,9} s_{1,3,9}^2'
```

```
>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_misc import comm_long_mono_to_
↪ string
>>> comm_long_mono_to_string(((Integer(1),Integer(2)),(Integer(0),Integer(3))), ↪
↪ Integer(2))
's_{24} s_{124}'
>>> comm_long_mono_to_string(((Integer(1),Integer(2)),(Integer(0),Integer(3))), ↪
↪ Integer(2), latex=True)
's_{24} s_{124}'
>>> comm_long_mono_to_string(((Integer(1), Integer(4)), (((Integer(1),Integer(2)),
↪ Integer(1)),((Integer(0),Integer(3)), Integer(2)))), Integer(5), generic=True)
```

(continues on next page)

(continued from previous page)

```
'Q_{1} Q_{4} s_{5,25} s_{1,5,25}^2'
>>> comm_long_mono_to_string(((Integer(1), Integer(4)), (((Integer(1), Integer(2)),
↳ Integer(1)), ((Integer(0), Integer(3)), Integer(2)))), Integer(3), latex=True,
↳ generic=True)
'Q_{1} Q_{4} s_{3,9} s_{1,3,9}^2'
```

The empty tuple represents the unit element:

```
sage: comm_long_mono_to_string((), p=2)
'1'
```

```
>>> from sage.all import *
>>> comm_long_mono_to_string((), p=Integer(2))
'1'
```

```
sage.algebras.steenrod.steenrod_algebra_misc.comm_mono_to_string(mono, latex=False,
                                                                    generic=False)
```

String representation of element of a commutator basis.

This is used by the `_repr_` and `_latex_` methods.

INPUT:

- `mono` – tuple of pairs of integers (s,t) with $s \geq 0, t > 0$
- `latex` – boolean (default: False); if True, output LaTeX string
- `generic` – whether to format generically, or for the prime 2 (default)

OUTPUT:

`string` – concatenation of strings of the form `c_{s,t}` for each pair (s,t)

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_misc import comm_mono_to_string
sage: comm_mono_to_string(((1,2), (0,3)), generic=False)
'c_{1,2} c_{0,3}'
sage: comm_mono_to_string(((1,2), (0,3)), latex=True)
'c_{1,2} c_{0,3}'
sage: comm_mono_to_string(((1, 4), (((1,2), 1), ((0,3), 2))), generic=True)
'Q_{1} Q_{4} c_{1,2} c_{0,3}^2'
sage: comm_mono_to_string(((1, 4), (((1,2), 1), ((0,3), 2))), latex=True,
↳ generic=True)
'Q_{1} Q_{4} c_{1,2} c_{0,3}^2'
```

```
>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_misc import comm_mono_to_string
>>> comm_mono_to_string(((Integer(1), Integer(2)), (Integer(0), Integer(3))),
↳ generic=False)
'c_{1,2} c_{0,3}'
>>> comm_mono_to_string(((Integer(1), Integer(2)), (Integer(0), Integer(3))),
↳ latex=True)
'c_{1,2} c_{0,3}'
>>> comm_mono_to_string(((Integer(1), Integer(4)), (((Integer(1), Integer(2)),
↳
```

(continues on next page)

(continued from previous page)

```

↪Integer(1)), ((Integer(0), Integer(3)), Integer(2))), generic=True)
'Q_{1} Q_{4} c_{1,2} c_{0,3}^2'
>>> comm_mono_to_string(((Integer(1), Integer(4)), ((Integer(1), Integer(2)),
↪Integer(1)), ((Integer(0), Integer(3)), Integer(2))), latex=True, generic=True)
'Q_{1} Q_{4} c_{1,2} c_{0,3}^2'

```

The empty tuple represents the unit element:

```

sage: comm_mono_to_string(())
'1'

```

```

>>> from sage.all import *
>>> comm_mono_to_string(())
'1'

```

sage.algebras.steenrod.steenrod_algebra_misc.convert_perm(*m*)

Convert tuple *m* of nonnegative integers to a permutation in one-line form.

INPUT:

- *m* – tuple of nonnegative integers with no repetitions

OUTPUT:

list – conversion of *m* to a permutation of the set 1,2,...,len(*m*)

If *m* = (3, 7, 4), then one can view *m* as representing the permutation of the set (3, 4, 7) sending 3 to 3, 4 to 7, and 7 to 4. This function converts *m* to the list [1, 3, 2], which represents essentially the same permutation, but of the set (1, 2, 3). This list can then be passed to `Permutation`, and its signature can be computed.

EXAMPLES:

```

sage: sage.algebras.steenrod.steenrod_algebra_misc.convert_perm((3,7,4))
[1, 3, 2]
sage: sage.algebras.steenrod.steenrod_algebra_misc.convert_perm((5,0,6,3))
[3, 1, 4, 2]

```

```

>>> from sage.all import *
>>> sage.algebras.steenrod.steenrod_algebra_misc.convert_perm((Integer(3),
↪Integer(7), Integer(4)))
[1, 3, 2]
>>> sage.algebras.steenrod.steenrod_algebra_misc.convert_perm((Integer(5),
↪Integer(0), Integer(6), Integer(3)))
[3, 1, 4, 2]

```

sage.algebras.steenrod.steenrod_algebra_misc.get_basis_name(*basis*, *p*, *generic=None*)

Return canonical basis named by string *basis* at the prime *p*.

INPUT:

- *basis* – string
- *p* – positive prime number
- *generic* – boolean (default: 'None')

OUTPUT: *basis_name* – string

Specify the names of the implemented bases. The input is converted to lower-case, then processed to return the canonical name for the basis.

For the Milnor and Serre-Cartan bases, use the list of synonyms defined by the variables `_steenrod_milnor_basis_names` and `_steenrod_serre_cartan_basis_names`. Their canonical names are ‘milnor’ and ‘serre-cartan’, respectively.

For the other bases, use pattern-matching rather than a list of synonyms:

- Search for ‘wood’ and ‘y’ or ‘wood’ and ‘z’ to get the Wood bases. Canonical names ‘woody’, ‘woodz’.
- Search for ‘arnon’ and ‘c’ for the Arnon C basis. Canonical name: ‘arnonc’.
- Search for ‘arnon’ (and no ‘c’) for the Arnon A basis. Also see if ‘long’ is present, for the long form of the basis. Canonical names: ‘arnona’, ‘arnona_long’.
- Search for ‘wall’ for the Wall basis. Also see if ‘long’ is present. Canonical names: ‘wall’, ‘wall_long’.
- Search for ‘pst’ for P^s_t bases, then search for the order type: ‘rlex’, ‘llex’, ‘deg’, ‘revz’. Canonical names: ‘pst_rlex’, ‘pst_llex’, ‘pst_deg’, ‘pst_revz’.
- For commutator types, search for ‘comm’, an order type, and also check to see if ‘long’ is present. Canonical names: ‘comm_rlex’, ‘comm_llex’, ‘comm_deg’, ‘comm_revz’, ‘comm_rlex_long’, ‘comm_llex_long’, ‘comm_deg_long’, ‘comm_revz_long’.

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_misc import get_basis_name
sage: get_basis_name('adem', 2)
'serre-cartan'
sage: get_basis_name('milnor', 2)
'milnor'
sage: get_basis_name('MiLNoR', 5)
'milnor'
sage: get_basis_name('pst-llex', 2)
'pst_llex'
sage: get_basis_name('wood_abcdedfg_y', 2)
'woody'
sage: get_basis_name('wood', 2)
Traceback (most recent call last):
...
ValueError: wood is not a recognized basis at the prime 2
sage: get_basis_name('arnon--hello--long', 2)
'arnona_long'
sage: get_basis_name('arnona_long', p=5)
Traceback (most recent call last):
...
ValueError: arnona_long is not a recognized basis at the prime 5
sage: get_basis_name('NOT_A_BASIS', 2)
Traceback (most recent call last):
...
ValueError: not_a_basis is not a recognized basis at the prime 2
sage: get_basis_name('woody', 2, generic=True)
Traceback (most recent call last):
...
ValueError: woody is not a recognized basis for the generic Steenrod algebra at
↳ the prime 2
```

```

>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_misc import get_basis_name
>>> get_basis_name('adem', Integer(2))
'serre-cartan'
>>> get_basis_name('milnor', Integer(2))
'milnor'
>>> get_basis_name('MiLNoR', Integer(5))
'milnor'
>>> get_basis_name('pst-llex', Integer(2))
'pst_llex'
>>> get_basis_name('wood_abcdedfg_y', Integer(2))
'woody'
>>> get_basis_name('wood', Integer(2))
Traceback (most recent call last):
...
ValueError: wood is not a recognized basis at the prime 2
>>> get_basis_name('arnon--hello--long', Integer(2))
'arnona_long'
>>> get_basis_name('arnona_long', p=Integer(5))
Traceback (most recent call last):
...
ValueError: arnona_long is not a recognized basis at the prime 5
>>> get_basis_name('NOT_A_BASIS', Integer(2))
Traceback (most recent call last):
...
ValueError: not_a_basis is not a recognized basis at the prime 2
>>> get_basis_name('woody', Integer(2), generic=True)
Traceback (most recent call last):
...
ValueError: woody is not a recognized basis for the generic Steenrod algebra at
↳the prime 2

```

`sage.algebras.steenrod.steenrod_algebra_misc.is_valid_profile(profile, truncation_type, p=2, generic=None)`

Return True if profile, together with truncation_type, is a valid profile at the prime p .

INPUT:

- profile – when $p = 2$, a tuple or list of numbers; when p is odd, a pair of such lists
- truncation_type – either 0 or ∞
- p – prime number (default: 2)
- generic – boolean (default: None)

OUTPUT: True if the profile function is valid, False otherwise

See the documentation for `sage.algebras.steenrod.steenrod_algebra` for descriptions of profile functions and how they correspond to sub-Hopf algebras of the Steenrod algebra. Briefly: at the prime 2, a profile function e is valid if it satisfies the condition

- $e(r) \geq \min(e(r-i) - i, e(i))$ for all $0 < i < r$.

At odd primes, a pair of profile functions e and k are valid if they satisfy

- $e(r) \geq \min(e(r-i) - i, e(i))$ for all $0 < i < r$.
- if $k(i+j) = 1$, then either $e(i) \leq j$ or $k(j) = 1$ for all $i \geq 1, j \geq 0$.

In this function, profile functions are lists or tuples, and `truncation_type` is appended as the last element of the list e before testing.

EXAMPLES:

$p = 2$:

```
sage: from sage.algebras.steenrod.steenrod_algebra_misc import is_valid_profile
sage: is_valid_profile([3,2,1], 0)
True
sage: is_valid_profile([3,2,1], Infinity)
True
sage: is_valid_profile([1,2,3], 0)
False
sage: is_valid_profile([6,2,0], Infinity)
False
sage: is_valid_profile([0,3], 0)
False
sage: is_valid_profile([0,0,4], 0)
False
sage: is_valid_profile([0,0,0,4,0], 0)
True
```

```
>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_misc import is_valid_profile
>>> is_valid_profile([Integer(3),Integer(2),Integer(1)], Integer(0))
True
>>> is_valid_profile([Integer(3),Integer(2),Integer(1)], Infinity)
True
>>> is_valid_profile([Integer(1),Integer(2),Integer(3)], Integer(0))
False
>>> is_valid_profile([Integer(6),Integer(2),Integer(0)], Infinity)
False
>>> is_valid_profile([Integer(0),Integer(3)], Integer(0))
False
>>> is_valid_profile([Integer(0),Integer(0),Integer(4)], Integer(0))
False
>>> is_valid_profile([Integer(0),Integer(0),Integer(0),Integer(4),Integer(0)],
↳ Integer(0))
True
```

Odd primes:

```
sage: is_valid_profile([0,0,0], [2,1,1,1,2,2]), 0, p=3)
True
sage: is_valid_profile([1], [2,2]), 0, p=3)
True
sage: is_valid_profile([1], [2]), 0, p=7)
False
sage: is_valid_profile([1,2,1], [], 0, p=7)
True
sage: is_valid_profile([0,0,0], [2,1,1,1,2,2]), 0, p=2, generic=True)
True
```

```

>>> from sage.all import *
>>> is_valid_profile([Integer(0), Integer(0), Integer(0)], [Integer(2), Integer(1),
↳ Integer(1), Integer(1), Integer(2), Integer(2)], Integer(0), p=Integer(3))
True
>>> is_valid_profile([Integer(1)], [Integer(2), Integer(2)], Integer(0),
↳ p=Integer(3))
True
>>> is_valid_profile([Integer(1)], [Integer(2)], Integer(0), p=Integer(7))
False
>>> is_valid_profile([Integer(1), Integer(2), Integer(1)], [], Integer(0),
↳ p=Integer(7))
True
>>> is_valid_profile([Integer(0), Integer(0), Integer(0)], [Integer(2), Integer(1),
↳ Integer(1), Integer(1), Integer(2), Integer(2)], Integer(0), p=Integer(2),
↳ generic=True)
True

```

`sage.algebras.steenrod.steenrod_algebra_misc.milnor_mono_to_string(mono, latex=False, generic=False)`

String representation of element of the Milnor basis.

This is used by the `_repr_` and `_latex_` methods.

INPUT:

- `mono` – if *generic* = *False*, tuple of nonnegative integers (a,b,c,...); if *generic* = *True*, pair of tuples of nonnegative integers ((e0, e1, e2, ...), (r1, r2, ...))
- `latex` – boolean (default: *False*); if *True*, output LaTeX string
- `generic` – whether to format generically, or for the prime 2 (default)

OUTPUT: `rep` – string

This returns a string like `Sq(a,b,c,...)` when *generic* = *False*, or a string like `Q_e0 Q_e1 Q_e2 ... P(r1, r2, ...)` when *generic* = *True*.

EXAMPLES:

```

sage: from sage.algebras.steenrod.steenrod_algebra_misc import milnor_mono_to_
↳ string
sage: milnor_mono_to_string((1,2,3,4))
'Sq(1,2,3,4)'
sage: milnor_mono_to_string((1,2,3,4), latex=True)
'\\text{Sq}(1,2,3,4)'
sage: milnor_mono_to_string(((1,0), (2,3,1)), generic=True)
'Q_{1} Q_{0} P(2,3,1)'
sage: milnor_mono_to_string(((1,0), (2,3,1)), latex=True, generic=True)
'Q_{1} Q_{0} \\mathcal{P}(2,3,1)'

```

```

>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_misc import milnor_mono_to_string
>>> milnor_mono_to_string((Integer(1), Integer(2), Integer(3), Integer(4)))
'Sq(1,2,3,4)'
>>> milnor_mono_to_string((Integer(1), Integer(2), Integer(3), Integer(4)),
↳ latex=True)

```

(continues on next page)

(continued from previous page)

```
'\\text{Sq}(1,2,3,4) '
>>> milnor_mono_to_string(((Integer(1),Integer(0)), (Integer(2),Integer(3),
↪Integer(1))), generic=True)
'Q_{1} Q_{0} P(2,3,1) '
>>> milnor_mono_to_string(((Integer(1),Integer(0)), (Integer(2),Integer(3),
↪Integer(1))), latex=True, generic=True)
'Q_{1} Q_{0} \\mathcal{P}(2,3,1) '
```

The empty tuple represents the unit element:

```
sage: milnor_mono_to_string(())
'1'
sage: milnor_mono_to_string((), generic=True)
'1'
```

```
>>> from sage.all import *
>>> milnor_mono_to_string(())
'1'
>>> milnor_mono_to_string((), generic=True)
'1'
```

```
sage.algebras.steenrod.steenrod_algebra_misc.normalize_profile(profile, precision=None,
                                                                truncation_type='auto', p=2,
                                                                generic=None)
```

Given a profile function and related data, return it in a standard form, suitable for hashing and caching as data defining a sub-Hopf algebra of the Steenrod algebra.

INPUT:

- `profile` – a profile function in form specified below
- `precision` – integer or `None` (default: `None`)
- `truncation_type` – 0 or ∞ or 'auto' (default: 'auto')
- `p` – prime (default: 2)
- `generic` – boolean (default: `None`)

OUTPUT:

a triple `profile, precision, truncation_type`, in standard form as described below.

The “standard form” is as follows: `profile` should be a tuple of integers (or ∞) with no trailing zeroes when $p = 2$, or a pair of such when p is odd or `generic` is `True`. `precision` should be a positive integer. `truncation_type` should be 0 or ∞ . Furthermore, this must be a valid profile, as determined by the function `is_valid_profile()`. See also the documentation for the module `sage.algebras.steenrod.steenrod_algebra` for information about profile functions.

For the inputs: when $p = 2$, `profile` should be a valid profile function, and it may be entered in any of the following forms:

- a list or tuple, e.g., `[3, 2, 1, 1]`
- a function from positive integers to nonnegative integers (and ∞), e.g., `lambda n: n+2`. This corresponds to the list `[3, 4, 5, ...]`.
- `None` or `Infinity` – use this for the profile function for the whole Steenrod algebra. This corresponds to the list `[Infinity, Infinity, Infinity, ...]`

To make this hashable, it gets turned into a tuple. In the first case it is clear how to do this; also in this case, `precision` is set to be one more than the length of this tuple. In the second case, construct a tuple of length one less than `precision` (default: 100). In the last case, the empty tuple is returned and `precision` is set to 1.

Once a sub-Hopf algebra of the Steenrod algebra has been defined using such a profile function, if the code requires any remaining terms (say, terms after the 100th), then they are given by `truncation_type` if that is 0 or ∞ . If `truncation_type` is 'auto', then in the case of a tuple, it gets set to 0, while for the other cases it gets set to ∞ .

See the examples below.

When p is odd, `profile` is a pair of “functions”, so it may have the following forms:

- a pair of lists or tuples, the second of which takes values in the set $\{1, 2\}$, e.g., `([3, 2, 1, 1], [1, 1, 2, 2, 1])`.
- a pair of functions, one (called e) from positive integers to nonnegative integers (and ∞), one (called k) from nonnegative integers to the set $\{1, 2\}$, e.g., `(lambda n: n+2, lambda n: 1)`. This corresponds to the pair `([3, 4, 5, ...], [1, 1, 1, ...])`.
- `None` or `Infinity` – use this for the profile function for the whole Steenrod algebra. This corresponds to the pair `([Infinity, Infinity, Infinity, ...], [2, 2, 2, ...])`.

You can also mix and match the first two, passing a pair with first entry a list and second entry a function, for instance. The values of `precision` and `truncation_type` are determined by the first entry.

EXAMPLES:

$p = 2$:

```
sage: from sage.algebras.steenrod.steenrod_algebra_misc import normalize_profile
sage: normalize_profile([1, 2, 1, 0, 0])
((1, 2, 1), 0)
```

```
>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_misc import normalize_profile
>>> normalize_profile([Integer(1), Integer(2), Integer(1), Integer(0), Integer(0)])
((1, 2, 1), 0)
```

The full mod 2 Steenrod algebra:

```
sage: normalize_profile(Infinity)
((), +Infinity)
sage: normalize_profile(None)
((), +Infinity)
sage: normalize_profile(lambda n: Infinity)
((), +Infinity)
```

```
>>> from sage.all import *
>>> normalize_profile(Infinity)
((), +Infinity)
>>> normalize_profile(None)
((), +Infinity)
>>> normalize_profile(lambda n: Infinity)
((), +Infinity)
```

The `precision` argument has no effect when the first argument is a list or tuple:

```
sage: normalize_profile([1,2,1,0,0], precision=12)
((1, 2, 1), 0)
```

```
>>> from sage.all import *
>>> normalize_profile([Integer(1), Integer(2), Integer(1), Integer(0), Integer(0)],
↳precision=Integer(12))
((1, 2, 1), 0)
```

If the first argument is a function, then construct a list of length one less than `precision`, by plugging in the numbers 1, 2, ..., `precision - 1`:

```
sage: normalize_profile(lambda n: 4-n, precision=4)
((3, 2, 1), +Infinity)
sage: normalize_profile(lambda n: 4-n, precision=4, truncation_type=0)
((3, 2, 1), 0)
```

```
>>> from sage.all import *
>>> normalize_profile(lambda n: Integer(4)-n, precision=Integer(4))
((3, 2, 1), +Infinity)
>>> normalize_profile(lambda n: Integer(4)-n, precision=Integer(4), truncation_
↳type=Integer(0))
((3, 2, 1), 0)
```

Negative numbers in profile functions are turned into zeroes:

```
sage: normalize_profile(lambda n: 4-n, precision=6)
((3, 2, 1, 0, 0), +Infinity)
```

```
>>> from sage.all import *
>>> normalize_profile(lambda n: Integer(4)-n, precision=Integer(6))
((3, 2, 1, 0, 0), +Infinity)
```

If it doesn't give a valid profile, an error is raised:

```
sage: normalize_profile(lambda n: 3, precision=4, truncation_type=0)
Traceback (most recent call last):
...
ValueError: Invalid profile
sage: normalize_profile(lambda n: 3, precision=4, truncation_type = Infinity)
((3, 3, 3), +Infinity)
```

```
>>> from sage.all import *
>>> normalize_profile(lambda n: Integer(3), precision=Integer(4), truncation_
↳type=Integer(0))
Traceback (most recent call last):
...
ValueError: Invalid profile
>>> normalize_profile(lambda n: Integer(3), precision=Integer(4), truncation_type_
↳= Infinity)
((3, 3, 3), +Infinity)
```

When p is odd, the behavior is similar:


```
sage: normalize_profile([2,1], [2,2,2]), p=13)
((2, 1), (2, 2, 2)), 0)
```

```
>>> from sage.all import *
>>> normalize_profile([Integer(2),Integer(1)], [Integer(2),Integer(2),
↪Integer(2)]), p=Integer(13))
((2, 1), (2, 2, 2)), 0)
```

The full mod p Steenrod algebra:

```
sage: normalize_profile(None, p=7)
(((), ()), +Infinity)
sage: normalize_profile(Infinity, p=11)
(((), ()), +Infinity)
sage: normalize_profile((lambda n: Infinity, lambda n: 2), p=17)
(((), ()), +Infinity)
```

```
>>> from sage.all import *
>>> normalize_profile(None, p=Integer(7))
(((), ()), +Infinity)
>>> normalize_profile(Infinity, p=Integer(11))
(((), ()), +Infinity)
>>> normalize_profile((lambda n: Infinity, lambda n: Integer(2)), p=Integer(17))
(((), ()), +Infinity)
```

Note that as at the prime 2, the precision argument has no effect on a list or tuple in either entry of profile. If truncation_type is 'auto', then it gets converted to either 0 or +Infinity depending on the *first* entry of profile:

```
sage: normalize_profile([2,1], [2,2,2]), precision=84, p=13)
((2, 1), (2, 2, 2)), 0)
sage: normalize_profile((lambda n: 0, lambda n: 2), precision=4, p=11)
((0, 0, 0), ()), +Infinity)
sage: normalize_profile((lambda n: 0, (1,1,1,1,1,1,1)), precision=4, p=11)
((0, 0, 0), (1, 1, 1, 1, 1, 1, 1)), +Infinity)
sage: normalize_profile((4,3,2,1), lambda n: 2), precision=6, p=11)
((4, 3, 2, 1), (2, 2, 2, 2, 2)), 0)
sage: normalize_profile((4,3,2,1), lambda n: 1), precision=3, p=11, truncation_
↪type=Infinity)
((4, 3, 2, 1), (1, 1)), +Infinity)
```

```
>>> from sage.all import *
>>> normalize_profile([Integer(2),Integer(1)], [Integer(2),Integer(2),
↪Integer(2)]), precision=Integer(84), p=Integer(13))
((2, 1), (2, 2, 2)), 0)
>>> normalize_profile((lambda n: Integer(0), lambda n: Integer(2)), ↪
↪precision=Integer(4), p=Integer(11))
((0, 0, 0), ()), +Infinity)
>>> normalize_profile((lambda n: Integer(0), (Integer(1),Integer(1),Integer(1),
↪Integer(1),Integer(1),Integer(1),Integer(1))), precision=Integer(4), ↪
↪p=Integer(11))
((0, 0, 0), (1, 1, 1, 1, 1, 1, 1)), +Infinity)
```

(continues on next page)

(continued from previous page)

```
>>> normalize_profile(((Integer(4), Integer(3), Integer(2), Integer(1)), lambda n: ~
↳ Integer(2)), precision=Integer(6), p=Integer(11))
((4, 3, 2, 1), (2, 2, 2, 2), 0)
>>> normalize_profile(((Integer(4), Integer(3), Integer(2), Integer(1)), lambda n: ~
↳ Integer(1)), precision=Integer(3), p=Integer(11), truncation_type=Infinity)
((4, 3, 2, 1), (1, 1), +Infinity)
```

As at the prime 2, negative numbers in the first component are converted to zeroes. Numbers in the second component must be either 1 and 2, or else an error is raised:

```
sage: normalize_profile((lambda n: -n, lambda n: 1), precision=4, p=11)
((0, 0, 0), (1, 1, 1), +Infinity)
sage: normalize_profile([0,0,0], [1,2,3,2,1], p=11)
Traceback (most recent call last):
...
ValueError: Invalid profile
```

```
>>> from sage.all import *
>>> normalize_profile((lambda n: -n, lambda n: Integer(1)), precision=Integer(4), ~
↳ p=Integer(11))
((0, 0, 0), (1, 1, 1), +Infinity)
>>> normalize_profile([Integer(0), Integer(0), Integer(0)], [Integer(1), Integer(2),
↳ Integer(3), Integer(2), Integer(1)], p=Integer(11))
Traceback (most recent call last):
...
ValueError: Invalid profile
```

`sage.algebras.steenrod.steenrod_algebra_misc.pst_mono_to_string(mono, latex=False, generic=False)`

String representation of element of a P_t^s -basis.

This is used by the `_repr_` and `_latex_` methods.

INPUT:

- `mono` – tuple of pairs of integers (s,t) with $s \geq 0, t > 0$
- `latex` – boolean (default: False); if True, output LaTeX string
- `generic` – whether to format generically, or for the prime 2 (default)

OUTPUT:

string – concatenation of strings of the form $P^{\{s\}}_{\{t\}}$ for each pair (s,t)

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_misc import pst_mono_to_string
sage: pst_mono_to_string(((1,2), (0,3)), generic=False)
'P^{1}_{2} P^{0}_{3}'
sage: pst_mono_to_string(((1,2), (0,3)), latex=True, generic=False)
'P^{1}_{2} P^{0}_{3}'
sage: pst_mono_to_string(((1,4), ((1,2), 1), ((0,3), 2))), generic=True)
'Q_{1} Q_{4} P^{1}_{2} (P^{0}_{3})^2'
sage: pst_mono_to_string(((1,4), ((1,2), 1), ((0,3), 2))), latex=True, ~
```

(continues on next page)

(continued from previous page)

```

↪generic=True)
'Q_{1} Q_{4} P^{\{1\}_{2}} (P^{\{0\}_{3}})^{\{2\}}'

>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_misc import pst_mono_to_string
>>> pst_mono_to_string(((Integer(1), Integer(2)), (Integer(0), Integer(3))), ↪
↪generic=False)
'P^{\{1\}_{2}} P^{\{0\}_{3}}'
>>> pst_mono_to_string(((Integer(1), Integer(2)), (Integer(0), Integer(3))), ↪
↪latex=True, generic=False)
'P^{\{1\}_{2}} P^{\{0\}_{3}}'
>>> pst_mono_to_string(((Integer(1), Integer(4)), ((Integer(1), Integer(2)), ↪
↪Integer(1)), ((Integer(0), Integer(3)), Integer(2)))), generic=True)
'Q_{1} Q_{4} P^{\{1\}_{2}} (P^{\{0\}_{3}})^{\{2\}}'
>>> pst_mono_to_string(((Integer(1), Integer(4)), ((Integer(1), Integer(2)), ↪
↪Integer(1)), ((Integer(0), Integer(3)), Integer(2)))), latex=True, generic=True)
'Q_{1} Q_{4} P^{\{1\}_{2}} (P^{\{0\}_{3}})^{\{2\}}'

```

The empty tuple represents the unit element:

```

sage: pst_mono_to_string(())
'1'

```

```

>>> from sage.all import *
>>> pst_mono_to_string(())
'1'

```

`sage.algebras.steenrod.steenrod_algebra_misc.serre_cartan_mono_to_string(mono, latex=False, generic=False)`

String representation of element of the Serre-Cartan basis.

This is used by the `_repr_` and `_latex_` methods.

INPUT:

- `mono` – tuple of positive integers (a,b,c,...) when *generic* = *False*, or tuple (e0, n1, e1, n2, ...) when *generic* = *True*, where each *ei* is 0 or 1, and each *ni* is positive
- `latex` – boolean (default: *False*); if *True*, output LaTeX string
- `generic` – whether to format generically, or for the prime 2 (default)

OUTPUT: `rep` – string

This returns a string like `Sq^{a} Sq^{b} Sq^{c} ...` when *generic* = *False*, or a string like `\beta^{e0} P^{n1} \beta^{e1} P^{n2} ...` when *generic* = *True*. is odd.

EXAMPLES:

```

sage: from sage.algebras.steenrod.steenrod_algebra_misc import serre_cartan_mono_
↪to_string
sage: serre_cartan_mono_to_string((1,2,3,4))
'Sq^{\{1\}} Sq^{\{2\}} Sq^{\{3\}} Sq^{\{4\}}'
sage: serre_cartan_mono_to_string((1,2,3,4), latex=True)
'\text{Sq}^{\{1\}} \text{Sq}^{\{2\}} \text{Sq}^{\{3\}} \text{Sq}^{\{4\}}'

```

(continues on next page)

(continued from previous page)

```
sage: serre_cartan_mono_to_string((0,5,1,1,0), generic=True)
'P^{5} beta P^{1}'
sage: serre_cartan_mono_to_string((0,5,1,1,0), generic=True, latex=True)
'\\mathcal{P}^{5} \\beta \\mathcal{P}^{1}'
```

```
>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_misc import serre_cartan_mono_to_
↳ string
>>> serre_cartan_mono_to_string((Integer(1),Integer(2),Integer(3),Integer(4)))
'Sq^{1} Sq^{2} Sq^{3} Sq^{4}'
>>> serre_cartan_mono_to_string((Integer(1),Integer(2),Integer(3),Integer(4)),
↳ latex=True)
'\\text{Sq}^{1} \\text{Sq}^{2} \\text{Sq}^{3} \\text{Sq}^{4}'
>>> serre_cartan_mono_to_string((Integer(0),Integer(5),Integer(1),Integer(1),
↳ Integer(0)), generic=True)
'P^{5} beta P^{1}'
>>> serre_cartan_mono_to_string((Integer(0),Integer(5),Integer(1),Integer(1),
↳ Integer(0)), generic=True, latex=True)
'\\mathcal{P}^{5} \\beta \\mathcal{P}^{1}'
```

The empty tuple represents the unit element 1:

```
sage: serre_cartan_mono_to_string(())
'1'
sage: serre_cartan_mono_to_string((), generic=True)
'1'
```

```
>>> from sage.all import *
>>> serre_cartan_mono_to_string(())
'1'
>>> serre_cartan_mono_to_string((), generic=True)
'1'
```

`sage.algebras.steenrod.steenrod_algebra_misc.wall_long_mono_to_string(mono, latex=False)`

Alternate string representation of element of Wall's basis.

This is used by the `_repr_` and `_latex_` methods.

INPUT:

- `mono` – tuple of pairs of nonnegative integers (m,k) with $m \geq k$
- `latex` – boolean (default: `False`); if `True`, output LaTeX string

OUTPUT:

string – concatenation of strings of the form $Sq^{(2^m)}$

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_misc import wall_long_mono_to_
↳ string
sage: wall_long_mono_to_string(((1,2),(3,0)))
'Sq^{1} Sq^{2} Sq^{4} Sq^{8}'
```

(continues on next page)

(continued from previous page)

```
sage: wall_long_mono_to_string(((1,2),(3,0)), latex=True)
'\\text{Sq}^{\{1\}} \\text{Sq}^{\{2\}} \\text{Sq}^{\{4\}} \\text{Sq}^{\{8\}}'
```

```
>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_misc import wall_long_mono_to_
↳ string
>>> wall_long_mono_to_string(((Integer(1), Integer(2)), (Integer(3), Integer(0))))
'Sq^{\{1\}} Sq^{\{2\}} Sq^{\{4\}} Sq^{\{8\}}'
>>> wall_long_mono_to_string(((Integer(1), Integer(2)), (Integer(3), Integer(0))),
↳ latex=True)
'\\text{Sq}^{\{1\}} \\text{Sq}^{\{2\}} \\text{Sq}^{\{4\}} \\text{Sq}^{\{8\}}'
```

The empty tuple represents the unit element:

```
sage: wall_long_mono_to_string(())
'1'
```

```
>>> from sage.all import *
>>> wall_long_mono_to_string(())
'1'
```

`sage.algebras.steenrod.steenrod_algebra_misc.wall_mono_to_string(mono, latex=False)`

String representation of element of Wall's basis.

This is used by the `_repr_` and `_latex_` methods.

INPUT:

- `mono` – tuple of pairs of nonnegative integers (m,k) with $m \geq k$
- `latex` – boolean (default: `False`); if `True`, output LaTeX string

OUTPUT:

string – concatenation of strings $Q^{\{m\}}_{\{k\}}$ for each pair (m,k)

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_misc import wall_mono_to_string
sage: wall_mono_to_string(((1,2),(3,0)))
'Q^{\{1\}}_{\{2\}} Q^{\{3\}}_{\{0\}}'
sage: wall_mono_to_string(((1,2),(3,0)), latex=True)
'Q^{\{1\}}_{\{2\}} Q^{\{3\}}_{\{0\}}'
```

```
>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_misc import wall_mono_to_string
>>> wall_mono_to_string(((Integer(1), Integer(2)), (Integer(3), Integer(0))))
'Q^{\{1\}}_{\{2\}} Q^{\{3\}}_{\{0\}}'
>>> wall_mono_to_string(((Integer(1), Integer(2)), (Integer(3), Integer(0))),
↳ latex=True)
'Q^{\{1\}}_{\{2\}} Q^{\{3\}}_{\{0\}}'
```

The empty tuple represents the unit element:

```
sage: wall_mono_to_string(())
'1'
```

```
>>> from sage.all import *
>>> wall_mono_to_string(())
'1'
```

`sage.algebras.steenrod.steenrod_algebra_misc.wood_mono_to_string(mono, latex=False)`

String representation of element of Wood's Y and Z bases.

This is used by the `_repr_` and `_latex_` methods.

INPUT:

- `mono` – tuple of pairs of nonnegative integers (s,t)
- `latex` – boolean (default: False); if True, output LaTeX string

OUTPUT:

string – concatenation of strings of the form $Sq^{2^s - (2^{t+1} - 1)}$ for each pair (s,t)

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_misc import wood_mono_to_string
sage: wood_mono_to_string(((1,2),(3,0)))
'Sq^{14} Sq^{8}'
sage: wood_mono_to_string(((1,2),(3,0)), latex=True)
'\\text{Sq}^{14} \\text{Sq}^{8}'
```

```
>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_misc import wood_mono_to_string
>>> wood_mono_to_string(((Integer(1),Integer(2)),(Integer(3),Integer(0))))
'Sq^{14} Sq^{8}'
>>> wood_mono_to_string(((Integer(1),Integer(2)),(Integer(3),Integer(0))),
↳ latex=True)
'\\text{Sq}^{14} \\text{Sq}^{8}'
```

The empty tuple represents the unit element:

```
sage: wood_mono_to_string(())
'1'
```

```
>>> from sage.all import *
>>> wood_mono_to_string(())
'1'
```

5.27 Multiplication for elements of the Steenrod algebra

AUTHORS:

- John H. Palmieri (2008-07-30: version 0.9) initial version: Milnor multiplication.
- John H. Palmieri (2010-06-30: version 1.0) multiplication of Serre-Cartan basis elements using the Adem relations.
- Simon King (2011-10-25): Fix the use of cached functions.

Milnor multiplication, $p = 2$

See Milnor's paper [Mil1958] for proofs, etc.

To multiply Milnor basis elements $\text{Sq}(r_1, r_2, \dots)$ and $\text{Sq}(s_1, s_2, \dots)$ at the prime 2, form all possible matrices M with rows and columns indexed starting at 0, with position (0,0) deleted (or ignored), with s_i equal to the sum of column i for each i , and with r_j equal to the 'weighted' sum of row j . The weights are as follows: elements from column i are multiplied by 2^i . For example, to multiply $\text{Sq}(2)$ and $\text{Sq}(1, 1)$, form the matrices

$$\begin{vmatrix} * & 1 & 1 \\ 2 & 0 & 0 \end{vmatrix} \quad \text{and} \quad \begin{vmatrix} * & 0 & 1 \\ 0 & 1 & 0 \end{vmatrix}$$

(The $*$ is the ignored (0,0)-entry of the matrix.) For each such matrix M , compute a multinomial coefficient, mod 2: for each diagonal $\{m_{ij} : i + j = n\}$, compute $(\sum m_{i,j}!)/(m_{0,n}!m_{1,n-1}!\dots m_{n,0}!)$. Multiply these together for all n . (To compute this mod 2, view the entries of the matrix as their base 2 expansions; then this coefficient is zero if and only if there is some diagonal containing two numbers which have a summand in common in their base 2 expansion. For example, if 3 and 10 are in the same diagonal, the coefficient is zero, because $3 = 1 + 2$ and $10 = 2 + 8$: they both have a summand of 2.)

Now, for each matrix with multinomial coefficient 1, let t_n be the sum of the n -th diagonal in the matrix; then

$$\text{Sq}(r_1, r_2, \dots)\text{Sq}(s_1, s_2, \dots) = \sum \text{Sq}(t_1, t_2, \dots)$$

The function `milnor_multiplication()` takes as input two tuples of nonnegative integers, r and s , which represent $\text{Sq}(r) = \text{Sq}(r_1, r_2, \dots)$ and $\text{Sq}(s) = \text{Sq}(s_1, s_2, \dots)$; it returns as output a dictionary whose keys are tuples $t = (t_1, t_2, \dots)$ of nonnegative integers, and for each tuple the associated value is the coefficient of $\text{Sq}(t)$ in the product formula. (Since we are working mod 2, this coefficient is 1 – if it is zero, the element is omitted from the dictionary altogether).

Milnor multiplication, odd primes

As for the $p = 2$ case, see Milnor's paper [Mil1958] for proofs.

Fix an odd prime p . There are three steps to multiply Milnor basis elements $Q_{f_1}Q_{f_2}\dots\mathcal{P}(q_1, q_2, \dots)$ and $Q_{g_1}Q_{g_2}\dots\mathcal{P}(s_1, s_2, \dots)$: first, use the formula

$$\mathcal{P}(q_1, q_2, \dots)Q_k = Q_k\mathcal{P}(q_1, q_2, \dots) + Q_{k+1}\mathcal{P}(q_1 - p^k, q_2, \dots) + Q_{k+2}\mathcal{P}(q_1, q_2 - p^k, \dots) + \dots$$

Second, use the fact that the Q_k 's form an exterior algebra: $Q_k^2 = 0$ for all k , and if $i \neq j$, then Q_i and Q_j anticommute: $Q_iQ_j = -Q_jQ_i$. After these two steps, the product is a linear combination of terms of the form

$$Q_{e_1}Q_{e_2}\dots\mathcal{P}(r_1, r_2, \dots)\mathcal{P}(s_1, s_2, \dots).$$

Finally, use Milnor matrices to multiply the pairs of $\mathcal{P}(\dots)$ terms, as at the prime 2: form all possible matrices M with rows and columns indexed starting at 0, with position (0,0) deleted (or ignored), with s_i equal to the sum of column i for each i , and with r_j equal to the weighted sum of row j : elements from column i are multiplied by p^i . For example when $p = 5$, to multiply $\mathcal{P}(5)$ and $\mathcal{P}(1, 1)$, form the matrices

$$\begin{vmatrix} * & 1 & 1 \\ 5 & 0 & 0 \end{vmatrix} \quad \text{and} \quad \begin{vmatrix} * & 0 & 1 \\ 0 & 1 & 0 \end{vmatrix}$$

For each such matrix M , compute a multinomial coefficient, mod p : for each diagonal $\{m_{ij} : i + j = n\}$, compute $(\sum m_{i,j}!)/(m_{0,n}!m_{1,n-1}!\dots m_{n,0}!)$. Multiply these together for all n .

Now, for each matrix with nonzero multinomial coefficient b_M , let t_n be the sum of the n -th diagonal in the matrix; then

$$\mathcal{P}(r_1, r_2, \dots)\mathcal{P}(s_1, s_2, \dots) = \sum b_M\mathcal{P}(t_1, t_2, \dots)$$

For example when $p = 5$, we have

$$\mathcal{P}(5)\mathcal{P}(1, 1) = \mathcal{P}(6, 1) + 2\mathcal{P}(0, 2).$$

The function `milnor_multiplication()` takes as input two pairs of tuples of nonnegative integers, (g, q) and (f, s) , which represent $Q_{g_1} Q_{g_2} \dots \mathcal{P}(q_1, q_2, \dots)$ and $Q_{f_1} Q_{f_2} \dots \mathcal{P}(s_1, s_2, \dots)$. It returns as output a dictionary whose keys are pairs of tuples (e, t) of nonnegative integers, and for each tuple the associated value is the coefficient in the product formula.

The Adem relations and admissible sequences

If $p = 2$, then the mod 2 Steenrod algebra is generated by Steenrod squares Sq^a for $a \geq 0$ (equal to the Milnor basis element $Sq(a)$). The *Adem relations* are as follows: if $a < 2b$,

$$Sq^a Sq^b = \sum_{j=0}^{a/2} \binom{b-j-1}{a-2j} Sq^{a+b-j} Sq^j$$

A monomial $Sq^{i_1} Sq^{i_2} \dots Sq^{i_n}$ is called *admissible* if $i_k \geq 2i_{k+1}$ for all k . One can use the Adem relations to show that the admissible monomials span the Steenrod algebra, as a vector space; with more work, one can show that the admissible monomials are also linearly independent. They form the *Serre-Cartan* basis for the Steenrod algebra. To multiply a collection of admissible monomials, concatenate them and see if the result is admissible. If it is, you're done. If not, find the first pair $Sq^a Sq^b$ where it fails to be admissible and apply the Adem relations there. Repeat with the resulting terms. One can prove that this process terminates in a finite number of steps, and therefore gives a procedure for multiplying elements of the Serre-Cartan basis.

At an odd prime p , the Steenrod algebra is generated by the p -th power operations $\mathcal{P}^a$ (the same as $\mathcal{P}(a)$ in the Milnor basis) and the Bockstein operation β ($= Q_0$ in the Milnor basis). The odd primary *Adem relations* are as follows: if $a < pb$,

$$\mathcal{P}^a \mathcal{P}^b = \sum_{j=0}^{a/p} (-1)^{a+j} \binom{(b-j)(p-1)-1}{a-pj} \mathcal{P}^{a+b-j} \mathcal{P}^j$$

Also, if $a \leq pb$,

$$\mathcal{P}^a \beta \mathcal{P}^b = \sum_{j=0}^{a/p} (-1)^{a+j} \binom{(b-j)(p-1)}{a-pj} \beta \mathcal{P}^{a+b-j} \mathcal{P}^j + \sum_{j=0}^{a/p} (-1)^{a+j-1} \binom{(b-j)(p-1)-1}{a-pj-1} \mathcal{P}^{a+b-j} \beta \mathcal{P}^j$$

The *admissible* monomials at an odd prime are products of the form

$$\beta^{\epsilon_0} \mathcal{P}^{s_1} \beta^{\epsilon_1} \mathcal{P}^{s_2} \dots \mathcal{P}^{s_n} \beta^{\epsilon_n}$$

where $s_k \geq \epsilon_{k+1} + ps_{k+1}$ for all k . As at the prime 2, these form a basis for the Steenrod algebra.

The main function for this is `make_mono_admissible()`, which converts a product of Steenrod squares or p -th power operations and Bocksteins into a dictionary representing a sum of admissible monomials.

```
sage.algebras.steenrod.steenrod_algebra_mult.adem(b, c=0, p=2, generic=None)
```

The mod p Adem relations.

INPUT:

- a, b, c – nonnegative integers (optional); corresponding to either $P^a P^b$ or (if c present) to $P^a \beta^b P^c$
- p – positive prime number (default: 2)
- `generic` – whether to use the generic Steenrod algebra, (default: depends on prime)

OUTPUT:

a dictionary representing the mod p Adem relations applied to $P^a P^b$ or (if c present) to $P^a \beta^b P^c$.

The mod p Adem relations for the mod p Steenrod algebra are as follows: if $p = 2$, then if $a < 2b$,

$$\mathrm{Sq}^a \mathrm{Sq}^b = \sum_{j=0}^{a/2} \binom{b-j-1}{a-2j} \mathrm{Sq}^{a+b-j} \mathrm{Sq}^j$$

If p is odd, then if $a < pb$,

$$P^a P^b = \sum_{j=0}^{a/p} (-1)^{a+j} \binom{(b-j)(p-1)-1}{a-pj} P^{a+b-j} P^j$$

Also for p odd, if $a \leq pb$,

$$P^a \beta P^b = \sum_{j=0}^{a/p} (-1)^{a+j} \binom{(b-j)(p-1)}{a-pj} \beta P^{a+b-j} P^j + \sum_{j=0}^{a/p} (-1)^{a+j-1} \binom{(b-j)(p-1)-1}{a-pj-1} P^{a+b-j} \beta P^j$$

EXAMPLES:

If two arguments (a and b) are given, then computations are done mod 2. If $a \geq 2b$, then the dictionary $\{(a,b): 1\}$ is returned. Otherwise, the right side of the mod 2 Adem relation for $\mathrm{Sq}^a \mathrm{Sq}^b$ is returned. For example, since $\mathrm{Sq}^2 \mathrm{Sq}^2 = \mathrm{Sq}^3 \mathrm{Sq}^1$, we have:

```
sage: from sage.algebras.steenrod.steenrod_algebra_mult import adem
sage: adem(2,2) # indirect doctest
{(3, 1): 1}
sage: adem(4,2)
{(4, 2): 1}
sage: adem(4,4) == {(6, 2): 1, (7, 1): 1}
True
```

```
>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_mult import adem
>>> adem(Integer(2), Integer(2)) # indirect doctest
{(3, 1): 1}
>>> adem(Integer(4), Integer(2))
{(4, 2): 1}
>>> adem(Integer(4), Integer(4)) == {(Integer(6), Integer(2)): Integer(1),
↳ (Integer(7), Integer(1)): Integer(1)}
True
```

If p is given and is odd, then with two inputs a and b , the Adem relation for $P^a P^b$ is computed. With three inputs a, b, c , the Adem relation for $P^a \beta^b P^c$ is computed. In either case, the keys in the output are all tuples of odd length, with $(i_1, i_2, \dots, i_m)$ representing

$$\beta^{i_1} P^{i_2} \beta^{i_3} P^{i_4} \dots \beta^{i_m}$$

For instance:

```
sage: adem(3,1, p=3)
{(0, 3, 0, 1, 0): 1}
sage: adem(3,0,1, p=3)
{(0, 3, 0, 1, 0): 1}
sage: adem(1,0,1, p=7)
{(0, 2, 0): 2}
```

(continues on next page)

(continued from previous page)

```
sage: adem(1,1,1, p=5) == {(0, 2, 1): 1, (1, 2, 0): 1}
True
sage: adem(1,1,2, p=5) == {(0, 3, 1): 1, (1, 3, 0): 2}
True
```

```
>>> from sage.all import *
>>> adem(Integer(3), Integer(1), p=Integer(3))
{(0, 3, 0, 1, 0): 1}
>>> adem(Integer(3), Integer(0), Integer(1), p=Integer(3))
{(0, 3, 0, 1, 0): 1}
>>> adem(Integer(1), Integer(0), Integer(1), p=Integer(7))
{(0, 2, 0): 2}
>>> adem(Integer(1), Integer(1), Integer(1), p=Integer(5)) == {(Integer(0), ↵
↵Integer(2), Integer(1)): Integer(1), (Integer(1), Integer(2), Integer(0)): ↵
↵Integer(1)}
True
>>> adem(Integer(1), Integer(1), Integer(2), p=Integer(5)) == {(Integer(0), ↵
↵Integer(3), Integer(1)): Integer(1), (Integer(1), Integer(3), Integer(0)): ↵
↵Integer(2)}
True
```

sage.algebras.steenrod.steenrod_algebra_mult.**binomial_mod2**(n, k)

The binomial coefficient $\binom{n}{k}$, computed mod 2.

INPUT:

- n, k – integers

OUTPUT: n choose k , mod 2

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_mult import binomial_mod2
sage: binomial_mod2(4, 2)
0
sage: binomial_mod2(5, 4)
1
sage: binomial_mod2(3 * 32768, 32768)
1
sage: binomial_mod2(4 * 32768, 32768)
0
```

```
>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_mult import binomial_mod2
>>> binomial_mod2(Integer(4), Integer(2))
0
>>> binomial_mod2(Integer(5), Integer(4))
1
>>> binomial_mod2(Integer(3) * Integer(32768), Integer(32768))
1
>>> binomial_mod2(Integer(4) * Integer(32768), Integer(32768))
0
```

sage.algebras.steenrod.steenrod_algebra_mult.**binomial_modp**(n, k, p)

The binomial coefficient $\binom{n}{k}$, computed mod p .

INPUT:

- n, k – integers
- p – prime number

OUTPUT: n choose k , mod p

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_mult import binomial_modp
sage: binomial_modp(5, 2, 3)
1
sage: binomial_modp(6, 2, 11) # 6 choose 2 = 15
4
```

```
>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_mult import binomial_modp
>>> binomial_modp(Integer(5), Integer(2), Integer(3))
1
>>> binomial_modp(Integer(6), Integer(2), Integer(11)) # 6 choose 2 = 15
4
```

`sage.algebras.steenrod.steenrod_algebra_mult.make_mono_admissible(p=2, generic=None)`

Given a tuple `mono`, view it as a product of Steenrod operations, and return a dictionary giving data equivalent to writing that product as a linear combination of admissible monomials.

When $p = 2$, the sequence (and hence the corresponding monomial) $(i_1, i_2, \dots)$ is admissible if $i_j \geq 2i_{j+1}$ for all j .

When p is odd, the sequence $(e_1, i_1, e_2, i_2, \dots)$ is admissible if $i_j \geq e_{j+1} + pi_{j+1}$ for all j .

INPUT:

- `mono` – tuple of nonnegative integers
- p – prime number (default: 2)
- `generic` – whether to use the generic Steenrod algebra (default: depends on prime)

OUTPUT:

Dictionary of terms of the form (tuple: coeff), where ‘tuple’ is an admissible tuple of nonnegative integers and ‘coeff’ is its coefficient. This corresponds to a linear combination of admissible monomials. When p is odd, each tuple must have an odd length: it should be of the form $(e_1, i_1, e_2, i_2, \dots, e_k)$ where each e_j is either 0 or 1 and each i_j is a positive integer: this corresponds to the admissible monomial

$$\beta^{e_1} \mathcal{P}^{i_1} \beta^{e_2} \mathcal{P}^{i_2} \dots \mathcal{P}^{i_k} \beta^{e_k}$$

ALGORITHM:

Given $(i_1, i_2, i_3, \dots)$, apply the Adem relations to the first pair (or triple when p is odd) where the sequence is inadmissible, and then apply this function recursively to each of the resulting tuples $(i_1, \dots, i_{j-1}, NEW, i_{j+2}, \dots)$, keeping track of the coefficients.

EXAMPLES:

```

sage: from sage.algebras.steenrod.steenrod_algebra_mult import make_mono_
      ↪admissible
sage: make_mono_admissible((12,)) # already admissible, indirect doctest
{(12,): 1}
sage: make_mono_admissible((2,1)) # already admissible
{(2, 1): 1}
sage: make_mono_admissible((2,2))
{(3, 1): 1}
sage: make_mono_admissible((2, 2, 2))
{(5, 1): 1}
sage: make_mono_admissible((0, 2, 0, 1, 0), p=7)
{(0, 3, 0): 3}

```

```

>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_mult import make_mono_admissible
>>> make_mono_admissible((Integer(12),)) # already admissible, indirect doctest
{(12,): 1}
>>> make_mono_admissible((Integer(2),Integer(1))) # already admissible
{(2, 1): 1}
>>> make_mono_admissible((Integer(2),Integer(2)))
{(3, 1): 1}
>>> make_mono_admissible((Integer(2), Integer(2), Integer(2)))
{(5, 1): 1}
>>> make_mono_admissible((Integer(0), Integer(2), Integer(0), Integer(1),
      ↪Integer(0)), p=Integer(7))
{(0, 3, 0): 3}

```

Test the fix from [Issue #13796](#):

```

sage: SteenrodAlgebra(p=2, basis='adem').Q(2) * (Sq(6) * Sq(2)) # indirect doctest
Sq^10 Sq^4 Sq^1 + Sq^10 Sq^5 + Sq^12 Sq^3 + Sq^13 Sq^2

```

```

>>> from sage.all import *
>>> SteenrodAlgebra(p=Integer(2), basis='adem').Q(Integer(2)) * (Sq(Integer(6)) *
      ↪Sq(Integer(2))) # indirect doctest
Sq^10 Sq^4 Sq^1 + Sq^10 Sq^5 + Sq^12 Sq^3 + Sq^13 Sq^2

```

`sage.algebras.steenrod.steenrod_algebra_mult.milnor_multiplication(r,s)`

Product of Milnor basis elements r and s at the prime 2.

INPUT:

- r – tuple of nonnegative integers
- s – tuple of nonnegative integers

OUTPUT:

Dictionary of terms of the form (tuple: coeff), where ‘tuple’ is a tuple of nonnegative integers and ‘coeff’ is 1.

This computes Milnor matrices for the product of $Sq(r)$ and $Sq(s)$, computes their multinomial coefficients, and for each matrix whose coefficient is 1, add $Sq(t)$ to the output, where t is the tuple formed by the diagonals sums from the matrix.

EXAMPLES:

```

sage: from sage.algebras.steenrod.steenrod_algebra_mult import milnor_
      ↪multiplication
sage: milnor_multiplication((2,), (1,)) == {(0, 1): 1, (3,): 1}
True
sage: sorted(milnor_multiplication((4,), (2,1)).items())
[(0, 3), 1], [(2, 0, 1), 1], [(6, 1), 1]
sage: sorted(milnor_multiplication((2,4), (0,1)).items())
[(2, 0, 0, 1), 1], [(2, 5), 1]

```

```

>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_mult import milnor_multiplication
>>> milnor_multiplication((Integer(2),), (Integer(1),)) == {(Integer(0), ↪
      ↪Integer(1)): Integer(1), (Integer(3,)): Integer(1)}
True
>>> sorted(milnor_multiplication((Integer(4),), (Integer(2),Integer(1))).items())
[(0, 3), 1], [(2, 0, 1), 1], [(6, 1), 1]
>>> sorted(milnor_multiplication((Integer(2),Integer(4)), (Integer(0),
      ↪Integer(1))).items())
[(2, 0, 0, 1), 1], [(2, 5), 1]

```

These examples correspond to the following product computations:

$$\begin{aligned}
 \text{Sq}(2)\text{Sq}(1) &= \text{Sq}(0,1) + \text{Sq}(3) \\
 \text{Sq}(4)\text{Sq}(2,1) &= \text{Sq}(6,1) + \text{Sq}(0,3) + \text{Sq}(2,0,1) \\
 \text{Sq}(2,4)\text{Sq}(0,1) &= \text{Sq}(2,5) + \text{Sq}(2,0,0,1)
 \end{aligned}$$

This uses the same algorithm Monks does in his Maple package: see <http://mathweb.scranton.edu/monks/software/Steenrod/steen.html>.

`sage.algebras.steenrod.steenrod_algebra_mult.milnor_multiplication_odd(m1, m2, p)`

Product of Milnor basis elements defined by m1 and m2 at the odd prime p.

INPUT:

- m1 – pair of tuples (e,r), where e is an increasing tuple of nonnegative integers and r is a tuple of nonnegative integers
- m2 – pair of tuples (f,s), same format as m1
- p – odd prime number

OUTPUT:

Dictionary of terms of the form (tuple: coeff), where ‘tuple’ is a pair of tuples, as for r and s, and ‘coeff’ is an integer mod p.

This computes the product of the Milnor basis elements $Q_{e_1}Q_{e_2}\dots P(r_1, r_2, \dots)$ and $Q_{f_1}Q_{f_2}\dots P(s_1, s_2, \dots)$.

EXAMPLES:

```

sage: from sage.algebras.steenrod.steenrod_algebra_mult import milnor_
      ↪multiplication_odd
sage: sorted(milnor_multiplication_odd(((0,2),(5,)), ((1,),(1,)), 5).items())
[(0, 1, 2), (0, 1)), 4], [(0, 1, 2), (6,)), 4]
sage: milnor_multiplication_odd(((0,2,4),()), ((1,3),()), 7)
{((0, 1, 2, 3, 4), ()): 6}
sage: milnor_multiplication_odd(((0,2,4),()), ((1,5),()), 7)

```

(continues on next page)

(continued from previous page)

```
{((0, 1, 2, 4, 5), ()): 1}
sage: sorted(milnor_multiplication_odd(((), (6,)), ((), (2,)), 3).items())
[((((), (0, 2)), 1), ((((), (4, 1)), 1), ((((), (8,)), 1))]
```

```
>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_mult import milnor_
    ↪multiplication_odd
>>> sorted(milnor_multiplication_odd(((Integer(0), Integer(2)), (Integer(5),)), ↪
    ↪((Integer(1),), (Integer(1),)), Integer(5)).items())
[(((0, 1, 2), (0, 1)), 4), (((0, 1, 2), (6,)), 4)]
>>> milnor_multiplication_odd(((Integer(0), Integer(2), Integer(4)), ()), ↪
    ↪((Integer(1), Integer(3)), ()), Integer(7))
{((0, 1, 2, 3, 4), ()): 6}
>>> milnor_multiplication_odd(((Integer(0), Integer(2), Integer(4)), ()), ↪
    ↪((Integer(1), Integer(5)), ()), Integer(7))
{((0, 1, 2, 4, 5), ()): 1}
>>> sorted(milnor_multiplication_odd(((), (Integer(6,)), ((), (Integer(2,)), ↪
    ↪Integer(3)).items())
[((((), (0, 2)), 1), ((((), (4, 1)), 1), ((((), (8,)), 1))]
```

These examples correspond to the following product computations:

$$\begin{aligned}
 p = 5: \quad Q_0 Q_2 \mathcal{P}(5) Q_1 \mathcal{P}(1) &= 4Q_0 Q_1 Q_2 \mathcal{P}(0, 1) + 4Q_0 Q_1 Q_2 \mathcal{P}(6) \\
 p = 7: \quad (Q_0 Q_2 Q_4)(Q_1 Q_3) &= 6Q_0 Q_1 Q_2 Q_3 Q_4 \\
 p = 7: \quad (Q_0 Q_2 Q_4)(Q_1 Q_5) &= Q_0 Q_1 Q_2 Q_3 Q_5 \\
 p = 3: \quad \mathcal{P}(6) \mathcal{P}(2) &= \mathcal{P}(0, 2) + \mathcal{P}(4, 1) + \mathcal{P}(8)
 \end{aligned}$$

The following used to fail until the trailing zeroes were eliminated in `p_mono`:

```
sage: A = SteenrodAlgebra(3)
sage: a = A.P(0, 3); b = A.P(12); c = A.Q(1, 2)
sage: (a+b)*c == a*c + b*c
True
```

```
>>> from sage.all import *
>>> A = SteenrodAlgebra(Integer(3))
>>> a = A.P(Integer(0), Integer(3)); b = A.P(Integer(12)); c = A.Q(Integer(1),
    ↪Integer(2))
>>> (a+b)*c == a*c + b*c
True
```

Test that the bug reported in [Issue #7212](#) has been fixed:

```
sage: A.P(36, 6)*A.P(27, 9, 81)
2 P(13, 21, 83) + P(14, 24, 82) + P(17, 20, 83) + P(25, 18, 83) + P(26, 21, 82) + P(36, 15,
    ↪80, 1) + P(49, 12, 83) + 2 P(50, 15, 82) + 2 P(53, 11, 83) + 2 P(63, 15, 81)
```

```
>>> from sage.all import *
>>> A.P(Integer(36), Integer(6))*A.P(Integer(27), Integer(9), Integer(81))
2 P(13, 21, 83) + P(14, 24, 82) + P(17, 20, 83) + P(25, 18, 83) + P(26, 21, 82) + P(36, 15,
    ↪80, 1) + P(49, 12, 83) + 2 P(50, 15, 82) + 2 P(53, 11, 83) + 2 P(63, 15, 81)
```

Associativity once failed because of a sign error:

```
sage: a,b,c = A.Q_exp(0,1), A.P(3), A.Q_exp(1,1)
sage: (a*b)*c == a*(b*c)
True
```

```
>>> from sage.all import *
>>> a,b,c = A.Q_exp(Integer(0),Integer(1)), A.P(Integer(3)), A.Q_exp(Integer(1),
↪Integer(1))
>>> (a*b)*c == a*(b*c)
True
```

This uses the same algorithm Monks does in his Maple package to iterate through the possible matrices: see <http://mathweb.scranton.edu/monks/software/Steenrod/steen.html>.

sage.algebras.steenrod.steenrod_algebra_mult.multinomial(list)

Multinomial coefficient of list, mod 2.

INPUT:

- list – list of integers

OUTPUT: none if the multinomial coefficient is 0, or sum of list if it is 1

Given the input $[n_1, n_2, n_3, \dots]$, this computes the multinomial coefficient $(n_1 + n_2 + n_3 + \dots)! / (n_1! n_2! n_3! \dots)$, mod 2. The method is roughly this: expand each n_i in binary. If there is a 1 in the same digit for any n_i and n_j with $i \neq j$, then the coefficient is 0; otherwise, it is 1.

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_mult import multinomial
sage: multinomial([1,2,4])
7
sage: multinomial([1,2,5])
sage: multinomial([1,2,12,192,256])
463
```

```
>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_mult import multinomial
>>> multinomial([Integer(1),Integer(2),Integer(4)])
7
>>> multinomial([Integer(1),Integer(2),Integer(5)])
>>> multinomial([Integer(1),Integer(2),Integer(12),Integer(192),Integer(256)])
463
```

This function does not compute any factorials, so the following are actually reasonable to do:

```
sage: multinomial([1,65536])
65537
sage: multinomial([4,65535])
sage: multinomial([32768,65536])
98304
```

```
>>> from sage.all import *
>>> multinomial([Integer(1),Integer(65536)])
65537
>>> multinomial([Integer(4),Integer(65535)])
```

(continues on next page)

(continued from previous page)

```
>>> multinomial([Integer(32768), Integer(65536)])
98304
```

```
sage.algebras.steenrod.steenrod_algebra_mult.multinomial_odd(list, p)
```

Multinomial coefficient of list, mod p .

INPUT:

- `list` – list of integers
- `p` – a prime number

OUTPUT: associated multinomial coefficient, mod p

Given the input $[n_1, n_2, n_3, \dots]$, this computes the multinomial coefficient $(n_1 + n_2 + n_3 + \dots)! / (n_1! n_2! n_3! \dots)$, mod p . The method is this: expand each n_i in base p : $n_i = \sum_j p^j n_{ij}$. Do the same for the sum of the n_i 's, which we call m : $m = \sum_j p^j m_j$. Then the multinomial coefficient is congruent, mod p , to the product of the multinomial coefficients $m_j! / (n_{1j}! n_{2j}! \dots)$.

Furthermore, any multinomial coefficient $m! / (n_1! n_2! \dots)$ can be computed as a product of binomial coefficients: it equals

$$\binom{n_1}{n_1} \binom{n_1 + n_2}{n_2} \binom{n_1 + n_2 + n_3}{n_3} \dots$$

This is convenient because Sage's binomial function returns integers, not rational numbers (as would be produced just by dividing factorials).

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_mult import multinomial_odd
sage: multinomial_odd([1, 2, 4], 2)
1
sage: multinomial_odd([1, 2, 4], 7)
0
sage: multinomial_odd([1, 2, 4], 11)
6
sage: multinomial_odd([1, 2, 4], 101)
4
sage: multinomial_odd([1, 2, 4], 107)
105
```

```
>>> from sage.all import *
>>> from sage.algebras.steenrod.steenrod_algebra_mult import multinomial_odd
>>> multinomial_odd([Integer(1), Integer(2), Integer(4)], Integer(2))
1
>>> multinomial_odd([Integer(1), Integer(2), Integer(4)], Integer(7))
0
>>> multinomial_odd([Integer(1), Integer(2), Integer(4)], Integer(11))
6
>>> multinomial_odd([Integer(1), Integer(2), Integer(4)], Integer(101))
4
>>> multinomial_odd([Integer(1), Integer(2), Integer(4)], Integer(107))
105
```


5.28 Weyl Algebras

AUTHORS:

- Travis Scrimshaw (2013-09-06): Initial version

class sage.algebras.weyl_algebra.DifferentialWeylAlgebra(*R*, *names=None*)

Bases: UniqueRepresentation, Parent

The differential Weyl algebra of a polynomial ring.

Let R be a commutative ring. The (differential) Weyl algebra W is the algebra generated by $x_1, x_2, \dots, x_n, \partial_{x_1}, \partial_{x_2}, \dots, \partial_{x_n}$ subject to the relations: $[x_i, x_j] = 0$, $[\partial_{x_i}, \partial_{x_j}] = 0$, and $\partial_{x_i} x_j = x_j \partial_{x_i} + \delta_{ij}$. Therefore ∂_{x_i} is acting as the partial differential operator on x_i .

The Weyl algebra can also be constructed as an iterated Ore extension of the polynomial ring $R[x_1, x_2, \dots, x_n]$ by adding x_i at each step. It can also be seen as a quantization of the symmetric algebra $Sym(V)$, where V is a finite dimensional vector space over a field of characteristic zero, by using a modified Groenewold-Moyal product in the symmetric algebra.

The Weyl algebra (even for $n = 1$) over a field of characteristic 0 has many interesting properties.

- It's a non-commutative domain.
- It's a simple ring (but not in positive characteristic) that is not a matrix ring over a division ring.
- It has no finite-dimensional representations.
- It's a quotient of the universal enveloping algebra of the Heisenberg algebra $\mathfrak{h}_n$.

REFERENCES:

- [Wikipedia article Weyl algebra](#)

INPUT:

- R – a (polynomial) ring
- *names* – (default: None) if None and R is a polynomial ring, then the variable names correspond to those of R ; otherwise if *names* is specified, then R is the base ring

EXAMPLES:

There are two ways to create a Weyl algebra, the first is from a polynomial ring:

```
sage: R.<x,y,z> = QQ[]
sage: W = DifferentialWeylAlgebra(R); W
Differential Weyl algebra of polynomials in x, y, z over Rational Field
```

```
>>> from sage.all import *
>>> R = QQ['x, y, z']; (x, y, z) = R._first_ngens(3)
>>> W = DifferentialWeylAlgebra(R); W
Differential Weyl algebra of polynomials in x, y, z over Rational Field
```

We can call `W.inject_variables()` to give the polynomial ring variables, now as elements of W , and the differentials:

```
sage: W.inject_variables()
Defining x, y, z, dx, dy, dz
sage: (dx * dy * dz) * (x^2 * y * z + x * z * dy + 1)
x*z*dx*dy^2*dz + z*dy^2*dz + x^2*y*z*dx*dy*dz + dx*dy*dz
```

(continues on next page)

(continued from previous page)

```
+ x*dx*dy^2 + 2*x*y*z*dy*dz + dy^2 + x^2*z*dx*dz + x^2*y*dx*dy
+ 2*x*z*dz + 2*x*y*dy + x^2*dx + 2*x
```

```
>>> from sage.all import *
>>> W.inject_variables()
Defining x, y, z, dx, dy, dz
>>> (dx * dy * dz) * (x**Integer(2) * y * z + x * z * dy + Integer(1))
x*z*dx*dy^2*dz + z*dy^2*dz + x^2*y*z*dx*dy*dz + dx*dy*dz
+ x*dx*dy^2 + 2*x*y*z*dy*dz + dy^2 + x^2*z*dx*dz + x^2*y*dx*dy
+ 2*x*z*dz + 2*x*y*dy + x^2*dx + 2*x
```

Or directly by specifying a base ring and variable names:

```
sage: W.<a,b> = DifferentialWeylAlgebra(QQ); W
Differential Weyl algebra of polynomials in a, b over Rational Field
```

```
>>> from sage.all import *
>>> W = DifferentialWeylAlgebra(QQ, names=('a', 'b',)); (a, b,) = W._first_
↳ngens(2); W
Differential Weyl algebra of polynomials in a, b over Rational Field
```

Todo

Implement the `graded_algebra()` as a polynomial ring once they are considered to be graded rings (algebras).

Element

alias of *DifferentialWeylAlgebraElement*

algebra_generators()

Return the algebra generators of `self`.

See also

variables(), *differentials()*

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: W = DifferentialWeylAlgebra(R)
sage: W.algebra_generators()
Finite family {'x': x, 'y': y, 'z': z, 'dx': dx, 'dy': dy, 'dz': dz}
```

```
>>> from sage.all import *
>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> W = DifferentialWeylAlgebra(R)
>>> W.algebra_generators()
Finite family {'x': x, 'y': y, 'z': z, 'dx': dx, 'dy': dy, 'dz': dz}
```

basis()

Return a basis of `self`.

EXAMPLES:

```
sage: W.<x,y> = DifferentialWeylAlgebra(QQ)
sage: B = W.basis()
sage: it = iter(B)
sage: [next(it) for i in range(20)]
[1, x, y, dx, dy, x^2, x*y, x*dx, x*dy, y^2, y*dx, y*dy,
 dx^2, dx*dy, dy^2, x^3, x^2*y, x^2*dx, x^2*dy, x*y^2]
sage: dx, dy = W.differentials()
sage: sorted((dx*x).monomials(), key=str)
[1, x*dx]
sage: B[(x*y).support()[0]]
x*y
sage: sorted((dx*x).monomial_coefficients().items())
[((0, 0), (0, 0)), 1), ((1, 0), (1, 0)), 1]]
```

```
>>> from sage.all import *
>>> W = DifferentialWeylAlgebra(QQ, names=('x', 'y')); (x, y) = W._first_
↳ngens(2)
>>> B = W.basis()
>>> it = iter(B)
>>> [next(it) for i in range(Integer(20))]
[1, x, y, dx, dy, x^2, x*y, x*dx, x*dy, y^2, y*dx, y*dy,
 dx^2, dx*dy, dy^2, x^3, x^2*y, x^2*dx, x^2*dy, x*y^2]
>>> dx, dy = W.differentials()
>>> sorted((dx*x).monomials(), key=str)
[1, x*dx]
>>> B[(x*y).support()[Integer(0)]]
x*y
>>> sorted((dx*x).monomial_coefficients().items())
[((0, 0), (0, 0)), 1), ((1, 0), (1, 0)), 1]]
```

degree_on_basis(i)

Return the degree of the basis element indexed by `i`.

EXAMPLES:

```
sage: W.<a,b> = DifferentialWeylAlgebra(QQ)
sage: W.degree_on_basis( ((1, 3, 2), (0, 1, 3)) )
10

sage: W.<x,y,z> = DifferentialWeylAlgebra(QQ)
sage: dx,dy,dz = W.differentials()
sage: elt = y*dy - (3*x - z)*dx
sage: elt.degree()
2
```

```
>>> from sage.all import *
>>> W = DifferentialWeylAlgebra(QQ, names=('a', 'b')); (a, b) = W._first_
↳ngens(2)
>>> W.degree_on_basis( ((Integer(1), Integer(3), Integer(2)), (Integer(0),
```

(continues on next page)

(continued from previous page)

```

↪Integer(1), Integer(3))) )
10

>>> W = DifferentialWeylAlgebra(QQ, names=('x', 'y', 'z',)); (x, y, z,) = W._
↪first_ngens(3)
>>> dx, dy, dz = W.differentials()
>>> elt = y*dy - (Integer(3)*x - z)*dx
>>> elt.degree()
2

```

diff_action()

Left action of this Weyl algebra on the underlying polynomial ring by differentiation.

EXAMPLES:

```

sage: R.<x,y> = QQ[]
sage: W = R.weyl_algebra()
sage: dx, dy = W.differentials()
sage: W.diff_action
Left action by Differential Weyl algebra of polynomials in x, y
over Rational Field on Multivariate Polynomial Ring in x, y over
Rational Field
sage: W.diff_action(dx^2 + dy + 1, x^3*y^3)
x^3*y^3 + 3*x^3*y^2 + 6*x*y^3

```

```

>>> from sage.all import *
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> W = R.weyl_algebra()
>>> dx, dy = W.differentials()
>>> W.diff_action
Left action by Differential Weyl algebra of polynomials in x, y
over Rational Field on Multivariate Polynomial Ring in x, y over
Rational Field
>>> W.diff_action(dx**Integer(2) + dy + Integer(1),
↪x**Integer(3)*y**Integer(3))
x^3*y^3 + 3*x^3*y^2 + 6*x*y^3

```

differentials()

Return the differentials of self.

See also

`algebra_generators()`, `variables()`

EXAMPLES:

```

sage: W.<x,y,z> = DifferentialWeylAlgebra(QQ)
sage: W.differentials()
Finite family {'dx': dx, 'dy': dy, 'dz': dz}

```

```
>>> from sage.all import *
>>> W = DifferentialWeylAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z) = W._
↳first_ngens(3)
>>> W.differentials()
Finite family {'dx': dx, 'dy': dy, 'dz': dz}
```

gen(i)

Return the i -th generator of `self`.

See also

`algebra_generators()`

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: W = DifferentialWeylAlgebra(R)
sage: [W.gen(i) for i in range(6)]
[x, y, z, dx, dy, dz]
```

```
>>> from sage.all import *
>>> R = QQ['x, y, z']; (x, y, z) = R._first_ngens(3)
>>> W = DifferentialWeylAlgebra(R)
>>> [W.gen(i) for i in range(Integer(6))]
[x, y, z, dx, dy, dz]
```

gens()

Return the algebra generators of `self`.

See also

`variables()`, `differentials()`

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: W = DifferentialWeylAlgebra(R)
sage: W.algebra_generators()
Finite family {'x': x, 'y': y, 'z': z, 'dx': dx, 'dy': dy, 'dz': dz}
```

```
>>> from sage.all import *
>>> R = QQ['x, y, z']; (x, y, z) = R._first_ngens(3)
>>> W = DifferentialWeylAlgebra(R)
>>> W.algebra_generators()
Finite family {'x': x, 'y': y, 'z': z, 'dx': dx, 'dy': dy, 'dz': dz}
```

ngens()

Return the number of generators of `self`.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: W = DifferentialWeylAlgebra(R)
sage: W.ngens()
6
```

```
>>> from sage.all import *
>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> W = DifferentialWeylAlgebra(R)
>>> W.ngens()
6
```

one()

Return the multiplicative identity element 1.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: W = DifferentialWeylAlgebra(R)
sage: W.one()
1
```

```
>>> from sage.all import *
>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> W = DifferentialWeylAlgebra(R)
>>> W.one()
1
```

options = Current options for DifferentialWeylAlgebra - factor_representation:
False

polynomial_ring()

Return the associated polynomial ring of self.

EXAMPLES:

```
sage: W.<a,b> = DifferentialWeylAlgebra(QQ)
sage: W.polynomial_ring()
Multivariate Polynomial Ring in a, b over Rational Field
```

```
>>> from sage.all import *
>>> W = DifferentialWeylAlgebra(QQ, names=('a', 'b',)); (a, b,) = W._first_
↪ngens(2)
>>> W.polynomial_ring()
Multivariate Polynomial Ring in a, b over Rational Field
```

```
sage: R.<x,y,z> = QQ[]
sage: W = DifferentialWeylAlgebra(R)
sage: W.polynomial_ring() == R
True
```

```
>>> from sage.all import *
>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
```

(continues on next page)

(continued from previous page)

```
>>> W = DifferentialWeylAlgebra(R)
>>> W.polynomial_ring() == R
True
```

variables()

Return the variables of self.

See also*algebra_generators(), differentials()***EXAMPLES:**

```
sage: W.<x,y,z> = DifferentialWeylAlgebra(QQ)
sage: W.variables()
Finite family {'x': x, 'y': y, 'z': z}
```

```
>>> from sage.all import *
>>> W = DifferentialWeylAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z) = W._
↳first_ngens(3)
>>> W.variables()
Finite family {'x': x, 'y': y, 'z': z}
```

zero()

Return the additive identity element 0.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: W = DifferentialWeylAlgebra(R)
sage: W.zero()
0
```

```
>>> from sage.all import *
>>> R = QQ['x, y, z']; (x, y, z) = R._first_ngens(3)
>>> W = DifferentialWeylAlgebra(R)
>>> W.zero()
0
```

class sage.algebras.weyl_algebra.DifferentialWeylAlgebraAction(*G*)Bases: *Action*

Left action of a Weyl algebra on its underlying polynomial ring by differentiation.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: W = R.weyl_algebra()
sage: dx, dy = W.differentials()
sage: W.diff_action
Left action by Differential Weyl algebra of polynomials in x, y
over Rational Field on Multivariate Polynomial Ring in x, y over
Rational Field
```

```
>>> from sage.all import *
>>> R = QQ['x, y']; (x, y) = R._first_ngens(2)
>>> W = R.weyl_algebra()
>>> dx, dy = W.differentials()
>>> W.diff_action
Left action by Differential Weyl algebra of polynomials in x, y
over Rational Field on Multivariate Polynomial Ring in x, y over
Rational Field
```

```
sage: g = dx^2 + x*dy
sage: p = x^5 + x^3 + y^2*x^2 + 1
sage: W.diff_action(g, p)
2*x^3*y + 20*x^3 + 2*y^2 + 6*x
```

```
>>> from sage.all import *
>>> g = dx**Integer(2) + x*dy
>>> p = x**Integer(5) + x**Integer(3) + y**Integer(2)*x**Integer(2) + Integer(1)
>>> W.diff_action(g, p)
2*x^3*y + 20*x^3 + 2*y^2 + 6*x
```

The action is a left action:

```
sage: h = dx*x + x*y
sage: W.diff_action(h, W.diff_action(g, p)) == W.diff_action(h*g, p)
True
```

```
>>> from sage.all import *
>>> h = dx*x + x*y
>>> W.diff_action(h, W.diff_action(g, p)) == W.diff_action(h*g, p)
True
```

The action endomorphism of a differential operator:

```
sage: dg = W.diff_action(g); dg
Action of dx^2 + x*dy on Multivariate Polynomial Ring in x, y over
Rational Field under Left action by Differential Weyl algebra...
sage: dg(p) == W.diff_action(g, p) == g.diff(p)
True
```

```
>>> from sage.all import *
>>> dg = W.diff_action(g); dg
Action of dx^2 + x*dy on Multivariate Polynomial Ring in x, y over
Rational Field under Left action by Differential Weyl algebra...
>>> dg(p) == W.diff_action(g, p) == g.diff(p)
True
```

class sage.algebras.weyl_algebra.DifferentialWeylAlgebraElement

Bases: IndexedFreeModuleElement

An element in a differential Weyl algebra.

diff(p)

Apply this differential operator to a polynomial.

INPUT:

- p – polynomial of the underlying polynomial ring

OUTPUT:

The result of the left action of the Weyl algebra on the polynomial ring via differentiation.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: W = R.weyl_algebra()
sage: dx, dy = W.differentials()
sage: dx.diff(x^3)
3*x^2
sage: (dx*dy).diff(W(x^3*y^3))
9*x^2*y^2
sage: (x*dx + dy + 1).diff(x^4*y^4 + 1)
5*x^4*y^4 + 4*x^4*y^3 + 1
```

```
>>> from sage.all import *
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> W = R.weyl_algebra()
>>> dx, dy = W.differentials()
>>> dx.diff(x**Integer(3))
3*x^2
>>> (dx*dy).diff(W(x**Integer(3)*y**Integer(3)))
9*x^2*y^2
>>> (x*dx + dy + Integer(1)).diff(x**Integer(4)*y**Integer(4) + Integer(1))
5*x^4*y^4 + 4*x^4*y^3 + 1
```

factor_differentials()

Return a dict representing *self* with the differentials factored out.

EXAMPLES:

```
sage: R.<t> = QQ[]
sage: D = DifferentialWeylAlgebra(R)
sage: t, dt = D.gens()
sage: x = dt^3*t^3 + dt^2*t^4
sage: x
t^3*dt^3 + t^4*dt^2 + 9*t^2*dt^2 + 8*t^3*dt + 18*t*dt + 12*t^2 + 6
sage: x.factor_differentials()
{(0,): 12*t^2 + 6, (1,): 8*t^3 + 18*t, (2,): t^4 + 9*t^2, (3,): t^3}
sage: D.zero().factor_differentials()
{}

sage: R.<x,y,z> = QQ[]
sage: D = DifferentialWeylAlgebra(R)
sage: x, y, z, dx, dy, dz = D.gens()
sage: elt = dx^3*x^3 + (y^3-z*x)*dx^3 + dy^3*x^3 + dx*dy*dz*x*y*z
sage: elt
x^3*dy^3 + x*y*z*dx*dy*dz + y^3*dx^3 + x^3*dx^3 - x*z*dx^3 + y*z*dy*dz
+ x*z*dx*dz + x*y*dx*dy + 9*x^2*dx^2 + z*dz + y*dy + 19*x*dx + 7
sage: elt.factor_differentials()
{(0, 0, 0): 7,
```

(continues on next page)

(continued from previous page)

```

(0, 0, 1): z,
(0, 1, 0): y,
(0, 1, 1): y*z,
(0, 3, 0): x^3,
(1, 0, 0): 19*x,
(1, 0, 1): x*z,
(1, 1, 0): x*y,
(1, 1, 1): x*y*z,
(2, 0, 0): 9*x^2,
(3, 0, 0): x^3 + y^3 - x*z}

```

```

>>> from sage.all import *
>>> R = QQ['t']; (t,) = R._first_ngens(1)
>>> D = DifferentialWeylAlgebra(R)
>>> t, dt = D.gens()
>>> x = dt**Integer(3)*t**Integer(3) + dt**Integer(2)*t**Integer(4)
>>> x
t^3*dt^3 + t^4*dt^2 + 9*t^2*dt^2 + 8*t^3*dt + 18*t*dt + 12*t^2 + 6
>>> x.factor_differentials()
{(0,): 12*t^2 + 6, (1,): 8*t^3 + 18*t, (2,): t^4 + 9*t^2, (3,): t^3}
>>> D.zero().factor_differentials()
{}

>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> D = DifferentialWeylAlgebra(R)
>>> x, y, z, dx, dy, dz = D.gens()
>>> elt = dx**Integer(3)*x**Integer(3) + (y**Integer(3)-z*x)*dx**Integer(3) +
↳ dy**Integer(3)*x**Integer(3) + dx*dy*dz*x*y*z
>>> elt
x^3*dy^3 + x*y*z*dx*dy*dz + y^3*dx^3 + x^3*dx^3 - x*z*dx^3 + y*z*dy*dz
+ x*z*dx*dz + x*y*dx*dy + 9*x^2*dx^2 + z*dz + y*dy + 19*x*dx + 7
>>> elt.factor_differentials()
{(0, 0, 0): 7,
 (0, 0, 1): z,
 (0, 1, 0): y,
 (0, 1, 1): y*z,
 (0, 3, 0): x^3,
 (1, 0, 0): 19*x,
 (1, 0, 1): x*z,
 (1, 1, 0): x*y,
 (1, 1, 1): x*y*z,
 (2, 0, 0): 9*x^2,
 (3, 0, 0): x^3 + y^3 - x*z}

```

list()Return `self` as a list.

This list consists of pairs (m, c) , where m is a pair of tuples indexing a basis element of `self`, and c is the coordinate of `self` corresponding to this basis element. (Only nonzero coordinates are shown.)

EXAMPLES:

```

sage: W.<x,y,z> = DifferentialWeylAlgebra(QQ)
sage: dx,dy,dz = W.differentials()
sage: elt = dy - (3*x - z)*dx
sage: elt.list()
[(((0, 0, 0), (0, 1, 0)), 1),
 (((0, 0, 1), (1, 0, 0)), 1),
 (((1, 0, 0), (1, 0, 0)), -3)]

```

```

>>> from sage.all import *
>>> W = DifferentialWeylAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z) = W._
↳first_ngens(3)
>>> dx,dy,dz = W.differentials()
>>> elt = dy - (Integer(3)*x - z)*dx
>>> elt.list()
[(((0, 0, 0), (0, 1, 0)), 1),
 (((0, 0, 1), (1, 0, 0)), 1),
 (((1, 0, 0), (1, 0, 0)), -3)]

```

`sage.algebras.weyl_algebra.repr_factored(w, latex_output=False)`

Return a string representation of w with the dx_i generators factored on the right.

EXAMPLES:

```

sage: from sage.algebras.weyl_algebra import repr_factored
sage: R.<t> = QQ[]
sage: D = DifferentialWeylAlgebra(R)
sage: t, dt = D.gens()
sage: x = dt^3*t^3 + dt^2*t^4
sage: x
t^3*dt^3 + t^4*dt^2 + 9*t^2*dt^2 + 8*t^3*dt + 18*t*dt + 12*t^2 + 6
sage: print(repr_factored(x))
(12*t^2 + 6) + (8*t^3 + 18*t)*dt + (t^4 + 9*t^2)*dt^2 + (t^3)*dt^3
sage: repr_factored(x, True)
(12 t^{2} + 6) + (8 t^{3} + 18 t) \frac{\partial}{\partial t}
+ (t^4 + 9 t^2) \frac{\partial^2}{\partial t^2}
+ (t^3) \frac{\partial^3}{\partial t^3}
sage: repr_factored(D.zero())
'0'

```

```

>>> from sage.all import *
>>> from sage.algebras.weyl_algebra import repr_factored
>>> R = QQ['t']; (t,) = R._first_ngens(1)
>>> D = DifferentialWeylAlgebra(R)
>>> t, dt = D.gens()
>>> x = dt**Integer(3)*t**Integer(3) + dt**Integer(2)*t**Integer(4)
>>> x
t^3*dt^3 + t^4*dt^2 + 9*t^2*dt^2 + 8*t^3*dt + 18*t*dt + 12*t^2 + 6
>>> print(repr_factored(x))
(12*t^2 + 6) + (8*t^3 + 18*t)*dt + (t^4 + 9*t^2)*dt^2 + (t^3)*dt^3
>>> repr_factored(x, True)
(12 t^{2} + 6) + (8 t^{3} + 18 t) \frac{\partial}{\partial t}
+ (t^4 + 9 t^2) \frac{\partial^2}{\partial t^2}
+ (t^3) \frac{\partial^3}{\partial t^3}

```

(continues on next page)

(continued from previous page)

```
>>> repr_factored(D.zero())
'0'
```

With multiple variables:

```
sage: R.<x,y,z> = QQ[]
sage: D = DifferentialWeylAlgebra(R)
sage: x, y, z, dx, dy, dz = D.gens()
sage: elt = dx^3*x^3 + (y^3-z*x)*dx^3 + dy^3*x^3 + dx*dy*dz*x*y*z
sage: elt
x^3*dy^3 + x*y*z*dx*dy*dz + y^3*dx^3 + x^3*dx^3 - x*z*dx^3 + y*z*dy*dz
+ x*z*dx*dz + x*y*dx*dy + 9*x^2*dx^2 + z*dz + y*dy + 19*x*dx + 7
sage: print(repr_factored(elt))
(7) + (z)*dz + (y)*dy + (y*z)*dy*dz + (x^3)*dy^3 + (19*x)*dx
+ (x*z)*dx*dz + (x*y)*dx*dy + (x*y*z)*dx*dy*dz
+ (9*x^2)*dx^2 + (x^3 + y^3 - x*z)*dx^3
sage: repr_factored(D.zero(), True)
0
```

```
>>> from sage.all import *
>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> D = DifferentialWeylAlgebra(R)
>>> x, y, z, dx, dy, dz = D.gens()
>>> elt = dx**Integer(3)*x**Integer(3) + (y**Integer(3)-z*x)*dx**Integer(3) +
↳dy**Integer(3)*x**Integer(3) + dx*dy*dz*x*y*z
>>> elt
x^3*dy^3 + x*y*z*dx*dy*dz + y^3*dx^3 + x^3*dx^3 - x*z*dx^3 + y*z*dy*dz
+ x*z*dx*dz + x*y*dx*dy + 9*x^2*dx^2 + z*dz + y*dy + 19*x*dx + 7
>>> print(repr_factored(elt))
(7) + (z)*dz + (y)*dy + (y*z)*dy*dz + (x^3)*dy^3 + (19*x)*dx
+ (x*z)*dx*dz + (x*y)*dx*dy + (x*y*z)*dx*dy*dz
+ (9*x^2)*dx^2 + (x^3 + y^3 - x*z)*dx^3
>>> repr_factored(D.zero(), True)
0
```

`sage.algebras.weyl_algebra.repr_from_monomials` (*monomials*, *term_repr*, *use_latex=False*)

Return a string representation of an element of a free module from the dictionary `monomials`.

INPUT:

- `monomials` – list of pairs `[m, c]` where `m` is the index and `c` is the coefficient
- `term_repr` – a function which returns a string given an index (can be `repr` or `latex`, for example)
- `use_latex` – boolean (default: `False`); if `True` then the output is in latex format

EXAMPLES:

```
sage: from sage.algebras.weyl_algebra import repr_from_monomials
sage: R.<x,y,z> = QQ[]
sage: d = [(z, 4/7), (y, sqrt(2)), (x, -5)] #_
↳needs sage.symbolic
sage: repr_from_monomials(d, lambda m: repr(m)) #_
↳needs sage.symbolic
```

(continues on next page)

(continued from previous page)

```
'4/7*z + sqrt(2)*y - 5*x'
sage: a = repr_from_monomials(d, lambda m: latex(m), True); a #_
↳needs sage.symbolic
\frac{4}{7} z + \sqrt{2} y - 5 x
sage: type(a) #_
↳needs sage.symbolic
<class 'sage.misc.latex.LatexExpr'>
```

```
>>> from sage.all import *
>>> from sage.algebras.weyl_algebra import repr_from_monomials
>>> R = QQ['x, y, z']; (x, y, z) = R._first_ngens(3)
>>> d = [(z, Integer(4)/Integer(7)), (y, sqrt(Integer(2))), (x, -Integer(5))] #_
↳ # needs sage.symbolic
>>> repr_from_monomials(d, lambda m: repr(m)) #_
↳needs sage.symbolic
'4/7*z + sqrt(2)*y - 5*x'
>>> a = repr_from_monomials(d, lambda m: latex(m), True); a #_
↳needs sage.symbolic
\frac{4}{7} z + \sqrt{2} y - 5 x
>>> type(a) #_
↳needs sage.symbolic
<class 'sage.misc.latex.LatexExpr'>
```

The zero element:

```
sage: repr_from_monomials([], lambda m: repr(m))
'0'
sage: a = repr_from_monomials([], lambda m: latex(m), True); a
0
sage: type(a)
<class 'sage.misc.latex.LatexExpr'>
```

```
>>> from sage.all import *
>>> repr_from_monomials([], lambda m: repr(m))
'0'
>>> a = repr_from_monomials([], lambda m: latex(m), True); a
0
>>> type(a)
<class 'sage.misc.latex.LatexExpr'>
```

A “unity” element:

```
sage: repr_from_monomials([(1, 1)], lambda m: repr(m))
'1'
sage: a = repr_from_monomials([(1, 1)], lambda m: latex(m), True); a
1
sage: type(a)
<class 'sage.misc.latex.LatexExpr'>
```

```
>>> from sage.all import *
>>> repr_from_monomials([(Integer(1), Integer(1))], lambda m: repr(m))
'1'
```

(continues on next page)

(continued from previous page)

```
>>> a = repr_from_monomials([(Integer(1), Integer(1))], lambda m: latex(m), True);
↪ a
1
>>> type(a)
<class 'sage.misc.latex.LatexExpr'>
```

```
sage: repr_from_monomials([(1, -1)], lambda m: repr(m))
'-1'
sage: a = repr_from_monomials([(1, -1)], lambda m: latex(m), True); a
-1
sage: type(a)
<class 'sage.misc.latex.LatexExpr'>
```

```
>>> from sage.all import *
>>> repr_from_monomials([(Integer(1), -Integer(1))], lambda m: repr(m))
'-1'
>>> a = repr_from_monomials([(Integer(1), -Integer(1))], lambda m: latex(m), ↪
↪ True); a
-1
>>> type(a)
<class 'sage.misc.latex.LatexExpr'>
```

Leading minus signs are dealt with appropriately:

```
sage: # needs sage.symbolic
sage: d = [(z, -4/7), (y, -sqrt(2)), (x, -5)]
sage: repr_from_monomials(d, lambda m: repr(m))
'-4/7*z - sqrt(2)*y - 5*x'
sage: a = repr_from_monomials(d, lambda m: latex(m), True); a
-\frac{4}{7} z - \sqrt{2} y - 5 x
sage: type(a)
<class 'sage.misc.latex.LatexExpr'>
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> d = [(z, -Integer(4)/Integer(7)), (y, -sqrt(Integer(2))), (x, -Integer(5))]
>>> repr_from_monomials(d, lambda m: repr(m))
'-4/7*z - sqrt(2)*y - 5*x'
>>> a = repr_from_monomials(d, lambda m: latex(m), True); a
-\frac{4}{7} z - \sqrt{2} y - 5 x
>>> type(a)
<class 'sage.misc.latex.LatexExpr'>
```

Indirect doctests using a class that uses this function:

```
sage: R.<x,y> = QQ[]
sage: A = CliffordAlgebra(QuadraticForm(R, 3, [x,0,-1,3,-4,5]))
sage: a,b,c = A.gens()
sage: a*b*c
e0*e1*e2
sage: b*c
e1*e2
```

(continues on next page)

(continued from previous page)

```
sage: (a*a + 2)
x + 2
sage: c*(a*a + 2)*b
(-x - 2)*e1*e2 - 4*x - 8
sage: latex(c*(a*a + 2)*b)
\left( -x - 2 \right) e_{1} e_{2} - 4 x - 8
```

```
>>> from sage.all import *
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> A = CliffordAlgebra(QuadraticForm(R, Integer(3), [x,Integer(0),-Integer(1),
↳ Integer(3),-Integer(4),Integer(5)]))
>>> a,b,c = A.gens()
>>> a*b*c
e0*e1*e2
>>> b*c
e1*e2
>>> (a*a + Integer(2))
x + 2
>>> c*(a*a + Integer(2))*b
(-x - 2)*e1*e2 - 4*x - 8
>>> latex(c*(a*a + Integer(2))*b)
\left( -x - 2 \right) e_{1} e_{2} - 4 x - 8
```

5.29 Yangians

AUTHORS:

- Travis Scrimshaw (2013-10-08): Initial version

class sage.algebras.yangian.GradedYangianBase(A, category=None)

Bases: *AssociatedGradedAlgebra*

Base class for graded algebras associated to a Yangian.

class sage.algebras.yangian.GradedYangianLoop(Y)

Bases: *GradedYangianBase*

The associated graded algebra corresponding to a Yangian $\text{gr } Y(\mathfrak{gl}_n)$ with the filtration of $\deg t_{ij}^{(r)} = r - 1$.

Using this filtration for the Yangian, the associated graded algebra is isomorphic to $U(\mathfrak{gl}_n[z])$, the universal enveloping algebra of the loop algebra of $\mathfrak{gl}_n$.

INPUT:

- Y – a Yangian with the loop filtration

antipode_on_basis(m)

Return the antipode on a basis element indexed by m.

EXAMPLES:

```
sage: grY = Yangian(QQ, 4).graded_algebra()
sage: grY.antipode_on_basis(grY.gen(2,1,1).leading_support())
-tbar(2)[1,1]
```

(continues on next page)

(continued from previous page)

```

sage: x = grY.an_element(); x
tbar(1) [1,1]*tbar(1) [1,2]^2*tbar(1) [1,3]^3*tbar(42) [1,1]
sage: grY.antipode_on_basis(x.leading_support())
-tbar(1) [1,1]*tbar(1) [1,2]^2*tbar(1) [1,3]^3*tbar(42) [1,1]
- 2*tbar(1) [1,1]*tbar(1) [1,2]*tbar(1) [1,3]^3*tbar(42) [1,2]
- 3*tbar(1) [1,1]*tbar(1) [1,2]^2*tbar(1) [1,3]^2*tbar(42) [1,3]
+ 5*tbar(1) [1,2]^2*tbar(1) [1,3]^3*tbar(42) [1,1]
+ 10*tbar(1) [1,2]*tbar(1) [1,3]^3*tbar(42) [1,2]
+ 15*tbar(1) [1,2]^2*tbar(1) [1,3]^2*tbar(42) [1,3]

sage: g = grY.indices().gens()
sage: x = grY(g[1,1,1] * g[1,1,2]^2 * g[1,1,3]^3 * g[3,1,1]); x
tbar(1) [1,1]*tbar(1) [1,2]^2*tbar(1) [1,3]^3*tbar(3) [1,1]
sage: grY.antipode_on_basis(x.leading_support())
-tbar(1) [1,1]*tbar(1) [1,2]^2*tbar(1) [1,3]^3*tbar(3) [1,1]
- 2*tbar(1) [1,1]*tbar(1) [1,2]*tbar(1) [1,3]^3*tbar(3) [1,2]
- 3*tbar(1) [1,1]*tbar(1) [1,2]^2*tbar(1) [1,3]^2*tbar(3) [1,3]
+ 5*tbar(1) [1,2]^2*tbar(1) [1,3]^3*tbar(3) [1,1]
+ 10*tbar(1) [1,2]*tbar(1) [1,3]^3*tbar(3) [1,2]
+ 15*tbar(1) [1,2]^2*tbar(1) [1,3]^2*tbar(3) [1,3]

```

```

>>> from sage.all import *
>>> grY = Yangian(QQ, Integer(4)).graded_algebra()
>>> grY.antipode_on_basis(grY.gen(Integer(2), Integer(1), Integer(1)).leading_
↳support())
-tbar(2) [1,1]

>>> x = grY.an_element(); x
tbar(1) [1,1]*tbar(1) [1,2]^2*tbar(1) [1,3]^3*tbar(42) [1,1]
>>> grY.antipode_on_basis(x.leading_support())
-tbar(1) [1,1]*tbar(1) [1,2]^2*tbar(1) [1,3]^3*tbar(42) [1,1]
- 2*tbar(1) [1,1]*tbar(1) [1,2]*tbar(1) [1,3]^3*tbar(42) [1,2]
- 3*tbar(1) [1,1]*tbar(1) [1,2]^2*tbar(1) [1,3]^2*tbar(42) [1,3]
+ 5*tbar(1) [1,2]^2*tbar(1) [1,3]^3*tbar(42) [1,1]
+ 10*tbar(1) [1,2]*tbar(1) [1,3]^3*tbar(42) [1,2]
+ 15*tbar(1) [1,2]^2*tbar(1) [1,3]^2*tbar(42) [1,3]

>>> g = grY.indices().gens()
>>> x = grY(g[Integer(1), Integer(1), Integer(1)] * g[Integer(1), Integer(1),
↳Integer(2)]**Integer(2) * g[Integer(1), Integer(1), Integer(3)]**Integer(3) *
↳g[Integer(3), Integer(1), Integer(1)]); x
tbar(1) [1,1]*tbar(1) [1,2]^2*tbar(1) [1,3]^3*tbar(3) [1,1]
>>> grY.antipode_on_basis(x.leading_support())
-tbar(1) [1,1]*tbar(1) [1,2]^2*tbar(1) [1,3]^3*tbar(3) [1,1]
- 2*tbar(1) [1,1]*tbar(1) [1,2]*tbar(1) [1,3]^3*tbar(3) [1,2]
- 3*tbar(1) [1,1]*tbar(1) [1,2]^2*tbar(1) [1,3]^2*tbar(3) [1,3]
+ 5*tbar(1) [1,2]^2*tbar(1) [1,3]^3*tbar(3) [1,1]
+ 10*tbar(1) [1,2]*tbar(1) [1,3]^3*tbar(3) [1,2]
+ 15*tbar(1) [1,2]^2*tbar(1) [1,3]^2*tbar(3) [1,3]

```

coproduct_on_basis(m)

Return the coproduct on the basis element indexed by m.

EXAMPLES:

```
sage: grY = Yangian(QQ, 4).graded_algebra()
sage: grY.coproduct_on_basis(grY.gen(2, 1, 1).leading_support())
1 # tbar(2)[1, 1] + tbar(2)[1, 1] # 1
sage: grY.gen(2, 3, 1).coproduct()
1 # tbar(2)[3, 1] + tbar(2)[3, 1] # 1
```

```
>>> from sage.all import *
>>> grY = Yangian(QQ, Integer(4)).graded_algebra()
>>> grY.coproduct_on_basis(grY.gen(Integer(2), Integer(1), Integer(1)).leading_
↳support())
1 # tbar(2)[1, 1] + tbar(2)[1, 1] # 1
>>> grY.gen(Integer(2), Integer(3), Integer(1)).coproduct()
1 # tbar(2)[3, 1] + tbar(2)[3, 1] # 1
```

counit_on_basis(*m*)

Return the antipode on the basis element indexed by *m*.

EXAMPLES:

```
sage: grY = Yangian(QQ, 4).graded_algebra()
sage: grY.counit_on_basis(grY.gen(2, 3, 1).leading_support())
0
sage: grY.gen(0, 0, 0).counit()
1
```

```
>>> from sage.all import *
>>> grY = Yangian(QQ, Integer(4)).graded_algebra()
>>> grY.counit_on_basis(grY.gen(Integer(2), Integer(3), Integer(1)).leading_
↳support())
0
>>> grY.gen(Integer(0), Integer(0), Integer(0)).counit()
1
```

class sage.algebras.yangian.**GradedYangianNatural**(*Y*)

Bases: *GradedYangianBase*

The associated graded algebra corresponding to a Yangian $\text{gr } Y(\mathfrak{gl}_n)$ with the natural filtration of $\deg t_{ij}^{(r)} = r$.

INPUT:

- *Y* – a Yangian with the natural filtration

product_on_basis(*x*, *y*)

Return the product on basis elements given by the indices *x* and *y*.

EXAMPLES:

```
sage: grY = Yangian(QQ, 4, filtration='natural').graded_algebra()
sage: x = grY.gen(12, 2, 1) * grY.gen(2, 1, 1) # indirect doctest
sage: x
tbar(2)[1, 1]*tbar(12)[2, 1]
sage: x == grY.gen(2, 1, 1) * grY.gen(12, 2, 1)
True
```

```

>>> from sage.all import *
>>> grY = Yangian(QQ, Integer(4), filtration='natural').graded_algebra()
>>> x = grY.gen(Integer(12), Integer(2), Integer(1)) * grY.gen(Integer(2), Integer(1), Integer(1)) # indirect doctest
>>> x
tbar(2) [1, 1] * tbar(12) [2, 1]
>>> x == grY.gen(Integer(2), Integer(1), Integer(1)) * grY.gen(Integer(12), Integer(2), Integer(1))
True

```

class sage.algebras.yangian.**Yangian**(base_ring, n, variable_name, filtration)

Bases: `CombinatorialFreeModule`

The Yangian $Y(\mathfrak{gl}_n)$.

Let A be a commutative ring with unity. The *Yangian* $Y(\mathfrak{gl}_n)$, associated with the Lie algebra $\mathfrak{gl}_n$ for $n \geq 1$, is defined to be the unital associative algebra generated by $\{t_{ij}^{(r)} \mid 1 \leq i, j \leq n, r \geq 1\}$ subject to the relations

$$[t_{ij}^{(M+1)}, t_{k\ell}^{(L)}] - [t_{ij}^{(M)}, t_{k\ell}^{(L+1)}] = t_{kj}^{(M)} t_{i\ell}^{(L)} - t_{kj}^{(L)} t_{i\ell}^{(M)},$$

where $L, M \geq 0$ and $t_{ij}^{(0)} = \delta_{ij} \cdot 1$. This system of quadratic relations is equivalent to the system of commutation relations

$$[t_{ij}^{(r)}, t_{k\ell}^{(s)}] = \sum_{p=0}^{\min\{r,s\}-1} (t_{kj}^{(p)} t_{i\ell}^{(r+s-1-p)} - t_{kj}^{(r+s-1-p)} t_{i\ell}^{(p)}),$$

where $1 \leq i, j, k, \ell \leq n$ and $r, s \geq 1$.

Let u be a formal variable and, for $1 \leq i, j \leq n$, define

$$t_{ij}(u) = \delta_{ij} + \sum_{r=1}^{\infty} t_{ij}^{(r)} u^{-r} \in Y(\mathfrak{gl}_n)[[u^{-1}]].$$

Thus, we can write the defining relations as

$$(u - v)[t_{ij}(u), t_{k\ell}(v)] = t_{kj}(u)t_{i\ell}(v) - t_{kj}(v)t_{i\ell}(u).$$

These series can be combined into a single matrix:

$$T(u) := \sum_{i,j=1}^n t_{ij}(u) \otimes E_{ij} \in Y(\mathfrak{gl}_n)[[u^{-1}]] \otimes \text{End}(\mathbf{C}^n),$$

where E_{ij} is the matrix with a 1 in the (i, j) position and zeros elsewhere.

For $m \geq 2$, define formal variables $u_1, \dots, u_m$. For any $1 \leq k \leq m$, set

$$T_k(u_k) := \sum_{i,j=1}^n t_{ij}(u_k) \otimes (E_{ij})_k \in Y(\mathfrak{gl}_n)[[u_1^{-1}, \dots, u_m^{-1}]] \otimes \text{End}(\mathbf{C}^n)^{\otimes m},$$

where $(E_{ij})_k = 1^{\otimes(k-1)} \otimes E_{ij} \otimes 1^{\otimes(m-k)}$. If we consider $m = 2$, we can then also write the defining relations as

$$R(u - v)T_1(u)T_2(v) = T_2(v)T_1(u)R(u - v),$$

where $R(u) = 1 - Pu^{-1}$ and P is the permutation operator that swaps the two factors. Moreover, we can write the Hopf algebra structure as

$$\Delta: T(u) \mapsto T_{[1]}(u)T_{[2]}(u), \quad S: T(u) \mapsto T^{-1}(u), \quad \epsilon: T(u) \mapsto 1,$$

where $T_{[a]} = \sum_{i,j=1}^n (1^{\otimes a-1} \otimes t_{ij}(u) \otimes 1^{2-a}) \otimes (E_{ij})_1$.

We can also impose two filtrations on $Y(\mathfrak{gl}_n)$: the *natural* filtration $\deg t_{ij}^{(r)} = r$ and the *loop* filtration $\deg t_{ij}^{(r)} = r - 1$. The natural filtration has a graded homomorphism with $U(\mathfrak{gl}_n)$ by $t_{ij}^{(r)} \mapsto (E^r)_{ij}$ and an associated graded algebra being polynomial algebra. Moreover, this shows a PBW theorem for the Yangian, that for any fixed order, we can write elements as unique linear combinations of ordered monomials using $t_{ij}^{(r)}$. For the loop filtration, the associated graded algebra is isomorphic (as Hopf algebras) to $U(\mathfrak{gl}_n[z])$ given by $\bar{t}_{ij}^{(r)} \mapsto E_{ij}x^{r-1}$, where $\bar{t}_{ij}^{(r)}$ is the image of $t_{ij}^{(r)}$ in the $(r - 1)$ -th component of $\text{gr } Y(\mathfrak{gl}_n)$.

INPUT:

- `base_ring` – the base ring
- `n` – the size n
- `level` – (optional) the level of the Yangian
- `variable_name` – (default: 't') the name of the variable
- `filtration` – (default: 'loop') the filtration and can be one of the following:
 - 'natural' – the filtration is given by $\deg t_{ij}^{(r)} = r$
 - 'loop' – the filtration is given by $\deg t_{ij}^{(r)} = r - 1$

Todo

Implement the antipode.

EXAMPLES:

```
sage: Y = Yangian(QQ, 4)
sage: t = Y.algebra_generators()
sage: t[6,2,1] * t[2,3,2]
-t(1)[2,2]*t(6)[3,1] + t(1)[3,1]*t(6)[2,2]
+ t(2)[3,2]*t(6)[2,1] - t(7)[3,1]
sage: t[6,2,1] * t[3,1,4]
t(1)[1,1]*t(7)[2,4] + t(1)[1,4]*t(6)[2,1] - t(1)[2,1]*t(6)[1,4]
- t(1)[2,4]*t(7)[1,1] + t(2)[1,1]*t(6)[2,4] - t(2)[2,4]*t(6)[1,1]
+ t(3)[1,4]*t(6)[2,1] + t(6)[2,4] + t(8)[2,4]
```

```
>>> from sage.all import *
>>> Y = Yangian(QQ, Integer(4))
>>> t = Y.algebra_generators()
>>> t[Integer(6),Integer(2),Integer(1)] * t[Integer(2),Integer(3),Integer(2)]
-t(1)[2,2]*t(6)[3,1] + t(1)[3,1]*t(6)[2,2]
+ t(2)[3,2]*t(6)[2,1] - t(7)[3,1]
>>> t[Integer(6),Integer(2),Integer(1)] * t[Integer(3),Integer(1),Integer(4)]
t(1)[1,1]*t(7)[2,4] + t(1)[1,4]*t(6)[2,1] - t(1)[2,1]*t(6)[1,4]
- t(1)[2,4]*t(7)[1,1] + t(2)[1,1]*t(6)[2,4] - t(2)[2,4]*t(6)[1,1]
+ t(3)[1,4]*t(6)[2,1] + t(6)[2,4] + t(8)[2,4]
```

We check that the natural filtration has a homomorphism to $U(\mathfrak{gl}_n)$ as algebras:

```
sage: Y = Yangian(QQ, 4, filtration='natural')
sage: t = Y.algebra_generators()
sage: gl4 = lie_algebras.gl(QQ, 4)
sage: Ugl4 = gl4.pbw_basis()
sage: E = matrix(Ugl4, 4, 4, Ugl4.gens())
sage: Esq = E^2
sage: t[2,1,3] * t[1,2,1]
t(1)[2,1]*t(2)[1,3] - t(2)[2,3]
sage: Esq[0,2] * E[1,0] == E[1,0] * Esq[0,2] - Esq[1,2]
True

sage: Em = [E^k for k in range(1,5)]
sage: S = list(t.some_elements())[:30:3]
sage: def convert(x):
....:     return sum(c * prod(Em[t[0]-1][t[1]-1,t[2]-1] ** e
....:                        for t,e in m._sorted_items())
....:                for m,c in x)
sage: for x in S:
....:     for y in S:
....:         ret = x * y
....:         rhs = convert(x) * convert(y)
....:         assert rhs == convert(ret)
....:         assert ret.maximal_degree() == rhs.maximal_degree()
```

```
>>> from sage.all import *
>>> Y = Yangian(QQ, Integer(4), filtration='natural')
>>> t = Y.algebra_generators()
>>> gl4 = lie_algebras.gl(QQ, Integer(4))
>>> Ugl4 = gl4.pbw_basis()
>>> E = matrix(Ugl4, Integer(4), Integer(4), Ugl4.gens())
>>> Esq = E**Integer(2)
>>> t[Integer(2),Integer(1),Integer(3)] * t[Integer(1),Integer(2),Integer(1)]
t(1)[2,1]*t(2)[1,3] - t(2)[2,3]
>>> Esq[Integer(0),Integer(2)] * E[Integer(1),Integer(0)] == E[Integer(1),
↳Integer(0)] * Esq[Integer(0),Integer(2)] - Esq[Integer(1),Integer(2)]
True

>>> Em = [E**k for k in range(Integer(1),Integer(5))]
>>> S = list(t.some_elements())[:Integer(30):Integer(3)]
>>> def convert(x):
...     return sum(c * prod(Em[t[Integer(0)]-Integer(1)][t[Integer(1)]-Integer(1),
↳t[Integer(2)]-Integer(1)] ** e
...                     for t,e in m._sorted_items())
...                 for m,c in x)
>>> for x in S:
...     for y in S:
...         ret = x * y
...         rhs = convert(x) * convert(y)
...         assert rhs == convert(ret)
...         assert ret.maximal_degree() == rhs.maximal_degree()
```

REFERENCES:

- [Wikipedia article Yangian](#)
- [MNO1994]
- [Mol2007]

algebra_generators()

Return the algebra generators of self.

EXAMPLES:

```
sage: Y = Yangian(QQ, 4)
sage: Y.algebra_generators()
Lazy family (generator(i))_{i in The Cartesian product of
(Positive integers, {1, 2, 3, 4}, {1, 2, 3, 4})}
```

```
>>> from sage.all import *
>>> Y = Yangian(QQ, Integer(4))
>>> Y.algebra_generators()
Lazy family (generator(i))_{i in The Cartesian product of
(Positive integers, {1, 2, 3, 4}, {1, 2, 3, 4})}
```

coproduct_on_basis(m)

Return the coproduct on the basis element indexed by m.

The coproduct $\Delta: Y(\mathfrak{gl}_n) \longrightarrow Y(\mathfrak{gl}_n) \otimes Y(\mathfrak{gl}_n)$ is defined by

$$\Delta(t_{ij}(u)) = \sum_{a=1}^n t_{ia}(u) \otimes t_{aj}(u).$$

EXAMPLES:

```
sage: Y = Yangian(QQ, 4)
sage: Y.gen(2,1,1).coproduct() # indirect doctest
1 # t(2) [1,1] + t(1) [1,1] # t(1) [1,1] + t(1) [1,2] # t(1) [2,1]
+ t(1) [1,3] # t(1) [3,1] + t(1) [1,4] # t(1) [4,1] + t(2) [1,1] # 1
sage: Y.gen(2,3,1).coproduct()
1 # t(2) [3,1] + t(1) [3,1] # t(1) [1,1] + t(1) [3,2] # t(1) [2,1]
+ t(1) [3,3] # t(1) [3,1] + t(1) [3,4] # t(1) [4,1] + t(2) [3,1] # 1
sage: Y.gen(2,2,3).coproduct()
1 # t(2) [2,3] + t(1) [2,1] # t(1) [1,3] + t(1) [2,2] # t(1) [2,3]
+ t(1) [2,3] # t(1) [3,3] + t(1) [2,4] # t(1) [4,3] + t(2) [2,3] # 1
```

```
>>> from sage.all import *
>>> Y = Yangian(QQ, Integer(4))
>>> Y.gen(Integer(2), Integer(1), Integer(1)).coproduct() # indirect doctest
1 # t(2) [1,1] + t(1) [1,1] # t(1) [1,1] + t(1) [1,2] # t(1) [2,1]
+ t(1) [1,3] # t(1) [3,1] + t(1) [1,4] # t(1) [4,1] + t(2) [1,1] # 1
>>> Y.gen(Integer(2), Integer(3), Integer(1)).coproduct()
1 # t(2) [3,1] + t(1) [3,1] # t(1) [1,1] + t(1) [3,2] # t(1) [2,1]
+ t(1) [3,3] # t(1) [3,1] + t(1) [3,4] # t(1) [4,1] + t(2) [3,1] # 1
>>> Y.gen(Integer(2), Integer(2), Integer(3)).coproduct()
1 # t(2) [2,3] + t(1) [2,1] # t(1) [1,3] + t(1) [2,2] # t(1) [2,3]
+ t(1) [2,3] # t(1) [3,3] + t(1) [2,4] # t(1) [4,3] + t(2) [2,3] # 1
```

counit_on_basis(*m*)

Return the counit on the basis element indexed by *m*.

EXAMPLES:

```
sage: Y = Yangian(QQ, 4)
sage: Y.gen(2,3,1).counit() # indirect doctest
0
sage: Y.gen(0,0,0).counit()
1
```

```
>>> from sage.all import *
>>> Y = Yangian(QQ, Integer(4))
>>> Y.gen(Integer(2), Integer(3), Integer(1)).counit() # indirect doctest
0
>>> Y.gen(Integer(0), Integer(0), Integer(0)).counit()
1
```

degree_on_basis(*m*)

Return the degree of the monomial index by *m*.

The degree of $t_{ij}^{(r)}$ is equal to $r - 1$ if *filtration* = 'loop' and is equal to r if *filtration* = 'natural'.

EXAMPLES:

```
sage: Y = Yangian(QQ, 4)
sage: Y.degree_on_basis(Y.gen(2,1,1).leading_support())
1
sage: x = Y.gen(5,2,3)^4
sage: Y.degree_on_basis(x.leading_support())
16
sage: elt = Y.gen(10,3,1) * Y.gen(2,1,1) * Y.gen(1,2,4); elt
t(1) [1, 1]*t(1) [2, 4]*t(10) [3, 1] - t(1) [2, 4]*t(1) [3, 1]*t(10) [1, 1]
+ t(1) [2, 4]*t(2) [1, 1]*t(10) [3, 1] + t(1) [2, 4]*t(10) [3, 1]
+ t(1) [2, 4]*t(11) [3, 1]
sage: for s in sorted(elt.support(), key=str): s, Y.degree_on_basis(s)
(t(1, 1, 1)*t(1, 2, 4)*t(10, 3, 1), 9)
(t(1, 2, 4)*t(1, 3, 1)*t(10, 1, 1), 9)
(t(1, 2, 4)*t(10, 3, 1), 9)
(t(1, 2, 4)*t(11, 3, 1), 10)
(t(1, 2, 4)*t(2, 1, 1)*t(10, 3, 1), 10)

sage: Y = Yangian(QQ, 4, filtration='natural')
sage: Y.degree_on_basis(Y.gen(2,1,1).leading_support())
2
sage: x = Y.gen(5,2,3)^4
sage: Y.degree_on_basis(x.leading_support())
20
sage: elt = Y.gen(10,3,1) * Y.gen(2,1,1) * Y.gen(1,2,4)
sage: for s in sorted(elt.support(), key=str): s, Y.degree_on_basis(s)
(t(1, 1, 1)*t(1, 2, 4)*t(10, 3, 1), 12)
(t(1, 2, 4)*t(1, 3, 1)*t(10, 1, 1), 12)
(t(1, 2, 4)*t(10, 3, 1), 11)
```

(continues on next page)

(continued from previous page)

```
(t(1, 2, 4)*t(11, 3, 1), 12)
(t(1, 2, 4)*t(2, 1, 1)*t(10, 3, 1), 13)
```

```
>>> from sage.all import *
>>> Y = Yangian(QQ, Integer(4))
>>> Y.degree_on_basis(Y.gen(Integer(2), Integer(1), Integer(1)).leading_
↳support())
1
>>> x = Y.gen(Integer(5), Integer(2), Integer(3))**Integer(4)
>>> Y.degree_on_basis(x.leading_support())
16
>>> elt = Y.gen(Integer(10), Integer(3), Integer(1)) * Y.gen(Integer(2),
↳Integer(1), Integer(1)) * Y.gen(Integer(1), Integer(2), Integer(4)); elt
t(1) [1, 1]*t(1) [2, 4]*t(10) [3, 1] - t(1) [2, 4]*t(1) [3, 1]*t(10) [1, 1]
+ t(1) [2, 4]*t(2) [1, 1]*t(10) [3, 1] + t(1) [2, 4]*t(10) [3, 1]
+ t(1) [2, 4]*t(11) [3, 1]
>>> for s in sorted(elt.support(), key=str): s, Y.degree_on_basis(s)
(t(1, 1, 1)*t(1, 2, 4)*t(10, 3, 1), 9)
(t(1, 2, 4)*t(1, 3, 1)*t(10, 1, 1), 9)
(t(1, 2, 4)*t(10, 3, 1), 9)
(t(1, 2, 4)*t(11, 3, 1), 10)
(t(1, 2, 4)*t(2, 1, 1)*t(10, 3, 1), 10)

>>> Y = Yangian(QQ, Integer(4), filtration='natural')
>>> Y.degree_on_basis(Y.gen(Integer(2), Integer(1), Integer(1)).leading_
↳support())
2
>>> x = Y.gen(Integer(5), Integer(2), Integer(3))**Integer(4)
>>> Y.degree_on_basis(x.leading_support())
20
>>> elt = Y.gen(Integer(10), Integer(3), Integer(1)) * Y.gen(Integer(2),
↳Integer(1), Integer(1)) * Y.gen(Integer(1), Integer(2), Integer(4))
>>> for s in sorted(elt.support(), key=str): s, Y.degree_on_basis(s)
(t(1, 1, 1)*t(1, 2, 4)*t(10, 3, 1), 12)
(t(1, 2, 4)*t(1, 3, 1)*t(10, 1, 1), 12)
(t(1, 2, 4)*t(10, 3, 1), 11)
(t(1, 2, 4)*t(11, 3, 1), 12)
(t(1, 2, 4)*t(2, 1, 1)*t(10, 3, 1), 13)
```

dimension()Return the dimension of self, which is ∞ .

EXAMPLES:

```
sage: Y = Yangian(QQ, 4)
sage: Y.dimension()
+Infinity
```

```
>>> from sage.all import *
>>> Y = Yangian(QQ, Integer(4))
>>> Y.dimension()
+Infinity
```

gen(*r, i=None, j=None*)

Return the generator $t_{ij}^{(r)}$ of self.

EXAMPLES:

```
sage: Y = Yangian(QQ, 4)
sage: Y.gen(2, 1, 3)
t(2) [1, 3]
sage: Y.gen(12, 2, 1)
t(12) [2, 1]
sage: Y.gen(0, 1, 1)
1
sage: Y.gen(0, 1, 3)
0
```

```
>>> from sage.all import *
>>> Y = Yangian(QQ, Integer(4))
>>> Y.gen(Integer(2), Integer(1), Integer(3))
t(2) [1, 3]
>>> Y.gen(Integer(12), Integer(2), Integer(1))
t(12) [2, 1]
>>> Y.gen(Integer(0), Integer(1), Integer(1))
1
>>> Y.gen(Integer(0), Integer(1), Integer(3))
0
```

graded_algebra()

Return the associated graded algebra of self.

EXAMPLES:

```
sage: Yangian(QQ, 4).graded_algebra()
Graded Algebra of Yangian of gl(4) in the loop filtration over Rational Field
sage: Yangian(QQ, 4, filtration='natural').graded_algebra()
Graded Algebra of Yangian of gl(4) in the natural filtration over Rational_
↪Field
```

```
>>> from sage.all import *
>>> Yangian(QQ, Integer(4)).graded_algebra()
Graded Algebra of Yangian of gl(4) in the loop filtration over Rational Field
>>> Yangian(QQ, Integer(4), filtration='natural').graded_algebra()
Graded Algebra of Yangian of gl(4) in the natural filtration over Rational_
↪Field
```

one_basis()

Return the basis index of the element 1.

EXAMPLES:

```
sage: Y = Yangian(QQ, 4)
sage: Y.one_basis()
1
```



```
>>> from sage.all import *
>>> Y = Yangian(QQ, Integer(4))
>>> Y.one_basis()
1
```

product_on_basis(x, y)

Return the product of two monomials given by x and y .

EXAMPLES:

```
sage: Y = Yangian(QQ, 4)
sage: Y.gen(12, 2, 1) * Y.gen(2, 1, 1) # indirect doctest
t(1) [1, 1] * t(12) [2, 1] - t(1) [2, 1] * t(12) [1, 1]
+ t(2) [1, 1] * t(12) [2, 1] + t(12) [2, 1] + t(13) [2, 1]
```

```
>>> from sage.all import *
>>> Y = Yangian(QQ, Integer(4))
>>> Y.gen(Integer(12), Integer(2), Integer(1)) * Y.gen(Integer(2), Integer(1),
↪ Integer(1)) # indirect doctest
t(1) [1, 1] * t(12) [2, 1] - t(1) [2, 1] * t(12) [1, 1]
+ t(2) [1, 1] * t(12) [2, 1] + t(12) [2, 1] + t(13) [2, 1]
```

product_on_gens(a, b)

Return the product on two generators indexed by a and b .

We assume $(r, i, j) \geq (s, k, \ell)$, and we start with the basic relation:

$$[t_{ij}^{(r)}, t_{k\ell}^{(s)}] - [t_{ij}^{(r-1)}, t_{k\ell}^{(s+1)}] = t_{kj}^{(r-1)} t_{i\ell}^{(s)} - t_{kj}^{(s)} t_{i\ell}^{(r-1)}.$$

Solving for the first term and using induction we get:

$$[t_{ij}^{(r)}, t_{k\ell}^{(s)}] = \sum_{a=1}^s \left(t_{kj}^{(a-1)} t_{i\ell}^{(r+s-a)} - t_{kj}^{(r+s-a)} t_{i\ell}^{(a-1)} \right).$$

Next applying induction on this we get

$$t_{ij}^{(r)} t_{k\ell}^{(s)} = t_{k\ell}^{(s)} t_{ij}^{(r)} + \sum C_{abcd}^{m\ell} t_{ab}^{(m)} t_{cd}^{(\ell)}$$

where $m + \ell < r + s$ and $t_{ab}^{(m)} < t_{cd}^{(\ell)}$.

EXAMPLES:

```
sage: Y = Yangian(QQ, 4)
sage: Y.product_on_gens((2, 1, 1), (12, 2, 1))
t(2) [1, 1] * t(12) [2, 1]
sage: Y.gen(2, 1, 1) * Y.gen(12, 2, 1)
t(2) [1, 1] * t(12) [2, 1]
sage: Y.product_on_gens((12, 2, 1), (2, 1, 1))
t(1) [1, 1] * t(12) [2, 1] - t(1) [2, 1] * t(12) [1, 1]
+ t(2) [1, 1] * t(12) [2, 1] + t(12) [2, 1] + t(13) [2, 1]
sage: Y.gen(12, 2, 1) * Y.gen(2, 1, 1)
t(1) [1, 1] * t(12) [2, 1] - t(1) [2, 1] * t(12) [1, 1]
+ t(2) [1, 1] * t(12) [2, 1] + t(12) [2, 1] + t(13) [2, 1]
```

```

>>> from sage.all import *
>>> Y = Yangian(QQ, Integer(4))
>>> Y.product_on_gens((Integer(2), Integer(1), Integer(1)), (Integer(12),
↪ Integer(2), Integer(1)))
t(2) [1, 1]*t(12) [2, 1]
>>> Y.gen(Integer(2), Integer(1), Integer(1)) * Y.gen(Integer(12), Integer(2),
↪ Integer(1))
t(2) [1, 1]*t(12) [2, 1]
>>> Y.product_on_gens((Integer(12), Integer(2), Integer(1)), (Integer(2),
↪ Integer(1), Integer(1)))
t(1) [1, 1]*t(12) [2, 1] - t(1) [2, 1]*t(12) [1, 1]
+ t(2) [1, 1]*t(12) [2, 1] + t(12) [2, 1] + t(13) [2, 1]
>>> Y.gen(Integer(12), Integer(2), Integer(1)) * Y.gen(Integer(2), Integer(1),
↪ Integer(1))
t(1) [1, 1]*t(12) [2, 1] - t(1) [2, 1]*t(12) [1, 1]
+ t(2) [1, 1]*t(12) [2, 1] + t(12) [2, 1] + t(13) [2, 1]

```

class sage.algebras.yangian.**YangianLevel** (*base_ring, n, level, variable_name, filtration*)

Bases: *Yangian*

The Yangian $Y_\ell(\mathfrak{gl}_n)$ of level ℓ .

The Yangian of level ℓ is the quotient of the Yangian $Y(\mathfrak{gl}_n)$ by the two-sided ideal generated by $t_{ij}^{(r)}$ for all $r > p$ and all $i, j \in \{1, \dots, n\}$.

EXAMPLES:

```

sage: Y = Yangian(QQ, 4, 3)
sage: elt = Y.gen(3, 2, 1) * Y.gen(1, 1, 3)
sage: elt * Y.gen(1, 1, 2)
t(1) [1, 2]*t(1) [1, 3]*t(3) [2, 1] + t(1) [1, 2]*t(3) [2, 3]
- t(1) [1, 3]*t(3) [1, 1] + t(1) [1, 3]*t(3) [2, 2] - t(3) [1, 3]

```

```

>>> from sage.all import *
>>> Y = Yangian(QQ, Integer(4), Integer(3))
>>> elt = Y.gen(Integer(3), Integer(2), Integer(1)) * Y.gen(Integer(1), Integer(1),
↪ Integer(3))
>>> elt * Y.gen(Integer(1), Integer(1), Integer(2))
t(1) [1, 2]*t(1) [1, 3]*t(3) [2, 1] + t(1) [1, 2]*t(3) [2, 3]
- t(1) [1, 3]*t(3) [1, 1] + t(1) [1, 3]*t(3) [2, 2] - t(3) [1, 3]

```

defining_polynomial (*i, j, u=None*)

Return the defining polynomial of *i* and *j*.

The defining polynomial is given by:

$$T_{ij}(u) = \delta_{ij}u^\ell + \sum_{k=1}^{\ell} t_{ij}^{(k)}u^{\ell-k}.$$

EXAMPLES:

```

sage: Y = Yangian(QQ, 3, 5)
sage: Y.defining_polynomial(3, 2)
t(1) [3, 2]*u^4 + t(2) [3, 2]*u^3 + t(3) [3, 2]*u^2 + t(4) [3, 2]*u + t(5) [3, 2]

```

(continues on next page)

(continued from previous page)

```
sage: Y.defined_polynomial(1, 1)
u^5 + t(1) [1,1]*u^4 + t(2) [1,1]*u^3 + t(3) [1,1]*u^2 + t(4) [1,1]*u + t(5) [1,1]
```

```
>>> from sage.all import *
>>> Y = Yangian(QQ, Integer(3), Integer(5))
>>> Y.defined_polynomial(Integer(3), Integer(2))
t(1) [3,2]*u^4 + t(2) [3,2]*u^3 + t(3) [3,2]*u^2 + t(4) [3,2]*u + t(5) [3,2]
>>> Y.defined_polynomial(Integer(1), Integer(1))
u^5 + t(1) [1,1]*u^4 + t(2) [1,1]*u^3 + t(3) [1,1]*u^2 + t(4) [1,1]*u + t(5) [1,1]
```

gen(*r*, *i=None*, *j=None*)

Return the generator $t_{ij}^{(r)}$ of self.

EXAMPLES:

```
sage: Y = Yangian(QQ, 4, 3)
sage: Y.gen(2, 1, 3)
t(2) [1,3]
sage: Y.gen(12, 2, 1)
0
sage: Y.gen(0, 1, 1)
1
sage: Y.gen(0, 1, 3)
0
```

```
>>> from sage.all import *
>>> Y = Yangian(QQ, Integer(4), Integer(3))
>>> Y.gen(Integer(2), Integer(1), Integer(3))
t(2) [1,3]
>>> Y.gen(Integer(12), Integer(2), Integer(1))
0
>>> Y.gen(Integer(0), Integer(1), Integer(1))
1
>>> Y.gen(Integer(0), Integer(1), Integer(3))
0
```

gens()

Return the generators of self.

EXAMPLES:

```
sage: Y = Yangian(QQ, 2, 2)
sage: Y.gens()
(t(1) [1,1], t(2) [1,1], t(1) [1,2], t(2) [1,2], t(1) [2,1],
 t(2) [2,1], t(1) [2,2], t(2) [2,2])
```

```
>>> from sage.all import *
>>> Y = Yangian(QQ, Integer(2), Integer(2))
>>> Y.gens()
(t(1) [1,1], t(2) [1,1], t(1) [1,2], t(2) [1,2], t(1) [2,1],
 t(2) [2,1], t(1) [2,2], t(2) [2,2])
```

level()

Return the level of self.

EXAMPLES:

```
sage: Y = Yangian(QQ, 3, 5)
sage: Y.level()
5
```

```
>>> from sage.all import *
>>> Y = Yangian(QQ, Integer(3), Integer(5))
>>> Y.level()
5
```

product_on_gens(a, b)

Return the product on two generators indexed by a and b.

See also*Yangian.product_on_gens()*

EXAMPLES:

```
sage: Y = Yangian(QQ, 4, 3)
sage: Y.gen(1,2,2) * Y.gen(2,1,3) # indirect doctest
t(1) [2,2]*t(2) [1,3]
sage: Y.gen(1,2,1) * Y.gen(2,1,3) # indirect doctest
t(1) [2,1]*t(2) [1,3]
sage: Y.gen(3,2,1) * Y.gen(1,1,3) # indirect doctest
t(1) [1,3]*t(3) [2,1] + t(3) [2,3]
```

```
>>> from sage.all import *
>>> Y = Yangian(QQ, Integer(4), Integer(3))
>>> Y.gen(Integer(1),Integer(2),Integer(2)) * Y.gen(Integer(2),Integer(1),
↪Integer(3)) # indirect doctest
t(1) [2,2]*t(2) [1,3]
>>> Y.gen(Integer(1),Integer(2),Integer(1)) * Y.gen(Integer(2),Integer(1),
↪Integer(3)) # indirect doctest
t(1) [2,1]*t(2) [1,3]
>>> Y.gen(Integer(3),Integer(2),Integer(1)) * Y.gen(Integer(1),Integer(1),
↪Integer(3)) # indirect doctest
t(1) [1,3]*t(3) [2,1] + t(3) [2,3]
```

quantum_determinant(u=None)

Return the quantum determinant of self.

The quantum determinant is defined by:

$$\text{qdet}(u) = \sum_{\sigma \in S_n} (-1)^\sigma \prod_{k=1}^n T_{\sigma(k),k}(u - k + 1).$$

EXAMPLES:

```

sage: Y = Yangian(QQ, 2, 2)
sage: Y.quantum_determinant()
u^4 + (-2 + t(1)[1,1] + t(1)[2,2])*u^3
+ (1 - t(1)[1,1] + t(1)[1,1]*t(1)[2,2] - t(1)[1,2]*t(1)[2,1]
- 2*t(1)[2,2] + t(2)[1,1] + t(2)[2,2])*u^2
+ (-t(1)[1,1]*t(1)[2,2] + t(1)[1,1]*t(2)[2,2]
+ t(1)[1,2]*t(1)[2,1] - t(1)[1,2]*t(2)[2,1]
- t(1)[2,1]*t(2)[1,2] + t(1)[2,2] + t(1)[2,2]*t(2)[1,1]
- t(2)[1,1] - t(2)[2,2])*u
- t(1)[1,1]*t(2)[2,2] + t(1)[1,2]*t(2)[2,1] + t(2)[1,1]*t(2)[2,2]
- t(2)[1,2]*t(2)[2,1] + t(2)[2,2]

```

```

>>> from sage.all import *
>>> Y = Yangian(QQ, Integer(2), Integer(2))
>>> Y.quantum_determinant()
u^4 + (-2 + t(1)[1,1] + t(1)[2,2])*u^3
+ (1 - t(1)[1,1] + t(1)[1,1]*t(1)[2,2] - t(1)[1,2]*t(1)[2,1]
- 2*t(1)[2,2] + t(2)[1,1] + t(2)[2,2])*u^2
+ (-t(1)[1,1]*t(1)[2,2] + t(1)[1,1]*t(2)[2,2]
+ t(1)[1,2]*t(1)[2,1] - t(1)[1,2]*t(2)[2,1]
- t(1)[2,1]*t(2)[1,2] + t(1)[2,2] + t(1)[2,2]*t(2)[1,1]
- t(2)[1,1] - t(2)[2,2])*u
- t(1)[1,1]*t(2)[2,2] + t(1)[1,2]*t(2)[2,1] + t(2)[1,1]*t(2)[2,2]
- t(2)[1,2]*t(2)[2,1] + t(2)[2,2]

```


HECKE ALGEBRAS

6.1 Ariki-Koike Algebras

The *Ariki-Koike algebras* were introduced by Ariki and Koike [AK1994] as a natural generalization of the Iwahori-Hecke algebras of types A and B (see [IwahoriHeckeAlgebra](#)). Soon afterwards, Broué and Malle defined analogues of the Hecke algebras for all complex reflection groups

Fix nonnegative integers r and n . The Ariki-Koike algebras are deformations of the group algebra of the complex reflection group $G(r, 1, n) = \mathbf{Z}/r\mathbf{Z} \wr \mathfrak{S}_n$. If R is a ring containing a *Hecke parameter* q and *cyclotomic parameters* $u_0, \dots, u_{r-1}$ then the Ariki-Koike algebra $H_n(q, u_1, \dots, u_r)$ is the unital associative r -algebra with generators $T_0, T_1, \dots, T_{n-1}$ and relations:

$$\begin{aligned} \prod_{i=0}^{r-1} (T_0 - u_i) &= 0, \\ T_i^2 &= (q-1)T_i + q \quad \text{for } 1 \leq i < n, \\ T_0 T_1 T_0 T_1 &= T_1 T_0 T_1 T_0, \\ T_i T_j &= T_j T_i \quad \text{if } |i-j| \geq 2, \\ T_i T_{i+1} T_i &= T_{i+1} T_i T_{i+1} \quad \text{for } 1 \leq i < n. \end{aligned}$$

AUTHORS:

- Travis Scrimshaw (2016-04): initial version
- Andrew Mathas (2016-07): improved multiplication code

REFERENCES:

- [AK1994]
- [BM1993]
- [MM1998]

class `sage.algebras.hecke_algebras.ariki_koike_algebra.ArikiKoikeAlgebra` (r, n, q, u, R)

Bases: `Parent, UniqueRepresentation`

The Ariki-Koike algebra $H_{r,n}(q, u)$.

Let R be an unital integral domain. Let $q, u_0, \dots, u_{r-1} \in R$ such that $q^{-1} \in R$. The *Ariki-Koike algebra* is the

unital associative algebra $H_{r,n}(q, u)$ generated by $T_0, \dots, T_{n-1}$ that satisfies the following relations:

$$\begin{aligned} \prod_{i=0}^{r-1} (T_0 - u_i) &= 0, \\ T_i^2 &= (q-1)T_i + q \quad \text{for } 1 \leq i < n, \\ T_0 T_1 T_0 T_1 &= T_1 T_0 T_1 T_0, \\ T_i T_j &= T_j T_i \quad \text{if } |i-j| \geq 2, \\ T_i T_{i+1} T_i &= T_{i+1} T_i T_{i+1} \quad \text{for } 1 \leq i < n. \end{aligned}$$

The parameter q is called the *Hecke parameter* and the parameters $u_0, \dots, u_{r-1}$ are called the *cyclotomic parameters*. Thus, the Ariki-Koike algebra is a deformation of the group algebra of the complex reflection group $G(r, 1, n) = \mathbf{Z}/r\mathbf{Z} \wr \mathfrak{S}_n$.

Next, we define *Jucys-Murphy elements*

$$L_i = q^{-i+1} T_{i-1} \cdots T_1 T_0 T_1 \cdots T_{i-1}$$

for $1 \leq i \leq n$.

Note

These element differ by a power of q from the corresponding elements in [AK1994]. However, these elements are more commonly used because they lead to nicer representation theoretic formulas.

Ariki and Koike [AK1994] showed that $H_{r,n}(q, u)$ is a free R -module with a basis given by

$$\{L_1^{c_1} \cdots L_n^{c_n} T_w \mid w \in S_n, 0 \leq c_i < r\}.$$

In particular, we have $\dim H_{r,n}(q, u) = r^n n! = |G(r, 1, n)|$. Moreover, we have $L_i L_j = L_j L_i$ for all $1 \leq i, j \leq n$.

The Ariki-Koike algebra $H_{r,n}(q, u)$ can be considered as a quotient of the group algebra of the braid group for $G(r, 1, n)$ by the ideal generated by $\prod_{i=0}^{r-1} (T_0 - u_i)$ and $(T_i - q)(T_i + 1)$. Furthermore, $H_{r,n}(q, u)$ can be constructed as a quotient of the extended affine Hecke algebra of type $A_{n-1}^{(1)}$ by $\prod_{i=0}^{r-1} (X_1 - u_i)$.

Since the Ariki-Koike algebra is a quotient of the group algebra of the braid group of $G(r, 1, n)$, we can recover the group algebra of $G(r, 1, n)$ as follows. Consider $u = (1, \zeta_r, \dots, \zeta_r^{r-1})$, where ζ_r is a primitive r -th root of unity, then we have

$$RG(r, 1, n) = H_{r,n}(1, u).$$

INPUT:

- `r` – the maximum power of L_i
- `n` – the rank S_n
- `q` – (optional) an invertible element in a commutative ring; the default is $q \in R[q, q^{-1}]$, where R is the ring containing the variables `u`
- `u` – (optional) the variables $u_1, \dots, u_r$; the default is the generators of $\mathbf{Z}[u_1, \dots, u_r]$
- `R` – (optional) a commutative ring containing q and `u`; the default is the parent of q and $u_1, \dots, u_r$
- `use_fraction_field` – (default: `False`) whether to use the fraction field or not

EXAMPLES:

We start by constructing an Ariki-Koike algebra where the values q, u are generic and do some computations:


```
sage: H = algebras.ArikiKoike(3, 4)
```

```
>>> from sage.all import *
>>> H = algebras.ArikiKoike(Integer(3), Integer(4))
```

Next, we do some computations using the LT basis:

```
sage: LT = H.LT()
sage: LT.inject_variables()
Defining L1, L2, L3, L4, T1, T2, T3
sage: T1 * T2 * T1 * T2
q*T[2,1] - (1-q)*T[2,1,2]
sage: T1 * L1 * T2 * L3 * T1 * T2
-(q-q^2)*L2*L3*T[2] + q*L1*L2*T[2,1] - (1-q)*L1*L2*T[2,1,2]
sage: L1^3
u0*u1*u2 - ((u0*u1+u0*u2+u1*u2))*L1 + ((u0+u1+u2))*L1^2
sage: L3 * L2 * L1
L1*L2*L3
sage: u = LT.u()
sage: q = LT.q()
sage: (q + 2*u[0]) * (T1 * T2) * L3
-(2*u0+(-2*u0+1)*q-q^2)*L3*T[1] - (2*u0+(-2*u0+1)*q-q^2)*L2*T[2]
+ (2*u0+q)*L1*T[1,2]
```

```
>>> from sage.all import *
>>> LT = H.LT()
>>> LT.inject_variables()
Defining L1, L2, L3, L4, T1, T2, T3
>>> T1 * T2 * T1 * T2
q*T[2,1] - (1-q)*T[2,1,2]
>>> T1 * L1 * T2 * L3 * T1 * T2
-(q-q^2)*L2*L3*T[2] + q*L1*L2*T[2,1] - (1-q)*L1*L2*T[2,1,2]
>>> L1**Integer(3)
u0*u1*u2 - ((u0*u1+u0*u2+u1*u2))*L1 + ((u0+u1+u2))*L1^2
>>> L3 * L2 * L1
L1*L2*L3
>>> u = LT.u()
>>> q = LT.q()
>>> (q + Integer(2)*u[Integer(0)]) * (T1 * T2) * L3
-(2*u0+(-2*u0+1)*q-q^2)*L3*T[1] - (2*u0+(-2*u0+1)*q-q^2)*L2*T[2]
+ (2*u0+q)*L1*T[1,2]
```

We check the defining relations:

```
sage: prod(L1 - val for val in u) == H.zero()
True
sage: L1 * T1 * L1 * T1 == T1 * L1 * T1 * L1
True
sage: T1 * T2 * T1 == T2 * T1 * T2
True
sage: T2 * T3 * T2 == T3 * T2 * T3
True
sage: L2 == q^-1 * T1 * L1 * T1
```

(continues on next page)

(continued from previous page)

```
True
sage: L3 == q^-2 * T2 * T1 * L1 * T1 * T2
True
```

```
>>> from sage.all import *
>>> prod(L1 - val for val in u) == H.zero()
True
>>> L1 * T1 * L1 * T1 == T1 * L1 * T1 * L1
True
>>> T1 * T2 * T1 == T2 * T1 * T2
True
>>> T2 * T3 * T2 == T3 * T2 * T3
True
>>> L2 == q**Integer(1) * T1 * L1 * T1
True
>>> L3 == q**Integer(2) * T2 * T1 * L1 * T1 * T2
True
```

We construct an Ariki-Koike algebra with $u = (1, \zeta_3, \zeta_3^2)$, where ζ_3 is a primitive third root of unity:

```
sage: # needs sage.rings.number_field
sage: F = CyclotomicField(3)
sage: zeta3 = F.gen()
sage: R.<q> = LaurentPolynomialRing(F)
sage: H = algebras.ArikiKoike(3, 4, q=q, u=[1, zeta3, zeta3^2], R=R)
sage: H.LT().inject_variables()
Defining L1, L2, L3, L4, T1, T2, T3
sage: L1^3
1
sage: L2^3
1 - (q^-1-1)*T[1] - (q^-1-1)*L1*L2^2*T[1] - (q^-1-1)*L1^2*L2*T[1]
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> F = CyclotomicField(Integer(3))
>>> zeta3 = F.gen()
>>> R = LaurentPolynomialRing(F, names=('q',)); (q,) = R._first_ngens(1)
>>> H = algebras.ArikiKoike(Integer(3), Integer(4), q=q, u=[Integer(1), zeta3,
↳ zeta3**Integer(2)], R=R)
>>> H.LT().inject_variables()
Defining L1, L2, L3, L4, T1, T2, T3
>>> L1**Integer(3)
1
>>> L2**Integer(3)
1 - (q^-1-1)*T[1] - (q^-1-1)*L1*L2^2*T[1] - (q^-1-1)*L1^2*L2*T[1]
```

Next, we additionally take $q = 1$ to obtain the group algebra of $G(r, 1, n)$:

```
sage: # needs sage.rings.number_field
sage: F = CyclotomicField(3)
sage: zeta3 = F.gen()
sage: H = algebras.ArikiKoike(3, 4, q=1, u=[1, zeta3, zeta3^2], R=F)
```

(continues on next page)

(continued from previous page)

```

sage: LT = H.LT()
sage: LT.inject_variables()
Defining L1, L2, L3, L4, T1, T2, T3
sage: A = ColoredPermutations(3, 4).algebra(F)
sage: s1, s2, s3, s0 = list(A.algebra_generators())
sage: all(L^3 == LT.one() for L in LT.L())
True
sage: J = [s0, s3*s0*s3, s2*s3*s0*s3*s2, s1*s2*s3*s0*s3*s2*s1]
sage: all(Ji^3 == A.one() for Ji in J)
True

```

```

>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> F = CyclotomicField(Integer(3))
>>> zeta3 = F.gen()
>>> H = algebras.ArikiKoike(Integer(3), Integer(4), q=Integer(1), u=[Integer(1),
↳ zeta3, zeta3**Integer(2)], R=F)
>>> LT = H.LT()
>>> LT.inject_variables()
Defining L1, L2, L3, L4, T1, T2, T3
>>> A = ColoredPermutations(Integer(3), Integer(4)).algebra(F)
>>> s1, s2, s3, s0 = list(A.algebra_generators())
>>> all(L**Integer(3) == LT.one() for L in LT.L())
True
>>> J = [s0, s3*s0*s3, s2*s3*s0*s3*s2, s1*s2*s3*s0*s3*s2*s1]
>>> all(Ji**Integer(3) == A.one() for Ji in J)
True

```

class `LT(algebra)`

Bases: `_Basis`

The basis of the Ariki-Koike algebra given by monomials of the form LT , where L is product of Jucys-Murphy elements and T is a product of $\{T_i | 0 < i < n\}$.

This was the basis defined in [AK1994] except using the renormalized Jucys-Murphy elements.

class `Element`

Bases: `IndexedFreeModuleElement`

`L(i=None)`

Return the generator(s) L_i .

INPUT:

- i – (default: None) the generator L_i or if None, then the list of all generators L_i

EXAMPLES:

```

sage: LT = algebras.ArikiKoike(8, 3).LT()
sage: LT.L(2)
L2
sage: LT.L()
[L1, L2, L3]

sage: LT = algebras.ArikiKoike(1, 3).LT()
sage: LT.L(2)

```

(continues on next page)

(continued from previous page)

```

u - (u*q^-1-u)*T[1]
sage: LT.L()
[u,
 u - (u*q^-1-u)*T[1],
 u - (u*q^-1-u)*T[2] - (u*q^-2-u*q^-1)*T[2,1,2]]

```

```

>>> from sage.all import *
>>> LT = algebras.ArikiKoike(Integer(8), Integer(3)).LT()
>>> LT.L(Integer(2))
L2
>>> LT.L()
[L1, L2, L3]

>>> LT = algebras.ArikiKoike(Integer(1), Integer(3)).LT()
>>> LT.L(Integer(2))
u - (u*q^-1-u)*T[1]
>>> LT.L()
[u,
 u - (u*q^-1-u)*T[1],
 u - (u*q^-1-u)*T[2] - (u*q^-2-u*q^-1)*T[2,1,2]]

```

T (*i=None*)Return the generator(s) T_i of self.

INPUT:

- *i* – (default: None) the generator T_i or if None, then the list of all generators T_i

EXAMPLES:

```

sage: LT = algebras.ArikiKoike(8, 3).LT()
sage: LT.T(1)
T[1]
sage: LT.T()
[L1, T[1], T[2]]
sage: LT.T(0)
L1

```

```

>>> from sage.all import *
>>> LT = algebras.ArikiKoike(Integer(8), Integer(3)).LT()
>>> LT.T(Integer(1))
T[1]
>>> LT.T()
[L1, T[1], T[2]]
>>> LT.T(Integer(0))
L1

```

algebra_generators()

Return the algebra generators of self.

EXAMPLES:

```

sage: LT = algebras.ArikiKoike(5, 3).LT()
sage: dict(LT.algebra_generators())

```

(continues on next page)

(continued from previous page)

```
{'L1': L1, 'L2': L2, 'L3': L3, 'T1': T[1], 'T2': T[2]}

sage: LT = algebras.ArikiKoike(1, 4).LT()
sage: dict(LT.algebra_generators())
{'T1': T[1], 'T2': T[2], 'T3': T[3]}
```

```
>>> from sage.all import *
>>> LT = algebras.ArikiKoike(Integer(5), Integer(3)).LT()
>>> dict(LT.algebra_generators())
{'L1': L1, 'L2': L2, 'L3': L3, 'T1': T[1], 'T2': T[2]}

>>> LT = algebras.ArikiKoike(Integer(1), Integer(4)).LT()
>>> dict(LT.algebra_generators())
{'T1': T[1], 'T2': T[2], 'T3': T[3]}
```

inverse_T(i)

Return the inverse of the generator T_i .

From the quadratic relation, we have

$$T_i^{-1} = q^{-1}T_i + (q^{-1} - 1).$$

EXAMPLES:

```
sage: LT = algebras.ArikiKoike(3, 4).LT()
sage: [LT.inverse_T(i) for i in range(1, 4)]
[(q^-1-1) + (q^-1)*T[1],
 (q^-1-1) + (q^-1)*T[2],
 (q^-1-1) + (q^-1)*T[3]]
```

```
>>> from sage.all import *
>>> LT = algebras.ArikiKoike(Integer(3), Integer(4)).LT()
>>> [LT.inverse_T(i) for i in range(Integer(1), Integer(4))]
[(q^-1-1) + (q^-1)*T[1],
 (q^-1-1) + (q^-1)*T[2],
 (q^-1-1) + (q^-1)*T[3]]
```

product_on_basis(m1, m2)

Return the product of the basis elements indexed by m1 and m2.

EXAMPLES:

```
sage: LT = algebras.ArikiKoike(6, 3).LT()
sage: m = ((1, 0, 2), Permutations(3)([2, 1, 3]))
sage: LT.product_on_basis(m, m)
q*L1*L2*L3^4

sage: LT = algebras.ArikiKoike(4, 3).LT()
sage: L1, L2, L3, T1, T2 = LT.algebra_generators()
sage: L1 * T1 * L1^2 * T1
q*L1*L2^2 + (1-q)*L1^2*L2*T[1]
sage: L1^2 * T1 * L1^2 * T1
q*L1^2*L2^2 + (1-q)*L1^3*L2*T[1]
```

(continues on next page)

(continued from previous page)

```

sage: L1^3 * T1 * L1^2 * T1
- (u0*u1*u2*u3-u0*u1*u2*u3*q) * L2*T[1]
+ ((u0*u1*u2+u0*u1*u3+u0*u2*u3+u1*u2*u3)+(-u0*u1*u2-u0*u1*u3-u0*u2*u3-
↪ u1*u2*u3) * q) * L1*L2*T[1]
- ((u0*u1+u0*u2+u1*u2+u0*u3+u1*u3+u2*u3)+(-u0*u1-u0*u2-u1*u2-u0*u3-u1*u3-
↪ u2*u3) * q) * L1^2*L2*T[1]
+ ((u0+u1+u2+u3)+(-u0-u1-u2-u3) * q) * L1^3*L2*T[1] + q*L1^3*L2^2

sage: L1^2 * T1 * L1^3 * T1
- (u0*u1*u2*u3-u0*u1*u2*u3*q) * L2*T[1]
+ ((u0*u1*u2+u0*u1*u3+u0*u2*u3+u1*u2*u3)+(-u0*u1*u2-u0*u1*u3-u0*u2*u3-
↪ u1*u2*u3) * q) * L1*L2*T[1]
- ((u0*u1+u0*u2+u1*u2+u0*u3+u1*u3+u2*u3)+(-u0*u1-u0*u2-u1*u2-u0*u3-u1*u3-
↪ u2*u3) * q) * L1^2*L2*T[1]
+ q*L1^2*L2^3
+ ((u0+u1+u2+u3)+(-u0-u1-u2-u3) * q) * L1^3*L2*T[1]
+ (1-q) * L1^3*L2^2*T[1]

sage: L1^2 * T1*T2*T1 * L2 * L3 * T2
(q-2*q^2+q^3) * L1^2*L2*L3 - (1-2*q+2*q^2-q^3) * L1^2*L2*L3*T[2]
- (q-q^2) * L1^3*L3*T[1] + (1-2*q+q^2) * L1^3*L3*T[1,2]
+ q*L1^3*L2*T[2,1] - (1-q) * L1^3*L2*T[2,1,2]

sage: LT = algebras.ArikiKoiike(2, 3).LT()
sage: L3 = LT.L(3)
sage: x = LT.an_element()
sage: (x * L3) * L3 == x * (L3 * L3)
True

```

```

>>> from sage.all import *
>>> LT = algebras.ArikiKoiike(Integer(6), Integer(3)).LT()
>>> m = ((Integer(1), Integer(0), Integer(2)), ↪
↪ Permutations(Integer(3))([Integer(2), Integer(1), Integer(3)]))
>>> LT.product_on_basis(m, m)
q*L1*L2*L3^4

>>> LT = algebras.ArikiKoiike(Integer(4), Integer(3)).LT()
>>> L1, L2, L3, T1, T2 = LT.algebra_generators()
>>> L1 * T1 * L1**Integer(2) * T1
q*L1*L2^2 + (1-q) * L1^2*L2*T[1]
>>> L1**Integer(2) * T1 * L1**Integer(2) * T1
q*L1^2*L2^2 + (1-q) * L1^3*L2*T[1]
>>> L1**Integer(3) * T1 * L1**Integer(2) * T1
- (u0*u1*u2*u3-u0*u1*u2*u3*q) * L2*T[1]
+ ((u0*u1*u2+u0*u1*u3+u0*u2*u3+u1*u2*u3)+(-u0*u1*u2-u0*u1*u3-u0*u2*u3-
↪ u1*u2*u3) * q) * L1*L2*T[1]
- ((u0*u1+u0*u2+u1*u2+u0*u3+u1*u3+u2*u3)+(-u0*u1-u0*u2-u1*u2-u0*u3-u1*u3-
↪ u2*u3) * q) * L1^2*L2*T[1]
+ ((u0+u1+u2+u3)+(-u0-u1-u2-u3) * q) * L1^3*L2*T[1] + q*L1^3*L2^2

>>> L1**Integer(2) * T1 * L1**Integer(3) * T1
- (u0*u1*u2*u3-u0*u1*u2*u3*q) * L2*T[1]

```

(continues on next page)

(continued from previous page)

```

+ ((u0*u1*u2+u0*u1*u3+u0*u2*u3+u1*u2*u3)+(-u0*u1*u2-u0*u1*u3-u0*u2*u3-
↪ u1*u2*u3)*q)*L1*L2*T[1]
- ((u0*u1+u0*u2+u1*u2+u0*u3+u1*u3+u2*u3)+(-u0*u1-u0*u2-u1*u2-u0*u3-u1*u3-
↪ u2*u3)*q)*L1^2*L2*T[1]
+ q*L1^2*L2^3
+ ((u0+u1+u2+u3)+(-u0-u1-u2-u3)*q)*L1^3*L2*T[1]
+ (1-q)*L1^3*L2^2*T[1]

>>> L1**Integer(2) * T1*T2*T1 * L2 * L3 * T2
(q-2*q^2+q^3)*L1^2*L2*L3 - (1-2*q+2*q^2-q^3)*L1^2*L2*L3*T[2]
- (q-q^2)*L1^3*L3*T[1] + (1-2*q+q^2)*L1^3*L3*T[1,2]
+ q*L1^3*L2*T[2,1] - (1-q)*L1^3*L2*T[2,1,2]

>>> LT = algebras.ArikiKoike(Integer(2), Integer(3)).LT()
>>> L3 = LT.L(Integer(3))
>>> x = LT.an_element()
>>> (x * L3) * L3 == x * (L3 * L3)
True

```

class **T** (*algebra*)Bases: `_Basis`The basis of the Ariki-Koike algebra given by monomials of the generators $\{T_i | 0 \leq i < n\}$.

We use the choice of reduced expression given by [BM1997]:

$$T_{1,a_1} \cdots T_{n,a_n} T_w,$$

where $T_{i,k} = T_{i-1} \cdots T_2 T_1 T_0^k$ (note that $T_{1,k} = T_0^k$) and w is a reduced expression of an element in $\mathfrak{S}_n$.**L** (*i=None*)Return the Jucys-Murphy element(s) L_i .The Jucys-Murphy element L_i is defined as

$$L_i = q^{-i+1} T_{i-1} \cdots T_1 T_0 T_1 \cdots T_{i-1} = q^{-1} T_{i-1} L_{i-1} T_{i-1}.$$

INPUT:

- *i* – (default: None) the Jucys-Murphy element L_i or if None, then the list of all L_i

EXAMPLES:

```

sage: T = algebras.ArikiKoike(8, 3).T()
sage: T.L(2)
(q^-1)*T[1,0,1]
sage: T.L()
[T[0], (q^-1)*T[1,0,1], (q^-2)*T[2,1,0,1,2]]

sage: T0,T1,T2 = T.T()
sage: q = T.q()
sage: T.L(1) == T0
True
sage: T.L(2) == q^-1 * T1*T0*T1
True
sage: T.L(3) == q^-2 * T2*T1*T0*T1*T2

```

(continues on next page)

(continued from previous page)

```

True

sage: T = algebras.ArikiKoike(1, 3).T()
sage: T.L(2)
u - (u*q^-1-u)*T[1]
sage: T.L()
[u,
 u - (u*q^-1-u)*T[1],
 u - (u*q^-1-u)*T[2] - (u*q^-2-u*q^-1)*T[2,1,2]]

```

```

>>> from sage.all import *
>>> T = algebras.ArikiKoike(Integer(8), Integer(3)).T()
>>> T.L(Integer(2))
(q^-1)*T[1,0,1]
>>> T.L()
[T[0], (q^-1)*T[1,0,1], (q^-2)*T[2,1,0,1,2]]

>>> T0,T1,T2 = T.T()
>>> q = T.q()
>>> T.L(Integer(1)) == T0
True
>>> T.L(Integer(2)) == q**Integer(1) * T1*T0*T1
True
>>> T.L(Integer(3)) == q**Integer(2) * T2*T1*T0*T1*T2
True

>>> T = algebras.ArikiKoike(Integer(1), Integer(3)).T()
>>> T.L(Integer(2))
u - (u*q^-1-u)*T[1]
>>> T.L()
[u,
 u - (u*q^-1-u)*T[1],
 u - (u*q^-1-u)*T[2] - (u*q^-2-u*q^-1)*T[2,1,2]]

```

T (*i=None*)Return the generator(s) T_i of self.

INPUT:

- *i* – (default: None) the generator T_i or if None, then the list of all generators T_i

EXAMPLES:

```

sage: T = algebras.ArikiKoike(8, 3).T()
sage: T.T(1)
T[1]
sage: T.T()
[T[0], T[1], T[2]]

sage: T = algebras.ArikiKoike(1, 4).T()

```

```

>>> from sage.all import *
>>> T = algebras.ArikiKoike(Integer(8), Integer(3)).T()
>>> T.T(Integer(1))

```

(continues on next page)

(continued from previous page)

```

T[1]
>>> T.T()
[T[0], T[1], T[2]]

>>> T = algebras.ArikiKoike(Integer(1), Integer(4)).T()

```

algebra_generators()

Return the algebra generators of self.

EXAMPLES:

```

sage: T = algebras.ArikiKoike(5, 3).T()
sage: dict(T.algebra_generators())
{0: T[0], 1: T[1], 2: T[2]}

sage: T = algebras.ArikiKoike(1, 4).T()
sage: dict(T.algebra_generators())
{1: T[1], 2: T[2], 3: T[3]}

```

```

>>> from sage.all import *
>>> T = algebras.ArikiKoike(Integer(5), Integer(3)).T()
>>> dict(T.algebra_generators())
{0: T[0], 1: T[1], 2: T[2]}

>>> T = algebras.ArikiKoike(Integer(1), Integer(4)).T()
>>> dict(T.algebra_generators())
{1: T[1], 2: T[2], 3: T[3]}

```

product_on_basis(m1, m2)

Return the product of the basis elements indexed by m1 and m2.

EXAMPLES:

```

sage: T = algebras.ArikiKoike(2, 3).T()
sage: T0, T1, T2 = T.T()
sage: T.product_on_basis(T0.leading_support(), T1.leading_support())
T[0,1]
sage: T1 * T2
T[1,2]
sage: T2 * T1
T[2,1]
sage: T2 * (T2 * T1 * T0)
-(1-q)*T[2,1,0] + q*T[1,0]
sage: (T1 * T0 * T1 * T0) * T0
-u0*u1*T[1,0,1] + ((u0+u1))*T[0,1,0,1]
sage: (T0 * T1 * T0 * T1) * (T0 * T1)
-u0*u1*q*T[1,0] + (u0*u1-u0*u1*q)*T[1,0,1]
+ ((u0+u1)*q)*T[0,1,0] - ((u0+u1)+(-u0-u1)*q)*T[0,1,0,1]
sage: T1 * (T0 * T2 * T1 * T0)
T[1,0,2,1,0]
sage: (T1 * T2) * (T2 * T1 * T0)
-(1-q)*T[2,1,0,2] - (q-q^2)*T[1,0] + q^2*T[0]
sage: (T2*T1*T2) * (T2*T1*T0*T1*T2)

```

(continues on next page)

(continued from previous page)

```

-(q-q^2)*T[2,1,0,1,2] + (1-2*q+q^2)*T[2,1,0,2,1,2]
- (q-q^2)*T[1,0,2,1,2] + q^2*T[0,2,1,2]

```

```

>>> from sage.all import *
>>> T = algebras.ArikiKoike(Integer(2), Integer(3)).T()
>>> T0, T1, T2 = T.T()
>>> T.product_on_basis(T0.leading_support(), T1.leading_support())
T[0,1]
>>> T1 * T2
T[1,2]
>>> T2 * T1
T[2,1]
>>> T2 * (T2 * T1 * T0)
-(1-q)*T[2,1,0] + q*T[1,0]
>>> (T1 * T0 * T1 * T0) * T0
-u0*u1*T[1,0,1] + ((u0+u1))*T[0,1,0,1]
>>> (T0 * T1 * T0 * T1) * (T0 * T1)
-u0*u1*q*T[1,0] + (u0*u1-u0*u1*q)*T[1,0,1]
+ ((u0+u1)*q)*T[0,1,0] - ((u0+u1)+(-u0-u1)*q)*T[0,1,0,1]
>>> T1 * (T0 * T2 * T1 * T0)
T[1,0,2,1,0]
>>> (T1 * T2) * (T2 * T1 * T0)
-(1-q)*T[2,1,0,2] - (q-q^2)*T[1,0] + q^2*T[0]
>>> (T2*T1*T2) * (T2*T1*T0*T1*T2)
-(q-q^2)*T[2,1,0,1,2] + (1-2*q+q^2)*T[2,1,0,2,1,2]
- (q-q^2)*T[1,0,2,1,2] + q^2*T[0,2,1,2]

```

We check some relations:

```

sage: T0 * T1 * T0 * T1 == T1 * T0 * T1 * T0
True
sage: T1 * T2 * T1 == T2 * T1 * T2
True
sage: (T1 * T0) * T0 == T1 * (T0 * T0)
True
sage: (T.L(1) * T.L(2)) * T.L(2) - T.L(1) * (T.L(2) * T.L(2))
0
sage: (T.L(2) * T.L(3)) * T.L(3) - T.L(2) * (T.L(3) * T.L(3))
0

```

```

>>> from sage.all import *
>>> T0 * T1 * T0 * T1 == T1 * T0 * T1 * T0
True
>>> T1 * T2 * T1 == T2 * T1 * T2
True
>>> (T1 * T0) * T0 == T1 * (T0 * T0)
True
>>> (T.L(Integer(1)) * T.L(Integer(2))) * T.L(Integer(2)) - T.
↳L(Integer(1)) * (T.L(Integer(2)) * T.L(Integer(2)))
0
>>> (T.L(Integer(2)) * T.L(Integer(3))) * T.L(Integer(3)) - T.
↳L(Integer(2)) * (T.L(Integer(3)) * T.L(Integer(3)))

```

(continues on next page)

(continued from previous page)

0

a_realization()Return a realization of *self*.

EXAMPLES:

```
sage: H = algebras.ArikiKoike(5, 2)
sage: H.a_realization()
Ariki-Koike algebra of rank 5 and order 2
with q=q and u=(u0, u1, u2, u3, u4) ... in the LT-basis
```

```
>>> from sage.all import *
>>> H = algebras.ArikiKoike(Integer(5), Integer(2))
>>> H.a_realization()
Ariki-Koike algebra of rank 5 and order 2
with q=q and u=(u0, u1, u2, u3, u4) ... in the LT-basis
```

cyclotomic_parameters()Return the cyclotomic parameters *u* of *self*.

EXAMPLES:

```
sage: H = algebras.ArikiKoike(5, 3)
sage: H.cyclotomic_parameters()
(u0, u1, u2, u3, u4)
```

```
>>> from sage.all import *
>>> H = algebras.ArikiKoike(Integer(5), Integer(3))
>>> H.cyclotomic_parameters()
(u0, u1, u2, u3, u4)
```

hecke_parameter()Return the Hecke parameter *q* of *self*.

EXAMPLES:

```
sage: H = algebras.ArikiKoike(5, 3)
sage: H.hecke_parameter()
q
```

```
>>> from sage.all import *
>>> H = algebras.ArikiKoike(Integer(5), Integer(3))
>>> H.hecke_parameter()
q
```

q()Return the Hecke parameter *q* of *self*.

EXAMPLES:

```
sage: H = algebras.ArikiKoike(5, 3)
sage: H.hecke_parameter()
q
```

```
>>> from sage.all import *
>>> H = algebras.ArikiKoike(Integer(5), Integer(3))
>>> H.hecke_parameter()
q
```

specht_module(*la*)

Return the Specht module of *self* corresponding to the shape *la*.

EXAMPLES:

```
sage: AK = algebras.ArikiKoike(4, 6)
sage: AK.specht_module([[2], [], [1, 1, 1], [1]])
Specht module of shape ([2], [], [1, 1, 1], [1]) for
Ariki-Koike algebra of rank 4 and order 6 with q=q and u=(u0, u1, u2, u3)
over ... over Integer Ring
```

```
>>> from sage.all import *
>>> AK = algebras.ArikiKoike(Integer(4), Integer(6))
>>> AK.specht_module([[Integer(2)], [], [Integer(1), Integer(1), Integer(1)],
↪ [Integer(1)]])
Specht module of shape ([2], [], [1, 1, 1], [1]) for
Ariki-Koike algebra of rank 4 and order 6 with q=q and u=(u0, u1, u2, u3)
over ... over Integer Ring
```

u()

Return the cyclotomic parameters *u* of *self*.

EXAMPLES:

```
sage: H = algebras.ArikiKoike(5, 3)
sage: H.cyclotomic_parameters()
(u0, u1, u2, u3, u4)
```

```
>>> from sage.all import *
>>> H = algebras.ArikiKoike(Integer(5), Integer(3))
>>> H.cyclotomic_parameters()
(u0, u1, u2, u3, u4)
```

6.2 Ariki-Koike Algebra Representations

AUTHORS:

- Travis Scrimshaw (2023-12-28): Initial version

class `sage.algebras.hecke_algebras.ariki_koike_specht_modules.SpechtModule`(*AK, la*)

Bases: `CombinatorialFreeModule`

Specht module of the Ariki-Koike algebra.

Let $H_{r,n}(q, u)$ be the Ariki-Koike algebra with parameters q and $u = (u_1, \dots, u_r)$ (note our indexing convention for the u parameters differs from `sage.algebras.hecke_algebras.ariki_koike_algebra`) over a commutative ring R . Let λ be a partition tuple of level r and size n . The *Specht module* of shape λ is the (right) $H_{r,n}(q, u)$ -representation S^λ given as free R -module with basis given by the standard tableau (tuples) of shape λ .

We will now describe the right action of the Ariki-Koike algebra, but we first need to set some notation and definitions. Let t be a standard tableau tuple of level r and size n . Define the *residue* of i in t to be $r_t(i) = q^{c-r}u_k$, where i is in cell (r, c) of the k -th tableau.

The action of L_i is given by $t \cdot L_i = r_T(i)t$. For T_i , we need to consider the following cases. If $i, i+1$ are in the same row (resp. column), then $t \cdot T_i = qt$ (resp. $t \cdot T_i = -t$). Otherwise if we swap $i, i+1$, the resulting tableau tuple s is again standard and the action is given by

$$t \cdot T_i = \frac{(q-1)r_t(i)}{r_s(i) - r_t(i)}t + \frac{qr_t(i) - r_s(i)}{r_s(i) - r_t(i)}s.$$

Note that $r_s(i) = r_t(i+1)$.

Over a field of characteristic 0, the set of Specht modules for all partition tuples of level r and size n form the complete set of irreducible modules for $H_{r,n}(q, u)$ [AK1994]. (The condition on the base ring can be weakened; see Theorem 3.2 of [Mathas2002].)

EXAMPLES:

We construct the Specht module $S^{(2,1,21)}$ for $H_{3,6}(q, u)$ with generic parameters q, u over $\mathbf{F}(3)$ and perform some basic computations. We change the tableaux to use the compact representation to condense the output:

```
sage: TableauTuples.options.display = 'compact'

sage: R = PolynomialRing(GF(3), 'u', 3)
sage: u = R.gens()
sage: q = R['q'].gen()
sage: H = algebras.ArikiKoike(3, 6, q, u, use_fraction_field=True)
sage: LT = H.LT()
sage: T0, T1, T2, T3, T4, T5 = LT.T()
sage: S = H.specht_module([[2], [1], [2, 1]])
sage: S.dimension()
120
sage: elt = S.an_element(); elt
S[1, 2|3|4, 5/6] - S[1, 3|2|4, 5/6] + S[1, 3|4|2, 5/6]
sage: elt * LT.L(3)
u1*S[1, 2|3|4, 5/6] + (-u0*q)*S[1, 3|2|4, 5/6] + u0*q*S[1, 3|4|2, 5/6]
sage: elt * T2
((( -u0-u1)*q-u1)/(-u0*q+u1))*S[1, 2|3|4, 5/6]
+ (((-u0+u2)*q)/(u0*q-u2))*S[1, 2|4|3, 5/6]
+ ((-u0*q^2-u0*q-u1)/(-u0*q+u1))*S[1, 3|2|4, 5/6]
+ ((u0*q^2-u0*q)/(u0*q-u2))*S[1, 3|4|2, 5/6]
sage: (elt * T3) * T2 == elt * (T3 * T2)
True
sage: elt * T2 * T3 * T2 == elt * T3 * T2 * T3
True
sage: elt * T0 * T1 * T0 * T1 == elt * T1 * T0 * T1 * T0
True
sage: elt * T2 * T5 == elt * T5 * T2
True
sage: TableauTuples.options._reset()
```

```
>>> from sage.all import *
>>> TableauTuples.options.display = 'compact'
```

(continues on next page)

(continued from previous page)

```

>>> R = PolynomialRing(GF(Integer(3)), 'u', Integer(3))
>>> u = R.gens()
>>> q = R['q'].gen()
>>> H = algebras.ArikiKoike(Integer(3), Integer(6), q, u, use_fraction_field=True)
>>> LT = H.LT()
>>> T0, T1, T2, T3, T4, T5 = LT.T()
>>> S = H.specht_module([[Integer(2)], [Integer(1)], [Integer(2), Integer(1)]])
>>> S.dimension()
120
>>> elt = S.an_element(); elt
S[1,2|3|4,5/6] - S[1,3|2|4,5/6] + S[1,3|4|2,5/6]
>>> elt * LT.L(Integer(3))
u1*S[1,2|3|4,5/6] + (-u0*q)*S[1,3|2|4,5/6] + u0*q*S[1,3|4|2,5/6]
>>> elt * T2
((( -u0-u1)*q-u1)/(-u0*q+u1))*S[1,2|3|4,5/6]
+ (((-u0+u2)*q)/(u0*q-u2))*S[1,2|4|3,5/6]
+ ((-u0*q^2-u0*q-u1)/(-u0*q+u1))*S[1,3|2|4,5/6]
+ ((u0*q^2-u0*q)/(u0*q-u2))*S[1,3|4|2,5/6]
>>> (elt * T3) * T2 == elt * (T3 * T2)
True
>>> elt * T2 * T3 * T2 == elt * T3 * T2 * T3
True
>>> elt * T0 * T1 * T0 * T1 == elt * T1 * T0 * T1 * T0
True
>>> elt * T2 * T5 == elt * T5 * T2
True
>>> TableauTuples.options._reset()

```

REFERENCES:

- [AK1994]
- [DJM1998]
- [DR2001]
- [Mathas2002]
- [Mathas2004]

class ElementBases: `IndexedFreeModuleElement`**L**(*i*)Return the (right) action of L_i on self.

INPUT:

- *i* – an integer or a list of integers

EXAMPLES:

```

sage: TableauTuples.options.display = 'compact' # compact tableau
      ↪ printing
sage: AK = algebras.ArikiKoike(3, 6, use_fraction_field=True)
sage: S = AK.specht_module([[2], [], [3,1]])
sage: elt = S.an_element(); elt

```

(continues on next page)

(continued from previous page)

```

S[1,2|-|3,4,5/6] + 2*S[1,3|-|2,4,5/6]
+ 3*S[2,3|-|1,4,5/6] + S[2,4|-|1,3,5/6]
sage: elt.L(1)
u0*S[1,2|-|3,4,5/6] + 2*u0*S[1,3|-|2,4,5/6]
+ 3*u2*S[2,3|-|1,4,5/6] + u2*S[2,4|-|1,3,5/6]
sage: elt.L(2)
u0*q*S[1,2|-|3,4,5/6] + 2*u2*S[1,3|-|2,4,5/6]
+ 3*u0*S[2,3|-|1,4,5/6] + u0*S[2,4|-|1,3,5/6]
sage: elt.L(6)
u2/q*S[1,2|-|3,4,5/6] + 2*u2/q*S[1,3|-|2,4,5/6]
+ 3*u2/q*S[2,3|-|1,4,5/6] + u2/q*S[2,4|-|1,3,5/6]
sage: elt.L([3,3,3])
u2^3*S[1,2|-|3,4,5/6] + 2*u0^3*q^3*S[1,3|-|2,4,5/6]
+ 3*u0^3*q^3*S[2,3|-|1,4,5/6] + u2^3*q^3*S[2,4|-|1,3,5/6]
sage: LT = AK.LT()
sage: elt.L([3,3,3]) == elt * (LT.L(3)^3)
True
sage: TableauTuples.options._reset() # reset

```

```

>>> from sage.all import *
>>> TableauTuples.options.display = 'compact' # compact tableau printing
>>> AK = algebras.ArikiKoike(Integer(3), Integer(6), use_fraction_
↪field=True)
>>> S = AK.specht_module([[Integer(2)], [], [Integer(3), Integer(1)]])
>>> elt = S.an_element(); elt
S[1,2|-|3,4,5/6] + 2*S[1,3|-|2,4,5/6]
+ 3*S[2,3|-|1,4,5/6] + S[2,4|-|1,3,5/6]
>>> elt.L(Integer(1))
u0*S[1,2|-|3,4,5/6] + 2*u0*S[1,3|-|2,4,5/6]
+ 3*u2*S[2,3|-|1,4,5/6] + u2*S[2,4|-|1,3,5/6]
>>> elt.L(Integer(2))
u0*q*S[1,2|-|3,4,5/6] + 2*u2*S[1,3|-|2,4,5/6]
+ 3*u0*S[2,3|-|1,4,5/6] + u0*S[2,4|-|1,3,5/6]
>>> elt.L(Integer(6))
u2/q*S[1,2|-|3,4,5/6] + 2*u2/q*S[1,3|-|2,4,5/6]
+ 3*u2/q*S[2,3|-|1,4,5/6] + u2/q*S[2,4|-|1,3,5/6]
>>> elt.L([Integer(3), Integer(3), Integer(3)])
u2^3*S[1,2|-|3,4,5/6] + 2*u0^3*q^3*S[1,3|-|2,4,5/6]
+ 3*u0^3*q^3*S[2,3|-|1,4,5/6] + u2^3*q^3*S[2,4|-|1,3,5/6]
>>> LT = AK.LT()
>>> elt.L([Integer(3), Integer(3), Integer(3)]) == elt * (LT.
↪L(Integer(3))**Integer(3))
True
>>> TableauTuples.options._reset() # reset

```

T(i)Return the (right) action of T_i on self.

INPUT:

- i – an integer or a list of integers

EXAMPLES:

```

sage: TableauTuples.options.display = 'compact' # compact tableau
      ↪printing
sage: AK = algebras.ArikiKoiike(3, 10, use_fraction_field=True)
sage: q = AK.q()
sage: S = AK.specht_module([[2,1], [], [3,2,2]])
sage: P = S.basis().keys()
sage: t = P([[2,4],[8]], [], [[1,5,7],[3,6],[9,10]])
sage: b = S.basis()[t]
sage: b.T(2)
((u2*q-u2)/(-u0*q+u2))*S[2,4/8|-|1,5,7/3,6/9,10]
+ ((u0*q^2-u2)/(-u0*q+u2))*S[3,4/8|-|1,5,7/2,6/9,10]
sage: b.T(6)
((-q)/(q+1))*S[2,4/8|-|1,5,6/3,7/9,10]
+ (q^2/(q+1))*S[2,4/8|-|1,5,7/3,6/9,10]
sage: b.T([2,1,2]) == b.T([1,2,1])
True
sage: b.T(9)
q*S[2,4/8|-|1,5,7/3,6/9,10]
sage: all(b.T([i,i]) == (q-1)*b.T(i) + q*b for i in range(1,10))
True
sage: b.T(0)
u2*S[2,4/8|-|1,5,7/3,6/9,10]
sage: b.T([0,1,0,1]) == b.T([1,0,1,0])
True
sage: TableauTuples.options._reset() # reset

```

```

>>> from sage.all import *
>>> TableauTuples.options.display = 'compact' # compact tableau printing
>>> AK = algebras.ArikiKoiike(Integer(3), Integer(10), use_fraction_
      ↪field=True)
>>> q = AK.q()
>>> S = AK.specht_module([[Integer(2),Integer(1)], [], [Integer(3),
      ↪Integer(2),Integer(2)]])
>>> P = S.basis().keys()
>>> t = P([[Integer(2),Integer(4)],[Integer(8)]], [], [[Integer(1),
      ↪Integer(5),Integer(7)],[Integer(3),Integer(6)],[Integer(9),
      ↪Integer(10)]])
>>> b = S.basis()[t]
>>> b.T(Integer(2))
((u2*q-u2)/(-u0*q+u2))*S[2,4/8|-|1,5,7/3,6/9,10]
+ ((u0*q^2-u2)/(-u0*q+u2))*S[3,4/8|-|1,5,7/2,6/9,10]
>>> b.T(Integer(6))
((-q)/(q+1))*S[2,4/8|-|1,5,6/3,7/9,10]
+ (q^2/(q+1))*S[2,4/8|-|1,5,7/3,6/9,10]
>>> b.T([Integer(2),Integer(1),Integer(2)]) == b.T([Integer(1),Integer(2),
      ↪Integer(1)])
True
>>> b.T(Integer(9))
q*S[2,4/8|-|1,5,7/3,6/9,10]
>>> all(b.T([i,i]) == (q-Integer(1))*b.T(i) + q*b for i in
      ↪range(Integer(1),Integer(10)))
True

```

(continues on next page)

(continued from previous page)

```

>>> b.T(Integer(0))
u2*S[2,4/8|-|1,5,7/3,6/9,10]
>>> b.T([Integer(0),Integer(1),Integer(0),Integer(1)]) == b.T([Integer(1),
↪Integer(0),Integer(1),Integer(0)])
True
>>> TableauTuples.options._reset() # reset

```

ariki_koike_algebra()

Return the Ariki-Koike algebra that `self` is a representation of.

EXAMPLES:

```

sage: AK = algebras.ArikiKoike(3, 6)
sage: S = AK.specht_module([[2], [], [3,1]])
sage: S.ariki_koike_algebra() is AK
True

```

```

>>> from sage.all import *
>>> AK = algebras.ArikiKoike(Integer(3), Integer(6))
>>> S = AK.specht_module([[Integer(2)], [], [Integer(3),Integer(1)]])
>>> S.ariki_koike_algebra() is AK
True

```

6.3 Iwahori-Hecke Algebras

AUTHORS:

- Daniel Bump, Nicolas Thiery (2010): Initial version
- Brant Jones, Travis Scrimshaw, Andrew Mathas (2013): Moved into the category framework and implemented the Kazhdan-Lusztig C and C' bases
- Chase Meadors, Tianyuan Xu (2021): Implemented direct computation of products in the C' basis using du Cloux's Coxeter3 package

class sage.algebras.iwahori_hecke_algebra.**IwahoriHeckeAlgebra** ($W, q1, q2, base\_ring$)

Bases: `Parent`, `UniqueRepresentation`

The Iwahori-Hecke algebra of the Coxeter group W with the specified parameters.

INPUT:

- W – a Coxeter group or Cartan type
- $q1$ – a parameter
- $q2$ – (default: -1) another parameter
- $base\_ring$ – (default: `q1.parent()`) a ring containing $q1$ and $q2$

The Iwahori-Hecke algebra [Iwa1964] is a deformation of the group algebra of a Weyl group or, more generally, a Coxeter group. These algebras are defined by generators and relations and they depend on a deformation parameter q . Taking $q = 1$, as in the following example, gives a ring isomorphic to the group algebra of the corresponding Coxeter group.

Let (W, S) be a Coxeter system and let R be a commutative ring containing elements q_1 and q_2 . Then the *Iwahori-Hecke algebra* $H = H_{q_1, q_2}(W, S)$ of (W, S) with parameters q_1 and q_2 is the unital associative algebra

with generators $\{T_s \mid s \in S\}$ and relations:

$$(T_s - q_1)(T_s - q_2) = 0$$

$$T_r T_s T_r \cdots = T_s T_r T_s \cdots,$$

where the number of terms on either side of the second relations (the braid relations) is the order of rs in the Coxeter group W , for $r, s \in S$.

Iwahori-Hecke algebras are fundamental in many areas of mathematics, ranging from the representation theory of Lie groups and quantum groups, to knot theory and statistical mechanics. For more information see, for example, [KL1979], [HKP2010], [Jon1987] and [Wikipedia article Iwahori-Hecke_algebra](#).

Bases

A reduced expression for an element $w \in W$ is any minimal length word $w = s_1 \cdots s_k$, with $s_i \in S$. If $w = s_1 \cdots s_k$ is a reduced expression for w then Matsumoto's Monoid Lemma implies that $T_w = T_{s_1} \cdots T_{s_k}$ depends on w and not on the choice of reduced expressions. Moreover, $\{T_w \mid w \in W\}$ is a basis for the Iwahori-Hecke algebra H and

$$T_s T_w = \begin{cases} T_{sw}, & \text{if } \ell(sw) = \ell(w) + 1, \\ (q_1 + q_2)T_w - q_1 q_2 T_{sw}, & \text{if } \ell(sw) = \ell(w) - 1. \end{cases}$$

The T -basis of H is implemented for any choice of parameters q_{-1} and q_{-2} :

```
sage: R.<u,v> = LaurentPolynomialRing(ZZ,2)
sage: H = IwahoriHeckeAlgebra('A3', u,v)
sage: T = H.T()
sage: T[1]
T[1]
sage: T[1,2,1] + T[2]
T[1,2,1] + T[2]
sage: T[1] * T[1,2,1]
(u+v)*T[1,2,1] + (-u*v)*T[2,1]
sage: T[1]^(-1)
(-u^(-1)*v^(-1))*T[1] + (v^(-1)+u^(-1))
```

```
>>> from sage.all import *
>>> R = LaurentPolynomialRing(ZZ,Integer(2), names=('u', 'v',)); (u, v,) = R._
->first_ngens(2)
>>> H = IwahoriHeckeAlgebra('A3', u,v)
>>> T = H.T()
>>> T[Integer(1)]
T[1]
>>> T[Integer(1),Integer(2),Integer(1)] + T[Integer(2)]
T[1,2,1] + T[2]
>>> T[Integer(1)] * T[Integer(1),Integer(2),Integer(1)]
(u+v)*T[1,2,1] + (-u*v)*T[2,1]
>>> T[Integer(1)]**Integer(1)
(-u^(-1)*v^(-1))*T[1] + (v^(-1)+u^(-1))
```

Working over the Laurent polynomial ring $Z[q^{\pm 1/2}]$ Kazhdan and Lusztig proved that there exist two distinguished bases $\{C'_w \mid w \in W\}$ and $\{C_w \mid w \in W\}$ of H which are uniquely determined by the properties that they are invariant under the bar involution on H and have triangular transitions matrices with polynomial entries of a certain form with the T -basis; see [KL1979] for a precise statement.

It turns out that the Kazhdan-Lusztig bases can be defined (by specialization) in H whenever $-q_1 q_2$ is a square in the base ring. The Kazhdan-Lusztig bases are implemented inside H whenever $-q_1 q_2$ has a square root:

```

sage: H = IwahoriHeckeAlgebra('A3', u^2, -v^2)
sage: T = H.T(); Cp = H.Cp(); C = H.C()
sage: T(Cp[1])
(u^-1*v^-1)*T[1] + (u^-1*v)
sage: T(C[1])
(u^-1*v^-1)*T[1] + (-u*v^-1)
sage: Cp(C[1])
Cp[1] + (-u*v^-1-u^-1*v)
sage: elt = Cp[2]*Cp[3]+C[1]; elt
Cp[2,3] + Cp[1] + (-u*v^-1-u^-1*v)
sage: c = C(elt); c
C[2,3] + C[1] + (u*v^-1+u^-1*v)*C[3] + (u*v^-1+u^-1*v)*C[2] + (u^2*v^-2+2+u^-2*v^
↪2)
sage: t = T(c); t
(u^-2*v^-2)*T[2,3] + (u^-1*v^-1)*T[1] + (u^-2)*T[3] + (u^-2)*T[2] + (-u*v^-1+u^-
↪2*v^2)
sage: Cp(t)
Cp[2,3] + Cp[1] + (-u*v^-1-u^-1*v)
sage: Cp(c)
Cp[2,3] + Cp[1] + (-u*v^-1-u^-1*v)

```

```

>>> from sage.all import *
>>> H = IwahoriHeckeAlgebra('A3', u**Integer(2), -v**Integer(2))
>>> T = H.T(); Cp = H.Cp(); C = H.C()
>>> T(Cp[Integer(1)])
(u^-1*v^-1)*T[1] + (u^-1*v)
>>> T(C[Integer(1)])
(u^-1*v^-1)*T[1] + (-u*v^-1)
>>> Cp(C[Integer(1)])
Cp[1] + (-u*v^-1-u^-1*v)
>>> elt = Cp[Integer(2)]*Cp[Integer(3)]+C[Integer(1)]; elt
Cp[2,3] + Cp[1] + (-u*v^-1-u^-1*v)
>>> c = C(elt); c
C[2,3] + C[1] + (u*v^-1+u^-1*v)*C[3] + (u*v^-1+u^-1*v)*C[2] + (u^2*v^-2+2+u^-2*v^
↪2)
>>> t = T(c); t
(u^-2*v^-2)*T[2,3] + (u^-1*v^-1)*T[1] + (u^-2)*T[3] + (u^-2)*T[2] + (-u*v^-1+u^-
↪2*v^2)
>>> Cp(t)
Cp[2,3] + Cp[1] + (-u*v^-1-u^-1*v)
>>> Cp(c)
Cp[2,3] + Cp[1] + (-u*v^-1-u^-1*v)

```

The conversions to and from the Kazhdan-Lusztig bases are done behind the scenes whenever the Kazhdan-Lusztig bases are well-defined. Once a suitable Iwahori-Hecke algebra is defined they will work without further intervention.

For example, with the “standard parameters”, so that $(T_r - q^2)(T_r + 1) = 0$:

```

sage: R.<q> = LaurentPolynomialRing(ZZ)
sage: H = IwahoriHeckeAlgebra('A3', q^2)
sage: T=H.T(); Cp=H.Cp(); C=H.C()
sage: C(T[1])

```

(continues on next page)

(continued from previous page)

```

q*C[1] + q^2
sage: elt = Cp(T[1,2,1]); elt
q^3*Cp[1,2,1] - q^2*Cp[2,1] - q^2*Cp[1,2] + q*Cp[1] + q*Cp[2] - 1
sage: C(elt)
q^3*C[1,2,1] + q^4*C[2,1] + q^4*C[1,2] + q^5*C[1] + q^5*C[2] + q^6

```

```

>>> from sage.all import *
>>> R = LaurentPolynomialRing(ZZ, names=('q',)); (q,) = R._first_ngens(1)
>>> H = IwahoriHeckeAlgebra('A3', q**Integer(2))
>>> T=H.T(); Cp=H.Cp(); C=H.C()
>>> C(T[Integer(1)])
q*C[1] + q^2
>>> elt = Cp(T[Integer(1),Integer(2),Integer(1)]); elt
q^3*Cp[1,2,1] - q^2*Cp[2,1] - q^2*Cp[1,2] + q*Cp[1] + q*Cp[2] - 1
>>> C(elt)
q^3*C[1,2,1] + q^4*C[2,1] + q^4*C[1,2] + q^5*C[1] + q^5*C[2] + q^6

```

With the “normalized presentation”, so that $(T_r - q)(T_r + q^{-1}) = 0$:

```

sage: R.<q> = LaurentPolynomialRing(ZZ)
sage: H = IwahoriHeckeAlgebra('A3', q, -q^-1)
sage: T=H.T(); Cp=H.Cp(); C=H.C()
sage: C(T[1])
C[1] + q
sage: elt = Cp(T[1,2,1]); elt
Cp[1,2,1] - (q^-1)*Cp[2,1] - (q^-1)*Cp[1,2] + (q^-2)*Cp[1] + (q^-2)*Cp[2] - (q^-3)
sage: C(elt)
C[1,2,1] + q*C[2,1] + q*C[1,2] + q^2*C[1] + q^2*C[2] + q^3

```

```

>>> from sage.all import *
>>> R = LaurentPolynomialRing(ZZ, names=('q',)); (q,) = R._first_ngens(1)
>>> H = IwahoriHeckeAlgebra('A3', q, -q**Integer(1))
>>> T=H.T(); Cp=H.Cp(); C=H.C()
>>> C(T[Integer(1)])
C[1] + q
>>> elt = Cp(T[Integer(1),Integer(2),Integer(1)]); elt
Cp[1,2,1] - (q^-1)*Cp[2,1] - (q^-1)*Cp[1,2] + (q^-2)*Cp[1] + (q^-2)*Cp[2] - (q^-3)
>>> C(elt)
C[1,2,1] + q*C[2,1] + q*C[1,2] + q^2*C[1] + q^2*C[2] + q^3

```

In the group algebra, so that $(T_r - 1)(T_r + 1) = 0$:

```

sage: H = IwahoriHeckeAlgebra('A3', 1)
sage: T=H.T(); Cp=H.Cp(); C=H.C()
sage: C(T[1])
C[1] + 1
sage: Cp(T[1,2,1])
Cp[1,2,1] - Cp[2,1] - Cp[1,2] + Cp[1] + Cp[2] - 1
sage: C(_)
C[1,2,1] + C[2,1] + C[1,2] + C[1] + C[2] + 1

```

```

>>> from sage.all import *
>>> H = IwahoriHeckeAlgebra('A3', Integer(1))
>>> T=H.T(); Cp=H.Cp(); C=H.C()
>>> C(T[Integer(1)])
C[1] + 1
>>> Cp(T[Integer(1), Integer(2), Integer(1)])
Cp[1,2,1] - Cp[2,1] - Cp[1,2] + Cp[1] + Cp[2] - 1
>>> C(_)
C[1,2,1] + C[2,1] + C[1,2] + C[1] + C[2] + 1

```

On the other hand, if the Kazhdan-Lusztig bases are not well-defined (when $-q_1q_2$ is not a square), attempting to use the Kazhdan-Lusztig bases triggers an error:

```

sage: R.<q>=LaurentPolynomialRing(ZZ)
sage: H = IwahoriHeckeAlgebra('A3', q)
sage: C=H.C()
Traceback (most recent call last):
...
ValueError: The Kazhdan_Lusztig bases are defined only when -q_1*q_2 is a square

```

```

>>> from sage.all import *
>>> R = LaurentPolynomialRing(ZZ, names=('q',)); (q,) = R._first_ngens(1)
>>> H = IwahoriHeckeAlgebra('A3', q)
>>> C=H.C()
Traceback (most recent call last):
...
ValueError: The Kazhdan_Lusztig bases are defined only when -q_1*q_2 is a square

```

We give an example in affine type:

```

sage: R.<v> = LaurentPolynomialRing(ZZ)
sage: H = IwahoriHeckeAlgebra(['A', 2, 1], v^2)
sage: T=H.T(); Cp=H.Cp(); C=H.C()
sage: C(T[1, 0, 2])
v^3*C[1,0,2] + v^4*C[1,0] + v^4*C[0,2] + v^4*C[1,2]
+ v^5*C[0] + v^5*C[2] + v^5*C[1] + v^6
sage: Cp(T[1, 0, 2])
v^3*Cp[1,0,2] - v^2*Cp[1,0] - v^2*Cp[0,2] - v^2*Cp[1,2]
+ v*Cp[0] + v*Cp[2] + v*Cp[1] - 1
sage: T(C[1, 0, 2])
(v^-3)*T[1,0,2] - (v^-1)*T[1,0] - (v^-1)*T[0,2] - (v^-1)*T[1,2]
+ v*T[0] + v*T[2] + v*T[1] - v^3
sage: T(Cp[1, 0, 2])
(v^-3)*T[1,0,2] + (v^-3)*T[1,0] + (v^-3)*T[0,2] + (v^-3)*T[1,2]
+ (v^-3)*T[0] + (v^-3)*T[2] + (v^-3)*T[1] + (v^-3)

```

```

>>> from sage.all import *
>>> R = LaurentPolynomialRing(ZZ, names=('v',)); (v,) = R._first_ngens(1)
>>> H = IwahoriHeckeAlgebra(['A', Integer(2), Integer(1)], v**Integer(2))
>>> T=H.T(); Cp=H.Cp(); C=H.C()
>>> C(T[Integer(1), Integer(0), Integer(2)])
v^3*C[1,0,2] + v^4*C[1,0] + v^4*C[0,2] + v^4*C[1,2]
+ v^5*C[0] + v^5*C[2] + v^5*C[1] + v^6

```

(continues on next page)

(continued from previous page)

```

>>> Cp(T[Integer(1), Integer(0), Integer(2)])
v^3*Cp[1,0,2] - v^2*Cp[1,0] - v^2*Cp[0,2] - v^2*Cp[1,2]
+ v*Cp[0] + v*Cp[2] + v*Cp[1] - 1
>>> T(C[Integer(1), Integer(0), Integer(2)])
(v^-3)*T[1,0,2] - (v^-1)*T[1,0] - (v^-1)*T[0,2] - (v^-1)*T[1,2]
+ v*T[0] + v*T[2] + v*T[1] - v^3
>>> T(Cp[Integer(1), Integer(0), Integer(2)])
(v^-3)*T[1,0,2] + (v^-3)*T[1,0] + (v^-3)*T[0,2] + (v^-3)*T[1,2]
+ (v^-3)*T[0] + (v^-3)*T[2] + (v^-3)*T[1] + (v^-3)

```

EXAMPLES:

We start by creating a Iwahori-Hecke algebra together with the three bases for these algebras that are currently supported:

```

sage: R.<v> = LaurentPolynomialRing(QQ, 'v')
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: T = H.T()
sage: C = H.C()
sage: Cp = H.Cp()

```

```

>>> from sage.all import *
>>> R = LaurentPolynomialRing(QQ, 'v', names=('v',)); (v,) = R._first_ngens(1)
>>> H = IwahoriHeckeAlgebra('A3', v**Integer(2))
>>> T = H.T()
>>> C = H.C()
>>> Cp = H.Cp()

```

It is also possible to define these three bases quickly using the `inject_shorthands()` method.

Next we create our generators for the T -basis and do some basic computations and conversions between the bases:

```

sage: T1, T2, T3 = T.algebra_generators()
sage: T1 == T[1]
True
sage: T1*T2 == T[1,2]
True
sage: T1 + T2
T[1] + T[2]
sage: T1*T1
-(1-v^2)*T[1] + v^2
sage: (T1 + T2)*T3 + T1*T1 - (v + v^-1)*T2
T[3,1] + T[2,3] - (1-v^2)*T[1] - (v^-1+v)*T[2] + v^2
sage: Cp(T1)
v*Cp[1] - 1
sage: Cp((v^1 - 1)*T1*T2 - T3)
-(v^2-v^3)*Cp[1,2] + (v-v^2)*Cp[1] - v*Cp[3] + (v-v^2)*Cp[2] + v
sage: C(T1)
v*C[1] + v^2
sage: p = C(T2*T3 - v*T1); p
v^2*C[2,3] - v^2*C[1] + v^3*C[3] + v^3*C[2] - (v^3-v^4)
sage: Cp(p)
v^2*Cp[2,3] - v^2*Cp[1] - v*Cp[3] - v*Cp[2] + (1+v)

```

(continues on next page)

(continued from previous page)

```
sage: Cp(T2*T3 - v*T1)
v^2*Cp[2,3] - v^2*Cp[1] - v*Cp[3] - v*Cp[2] + (1+v)
```

```
>>> from sage.all import *
>>> T1,T2,T3 = T.algebra_generators()
>>> T1 == T[Integer(1)]
True
>>> T1*T2 == T[Integer(1),Integer(2)]
True
>>> T1 + T2
T[1] + T[2]
>>> T1*T1
-(1-v^2)*T[1] + v^2
>>> (T1 + T2)*T3 + T1*T1 - (v + v**Integer(1))*T2
T[3,1] + T[2,3] - (1-v^2)*T[1] - (v-1+v)*T[2] + v^2
>>> Cp(T1)
v*Cp[1] - 1
>>> Cp((v**Integer(1) - Integer(1))*T1*T2 - T3)
-(v^2-v^3)*Cp[1,2] + (v-v^2)*Cp[1] - v*Cp[3] + (v-v^2)*Cp[2] + v
>>> C(T1)
v*C[1] + v^2
>>> p = C(T2*T3 - v*T1); p
v^2*C[2,3] - v^2*C[1] + v^3*C[3] + v^3*C[2] - (v^3-v^4)
>>> Cp(p)
v^2*Cp[2,3] - v^2*Cp[1] - v*Cp[3] - v*Cp[2] + (1+v)
>>> Cp(T2*T3 - v*T1)
v^2*Cp[2,3] - v^2*Cp[1] - v*Cp[3] - v*Cp[2] + (1+v)
```

In addition to explicitly creating generators, we have two shortcuts to basis elements. The first is by using elements of the underlying Coxeter group, the other is by using reduced words:

```
sage: s1,s2,s3 = H.coxeter_group().gens()
sage: T[s1*s2*s1*s3] == T[1,2,1,3]
True
sage: T[1,2,1,3] == T1*T2*T1*T3
True
```

```
>>> from sage.all import *
>>> s1,s2,s3 = H.coxeter_group().gens()
>>> T[s1*s2*s1*s3] == T[Integer(1),Integer(2),Integer(1),Integer(3)]
True
>>> T[Integer(1),Integer(2),Integer(1),Integer(3)] == T1*T2*T1*T3
True
```

Todo

Implement multi-parameter Iwahori-Hecke algebras together with their Kazhdan-Lusztig bases. That is, Iwahori-Hecke algebras with (possibly) different parameters for each conjugacy class of simple reflections in the underlying Coxeter group.

Todo

When given “generic parameters” we should return the generic Iwahori-Hecke algebra with these parameters and allow the user to work inside this algebra rather than doing calculations behind the scenes in a copy of the generic Iwahori-Hecke algebra. The main problem is that it is not clear how to recognise when the parameters are “generic”.

class `A` (*IHAlgebra*, *prefix=None*)

Bases: `_Basis`

The A -basis of an Iwahori-Hecke algebra.

The A -basis of the Iwahori-Hecke algebra is the simplest basis that is invariant under the Goldman involution $\#$, up to sign. For w in the underlying Coxeter group define:

$$A_w = T_w + (-1)^{\ell(w)} T_w^\# = T_w + (-1)^{\ell(w)} T_{w^{-1}}^{-1}$$

This gives a basis of the Iwahori-Hecke algebra whenever 2 is a unit in the base ring. The A -basis induces a $\mathbb{Z}/2\mathbb{Z}$ -grading on the Iwahori-Hecke algebra.

The A -basis is a basis only when 2 is invertible. An error is raised whenever 2 is not a unit in the base ring.

EXAMPLES:

```
sage: R.<v> = LaurentPolynomialRing(QQ, 'v')
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: A=H.A(); T=H.T()
sage: T(A[1])
T[1] + (1/2-1/2*v^2)
sage: T(A[1,2])
T[1,2] + (1/2-1/2*v^2)*T[1] + (1/2-1/2*v^2)*T[2] + (1/2-v^2+1/2*v^4)
sage: A[1]*A[2]
A[1,2] - (1/4-1/2*v^2+1/4*v^4)
```

```
>>> from sage.all import *
>>> R = LaurentPolynomialRing(QQ, 'v', names=('v',)); (v,) = R._first_ngens(1)
>>> H = IwahoriHeckeAlgebra('A3', v**Integer(2))
>>> A=H.A(); T=H.T()
>>> T(A[Integer(1)])
T[1] + (1/2-1/2*v^2)
>>> T(A[Integer(1),Integer(2)])
T[1,2] + (1/2-1/2*v^2)*T[1] + (1/2-1/2*v^2)*T[2] + (1/2-v^2+1/2*v^4)
>>> A[Integer(1)]*A[Integer(2)]
A[1,2] - (1/4-1/2*v^2+1/4*v^4)
```

goldman_involution_on_basis (w)

Return the effect of applying the Goldman involution to the basis element `self[w]`.

This function is not intended to be called directly. Instead, use `goldman_involution()`.

EXAMPLES:

```
sage: R.<v> = LaurentPolynomialRing(QQ, 'v')
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: A=H.A()
```

(continues on next page)

(continued from previous page)

```

sage: s=H.coxeter_group().simple_reflection(1)
sage: A.goldman_involution_on_basis(s)
-A[1]
sage: A[1,2].goldman_involution()
A[1,2]

```

```

>>> from sage.all import *
>>> R = LaurentPolynomialRing(QQ, 'v', names=('v',)); (v,) = R._first_
    ↪ngens(1)
>>> H = IwahoriHeckeAlgebra('A3', v**Integer(2))
>>> A=H.A()
>>> s=H.coxeter_group().simple_reflection(Integer(1))
>>> A.goldman_involution_on_basis(s)
-A[1]
>>> A[Integer(1),Integer(2)].goldman_involution()
A[1,2]

```

to_T_basis(w)

Return the A -basis element $\text{self}[w]$ as a linear combination of T -basis elements.

EXAMPLES:

```

sage: R.<v> = LaurentPolynomialRing(QQ)
sage: H = IwahoriHeckeAlgebra('A3', v**2); A=H.A(); T=H.T()
sage: s=H.coxeter_group().simple_reflection(1)
sage: A.to_T_basis(s)
T[1] + (1/2-1/2*v^2)
sage: T(A[1,2])
T[1,2] + (1/2-1/2*v^2)*T[1] + (1/2-1/2*v^2)*T[2] + (1/2-v^2+1/2*v^4)
sage: A(T[1,2])
A[1,2] - (1/2-1/2*v^2)*A[1] - (1/2-1/2*v^2)*A[2]

```

```

>>> from sage.all import *
>>> R = LaurentPolynomialRing(QQ, names=('v',)); (v,) = R._first_ngens(1)
>>> H = IwahoriHeckeAlgebra('A3', v**Integer(2)); A=H.A(); T=H.T()
>>> s=H.coxeter_group().simple_reflection(Integer(1))
>>> A.to_T_basis(s)
T[1] + (1/2-1/2*v^2)
>>> T(A[Integer(1),Integer(2)])
T[1,2] + (1/2-1/2*v^2)*T[1] + (1/2-1/2*v^2)*T[2] + (1/2-v^2+1/2*v^4)
>>> A(T[Integer(1),Integer(2)])
A[1,2] - (1/2-1/2*v^2)*A[1] - (1/2-1/2*v^2)*A[2]

```

class B(IHAlgebra, prefix=None)

Bases: `_Basis`

The B -basis of an Iwahori-Hecke algebra.

The B -basis is the unique basis of the Iwahori-Hecke algebra that is invariant under the Goldman involution, up to sign, and invariant under the Kazhdan-Lusztig bar involution. In the generic case, the B -basis becomes the group basis of the group algebra of the Coxeter group the B -basis upon setting the Hecke parameters equal to 1. If w is an element of the corresponding Coxeter group then the B -basis element B_w is uniquely

determined by the conditions that $B_w^\# = (-1)^{\ell(w)} B_w$, where $\#$ is the Goldman involution and

$$B_w = T_w + \sum_{v < w} b_{vw}(q) T_v$$

where $b_{vw}(q) \neq 0$ only if $v < w$ in the Bruhat order and $\ell(v) \not\equiv \ell(w) \pmod{2}$.

This gives a basis of the Iwahori-Hecke algebra whenever 2 is a unit in the base ring. The B -basis induces a $\mathbf{Z}/2\mathbf{Z}$ -grading on the Iwahori-Hecke algebra. The B -basis elements are also invariant under the Kazhdan-Lusztig bar involution and hence are related to the Kazhdan-Lusztig bases.

The B -basis is a basis only when 2 is invertible. An error is raised whenever 2 is not a unit in the base ring.

EXAMPLES:

```
sage: R.<v> = LaurentPolynomialRing(QQ, 'v')
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: A=H.A(); T=H.T(); Cp=H.Cp()
sage: T(A[1])
T[1] + (1/2-1/2*v^2)
sage: T(A[1,2])
T[1,2] + (1/2-1/2*v^2)*T[1] + (1/2-1/2*v^2)*T[2] + (1/2-v^2+1/2*v^4)
sage: A[1]*A[2]
A[1,2] - (1/4-1/2*v^2+1/4*v^4)
sage: Cp(A[1]*A[2])
v^2*Cp[1,2] - (1/2*v+1/2*v^3)*Cp[1] - (1/2*v+1/2*v^3)*Cp[2]
+ (1/4+1/2*v^2+1/4*v^4)
sage: Cp(A[1])
v*Cp[1] - (1/2+1/2*v^2)
sage: Cp(A[1,2])
v^2*Cp[1,2] - (1/2*v+1/2*v^3)*Cp[1]
- (1/2*v+1/2*v^3)*Cp[2] + (1/2+1/2*v^4)
sage: Cp(A[1,2,1])
v^3*Cp[1,2,1] - (1/2*v^2+1/2*v^4)*Cp[2,1]
- (1/2*v^2+1/2*v^4)*Cp[1,2] + (1/2*v+1/2*v^5)*Cp[1]
+ (1/2*v+1/2*v^5)*Cp[2] - (1/2+1/2*v^6)
```

```
>>> from sage.all import *
>>> R = LaurentPolynomialRing(QQ, 'v', names=('v',)); (v,) = R._first_ngens(1)
>>> H = IwahoriHeckeAlgebra('A3', v**Integer(2))
>>> A=H.A(); T=H.T(); Cp=H.Cp()
>>> T(A[Integer(1)])
T[1] + (1/2-1/2*v^2)
>>> T(A[Integer(1),Integer(2)])
T[1,2] + (1/2-1/2*v^2)*T[1] + (1/2-1/2*v^2)*T[2] + (1/2-v^2+1/2*v^4)
>>> A[Integer(1)]*A[Integer(2)]
A[1,2] - (1/4-1/2*v^2+1/4*v^4)
>>> Cp(A[Integer(1)]*A[Integer(2)])
v^2*Cp[1,2] - (1/2*v+1/2*v^3)*Cp[1] - (1/2*v+1/2*v^3)*Cp[2]
+ (1/4+1/2*v^2+1/4*v^4)
>>> Cp(A[Integer(1)])
v*Cp[1] - (1/2+1/2*v^2)
>>> Cp(A[Integer(1),Integer(2)])
v^2*Cp[1,2] - (1/2*v+1/2*v^3)*Cp[1]
- (1/2*v+1/2*v^3)*Cp[2] + (1/2+1/2*v^4)
```

(continues on next page)

(continued from previous page)

```
>>> Cp(A[Integer(1), Integer(2), Integer(1)])
v^3*Cp[1, 2, 1] - (1/2*v^2+1/2*v^4)*Cp[2, 1]
- (1/2*v^2+1/2*v^4)*Cp[1, 2] + (1/2*v+1/2*v^5)*Cp[1]
+ (1/2*v+1/2*v^5)*Cp[2] - (1/2+1/2*v^6)
```

goldman_involution_on_basis(w)

Return the Goldman involution to the basis element indexed by w.

This function is not intended to be called directly. Instead, use `goldman_involution()`.

EXAMPLES:

```
sage: R.<v> = LaurentPolynomialRing(QQ, 'v')
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: B=H.B()
sage: s=H.coxeter_group().simple_reflection(1)
sage: B.goldman_involution_on_basis(s)
-B[1]
sage: B[1, 2].goldman_involution()
B[1, 2]
```

```
>>> from sage.all import *
>>> R = LaurentPolynomialRing(QQ, 'v', names=('v',)); (v,) = R._first_
↳ngens(1)
>>> H = IwahoriHeckeAlgebra('A3', v**Integer(2))
>>> B=H.B()
>>> s=H.coxeter_group().simple_reflection(Integer(1))
>>> B.goldman_involution_on_basis(s)
-B[1]
>>> B[Integer(1), Integer(2)].goldman_involution()
B[1, 2]
```

to_T_basis(w)

Return the B -basis element `self[w]` as a linear combination of T -basis elements.

EXAMPLES:

```
sage: R.<v> = LaurentPolynomialRing(QQ)
sage: H = IwahoriHeckeAlgebra('A3', v**2); B=H.B(); T=H.T()
sage: s=H.coxeter_group().simple_reflection(1)
sage: B.to_T_basis(s)
T[1] + (1/2-1/2*v^2)
sage: T(B[1, 2])
T[1, 2] + (1/2-1/2*v^2)*T[1] + (1/2-1/2*v^2)*T[2]
sage: B(T[1, 2])
B[1, 2] - (1/2-1/2*v^2)*B[1] - (1/2-1/2*v^2)*B[2] + (1/2-v^2+1/2*v^4)
```

```
>>> from sage.all import *
>>> R = LaurentPolynomialRing(QQ, names=('v',)); (v,) = R._first_
↳ngens(1)
>>> H = IwahoriHeckeAlgebra('A3', v**Integer(2)); B=H.B(); T=H.T()
>>> s=H.coxeter_group().simple_reflection(Integer(1))
>>> B.to_T_basis(s)
T[1] + (1/2-1/2*v^2)
```

(continues on next page)

(continued from previous page)

```

>>> T(B(Integer(1), Integer(2)))
T[1, 2] + (1/2-1/2*v^2)*T[1] + (1/2-1/2*v^2)*T[2]
>>> B(T(Integer(1), Integer(2)))
B[1, 2] - (1/2-1/2*v^2)*B[1] - (1/2-1/2*v^2)*B[2] + (1/2-v^2+1/2*v^4)

```

class C (IHAAlgebra, prefix=None)

Bases: _KLHeckeBasis

The Kazhdan-Lusztig C -basis of Iwahori-Hecke algebra.

Assuming the standard quadratic relations of $(T_r - q)(T_r + 1) = 0$, for every element w in the Coxeter group, there is a unique element C_w in the Iwahori-Hecke algebra which is uniquely determined by the two properties:

$$\overline{C_w} = C_w$$

$$C_w = (-1)^{\ell(w)} q^{\ell(w)/2} \sum_{v \leq w} (-q)^{-\ell(v)} \overline{P_{v,w}(q)} T_v$$

where $\leq$ is the Bruhat order on the underlying Coxeter group and $P_{v,w}(q) \in \mathbf{Z}[q, q^{-1}]$ are polynomials in $\mathbf{Z}[q]$ such that $P_{w,w}(q) = 1$ and if $v < w$ then $\deg P_{v,w}(q) \leq \frac{1}{2}(\ell(w) - \ell(v) - 1)$. This is related to the C' Kazhdan-Lusztig basis by $C_i = -\alpha(C'_i)$ where α is the $\mathbf{Z}$ -linear Hecke involution defined by $q^{1/2} \mapsto q^{-1/2}$ and $\alpha(T_i) = -(q_1 q_2)^{-1/2} T_i$.

More generally, if the quadratic relations are of the form $(T_{s-q_1})(T_{s-q_2})=0$ and $\sqrt{-q_1 q_2}$ exists then, for a simple reflection s , the corresponding Kazhdan-Lusztig basis element is:

$$C_s = (-q_1 q_2)^{1/2} (1 - (-q_1 q_2)^{-1/2} T_s).$$

See [KL1979] for more details.

EXAMPLES:

```

sage: R.<v> = LaurentPolynomialRing(QQ)
sage: H = IwahoriHeckeAlgebra('A5', v**2)
sage: W = H.coxeter_group()
sage: s1,s2,s3,s4,s5 = W.simple_reflections()
sage: T = H.T()
sage: C = H.C()
sage: T(s1)**2
-(1-v^2)*T[1] + v^2
sage: T(C(s1))
(v^-1)*T[1] - v
sage: T(C(s1)*C(s2)*C(s1))
(v^-3)*T[1,2,1] - (v^-1)*T[2,1] - (v^-1)*T[1,2]
+ (v^-1+v)*T[1] + v*T[2] - (v+v^3)

```

```

>>> from sage.all import *
>>> R = LaurentPolynomialRing(QQ, names=('v',)); (v,) = R._first_ngens(1)
>>> H = IwahoriHeckeAlgebra('A5', v**Integer(2))
>>> W = H.coxeter_group()
>>> s1,s2,s3,s4,s5 = W.simple_reflections()
>>> T = H.T()
>>> C = H.C()
>>> T(s1)**Integer(2)

```

(continues on next page)

(continued from previous page)

```

-(1-v^2)*T[1] + v^2
>>> T(C(s1))
(v^-1)*T[1] - v
>>> T(C(s1)*C(s2)*C(s1))
(v^-3)*T[1,2,1] - (v^-1)*T[2,1] - (v^-1)*T[1,2]
+ (v^-1+v)*T[1] + v*T[2] - (v+v^3)

```

```

sage: R.<v> = LaurentPolynomialRing(QQ)
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: W = H.coxeter_group()
sage: s1,s2,s3 = W.simple_reflections()
sage: C = H.C()
sage: C(s1*s2*s1)
C[1,2,1]
sage: C(s1)**2
-(v^-1+v)*C[1]
sage: C(s1)*C(s2)*C(s1)
C[1,2,1] + C[1]

```

```

>>> from sage.all import *
>>> R = LaurentPolynomialRing(QQ, names=('v',)); (v,) = R._first_ngens(1)
>>> H = IwahoriHeckeAlgebra('A3', v**Integer(2))
>>> W = H.coxeter_group()
>>> s1,s2,s3 = W.simple_reflections()
>>> C = H.C()
>>> C(s1*s2*s1)
C[1,2,1]
>>> C(s1)**Integer(2)
-(v^-1+v)*C[1]
>>> C(s1)*C(s2)*C(s1)
C[1,2,1] + C[1]

```

hash_involution_on_basis(w)

Return the effect of applying the hash involution to the basis element `self[w]`.

This function is not intended to be called directly. Instead, use `hash_involution()`.

EXAMPLES:

```

sage: R.<v> = LaurentPolynomialRing(QQ, 'v')
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: C=H.C()
sage: s=H.coxeter_group().simple_reflection(1)
sage: C.hash_involution_on_basis(s)
-C[1] - (v^-1+v)
sage: C[s].hash_involution()
-C[1] - (v^-1+v)

```

```

>>> from sage.all import *
>>> R = LaurentPolynomialRing(QQ, 'v', names=('v',)); (v,) = R._first_
↪ngens(1)
>>> H = IwahoriHeckeAlgebra('A3', v**Integer(2))

```

(continues on next page)

(continued from previous page)

```

>>> C=H.C()
>>> s=H.coxeter_group().simple_reflection(Integer(1))
>>> C.hash_involution_on_basis(s)
-C[1] - (v^-1+v)
>>> C[s].hash_involution()
-C[1] - (v^-1+v)

```

C_primealias of `Cp`**class Cp** (*IHAlgebra*, *prefix=None*)Bases: `_KLHeckeBasis`The C' Kazhdan-Lusztig basis of Iwahori-Hecke algebra.

Assuming the standard quadratic relations of $(T_r - q)(T_r + 1) = 0$, for every element w in the Coxeter group, there is a unique element C'_w in the Iwahori-Hecke algebra which is uniquely determined by the two properties:

$$\overline{C'_w} = C'_w,$$

$$C'_w = q^{-\ell(w)/2} \sum_{v \leq w} P_{v,w}(q) T_v,$$

where $\leq$ is the Bruhat order on the underlying Coxeter group and $P_{v,w}(q) \in \mathbf{Z}[q, q^{-1}]$ are polynomials in $\mathbf{Z}[q]$ such that $P_{w,w}(q) = 1$ and if $v < w$ then $\deg P_{v,w}(q) \leq \frac{1}{2}(\ell(w) - \ell(v) - 1)$.

More generally, if the quadratic relations are of the form $(T_{s-q_1})(T_{s-q_2})=0$ and $\sqrt{-q_1 q_2}$ exists then, for a simple reflection s , the corresponding Kazhdan-Lusztig basis element is:

$$C'_s = (-q_1 q_2)^{-1/2} (T_s + 1).$$

See [KL1979] for more details.

If the optional `coxeter3` package is available and the Iwahori-Hecke algebra was initialized in the “standard” presentation where $\{q_1, q_2\} = \{v^2, 1\}$ as sets or the “normalized” presentation where $\{q_1, q_2\} = \{v, -v^{-1}\}$ as sets, the function `product_on_basis()` in this class computes products in the C' -basis directly in the basis itself, using `coxeter3` to calculate certain μ -coefficients quickly. If the above conditions are not all met, the function computes such products indirectly, by converting elements to the T -basis, computing products there, and converting back. The indirect method can be prohibitively slow for more complex calculations; the direct method is faster.

EXAMPLES:

```

sage: R = LaurentPolynomialRing(QQ, 'v')
sage: v = R.gen(0)
sage: H = IwahoriHeckeAlgebra('A5', v**2)
sage: W = H.coxeter_group()
sage: s1,s2,s3,s4,s5 = W.simple_reflections()
sage: T = H.T()
sage: Cp = H.Cp()
sage: T(s1)**2
-(1-v^2)*T[1] + v^2
sage: T(Cp(s1))
(v^-1)*T[1] + (v^-1)
sage: T(Cp(s1)*Cp(s2)*Cp(s1))

```

(continues on next page)

(continued from previous page)

```
(v^-3)*T[1,2,1] + (v^-3)*T[2,1] + (v^-3)*T[1,2]
+ (v^-3+v^-1)*T[1] + (v^-3)*T[2] + (v^-3+v^-1)
```

```
>>> from sage.all import *
>>> R = LaurentPolynomialRing(QQ, 'v')
>>> v = R.gen(Integer(0))
>>> H = IwahoriHeckeAlgebra('A5', v**Integer(2))
>>> W = H.coxeter_group()
>>> s1,s2,s3,s4,s5 = W.simple_reflections()
>>> T = H.T()
>>> Cp = H.Cp()
>>> T(s1)**Integer(2)
-(1-v^2)*T[1] + v^2
>>> T(Cp(s1))
(v^-1)*T[1] + (v^-1)
>>> T(Cp(s1)*Cp(s2)*Cp(s1))
(v^-3)*T[1,2,1] + (v^-3)*T[2,1] + (v^-3)*T[1,2]
+ (v^-3+v^-1)*T[1] + (v^-3)*T[2] + (v^-3+v^-1)
```

```
sage: R = LaurentPolynomialRing(QQ, 'v')
sage: v = R.gen(0)
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: W = H.coxeter_group()
sage: s1,s2,s3 = W.simple_reflections()
sage: Cp = H.Cp()
sage: Cp(s1*s2*s1)
Cp[1,2,1]
sage: Cp(s1)**2
(v^-1+v)*Cp[1]
sage: Cp(s1)*Cp(s2)*Cp(s1)
Cp[1,2,1] + Cp[1]
sage: Cp(s1)*Cp(s2)*Cp(s3)*Cp(s1)*Cp(s2)    # long time
Cp[1,2,3,1,2] + Cp[1,2,1] + Cp[3,1,2]
```

```
>>> from sage.all import *
>>> R = LaurentPolynomialRing(QQ, 'v')
>>> v = R.gen(Integer(0))
>>> H = IwahoriHeckeAlgebra('A3', v**Integer(2))
>>> W = H.coxeter_group()
>>> s1,s2,s3 = W.simple_reflections()
>>> Cp = H.Cp()
>>> Cp(s1*s2*s1)
Cp[1,2,1]
>>> Cp(s1)**Integer(2)
(v^-1+v)*Cp[1]
>>> Cp(s1)*Cp(s2)*Cp(s1)
Cp[1,2,1] + Cp[1]
>>> Cp(s1)*Cp(s2)*Cp(s3)*Cp(s1)*Cp(s2)    # long time
Cp[1,2,3,1,2] + Cp[1,2,1] + Cp[3,1,2]
```

In the following product computations, whether `coxeter3` is installed makes a big difference: without `coxeter3` the product in type H_4 takes about 5 seconds to compute and the product in type A_9 seems infeasible,

while with `coxeter3` both the computations are instant:

```
sage: H = IwahoriHeckeAlgebra('H4', v**2) # optional - coxeter3
sage: Cp = H.Cp() # optional - coxeter3
sage: Cp[3,4,3]*Cp[3,4,3,4]*Cp[1,2,3,4] # optional - coxeter3
(v^-2+2+v^2)*Cp[3,4,3,4,1,2,3,4,2]
+ (v^-2+2+v^2)*Cp[3,4,3,4,3,1,2]
+ (v^-3+3*v^-1+3*v+v^3)*Cp[3,4,3,4,3,1]
+ (v^-1+v)*Cp[3,4,1,2,3,4]
+ (v^-1+v)*Cp[3,4,1,2]

sage: H = IwahoriHeckeAlgebra('A9', v**2) # optional - coxeter3
sage: Cp = H.Cp() # optional - coxeter3
sage: Cp[1,2,1,8,9,8]*Cp[1,2,3,7,8,9] # optional - coxeter3
(v^-2+2+v^2)*Cp[7,8,9,7,8,7,1,2,3,1]
+ (v^-2+2+v^2)*Cp[8,9,8,7,1,2,3,1]
+ (v^-3+3*v^-1+3*v+v^3)*Cp[8,9,8,1,2,3,1]
```

```
>>> from sage.all import *
>>> H = IwahoriHeckeAlgebra('H4', v**Integer(2)) # optional - coxeter3
>>> Cp = H.Cp() # optional - coxeter3
>>> Cp[Integer(3),Integer(4),Integer(3)]*Cp[Integer(3),Integer(4),Integer(3),
↪Integer(4)]*Cp[Integer(1),Integer(2),Integer(3),Integer(4)] # optional -
↪ coxeter3
(v^-2+2+v^2)*Cp[3,4,3,4,1,2,3,4,2]
+ (v^-2+2+v^2)*Cp[3,4,3,4,3,1,2]
+ (v^-3+3*v^-1+3*v+v^3)*Cp[3,4,3,4,3,1]
+ (v^-1+v)*Cp[3,4,1,2,3,4]
+ (v^-1+v)*Cp[3,4,1,2]

>>> H = IwahoriHeckeAlgebra('A9', v**Integer(2)) # optional - coxeter3
>>> Cp = H.Cp() # optional - coxeter3
>>> Cp[Integer(1),Integer(2),Integer(1),Integer(8),Integer(9),
↪Integer(8)]*Cp[Integer(1),Integer(2),Integer(3),Integer(7),Integer(8),
↪Integer(9)] # optional - coxeter3
(v^-2+2+v^2)*Cp[7,8,9,7,8,7,1,2,3,1]
+ (v^-2+2+v^2)*Cp[8,9,8,7,1,2,3,1]
+ (v^-3+3*v^-1+3*v+v^3)*Cp[8,9,8,1,2,3,1]
```

To use `coxeter3` for product computations most efficiently, we recommend creating the Iwahori-Hecke algebra from a Coxeter group implemented with `coxeter3` to avoid unnecessary conversions, as in the following example with the same product computed in the last one:

```
sage: # optional - coxeter3
sage: R = LaurentPolynomialRing(QQ, 'v')
sage: v = R.gen(0)
sage: W = CoxeterGroup('A9', implementation='coxeter3')
sage: H = IwahoriHeckeAlgebra(W, v**2)
sage: Cp = H.Cp()
sage: Cp[1,2,1,8,9,8]*Cp[1,2,3,7,8,9]
(v^-2+2+v^2)*Cp[1,2,1,3,7,8,7,9,8,7]
+ (v^-2+2+v^2)*Cp[1,2,1,3,8,9,8,7]
+ (v^-3+3*v^-1+3*v+v^3)*Cp[1,2,1,3,8,9,8]
```



```

>>> from sage.all import *
>>> # optional - coxeter3
>>> R = LaurentPolynomialRing(QQ, 'v')
>>> v = R.gen(Integer(0))
>>> W = CoxeterGroup('A9', implementation='coxeter3')
>>> H = IwahoriHeckeAlgebra(W, v**Integer(2))
>>> Cp = H.Cp()
>>> Cp[Integer(1), Integer(2), Integer(1), Integer(8), Integer(9),
↪ Integer(8)] * Cp[Integer(1), Integer(2), Integer(3), Integer(7), Integer(8),
↪ Integer(9)]
(v^-2+2+v^2) * Cp[1, 2, 1, 3, 7, 8, 7, 9, 8, 7]
+ (v^-2+2+v^2) * Cp[1, 2, 1, 3, 8, 9, 8, 7]
+ (v^-3+3*v^-1+3*v+v^3) * Cp[1, 2, 1, 3, 8, 9, 8]

```

hash_involution_on_basis(w)

Return the effect of applying the hash involution to the basis element `self[w]`.

This function is not intended to be called directly. Instead, use `hash_involution()`.

EXAMPLES:

```

sage: R.<v> = LaurentPolynomialRing(QQ, 'v')
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: Cp=H.Cp()
sage: s=H.coxeter_group().simple_reflection(1)
sage: Cp.hash_involution_on_basis(s)
-Cp[1] + (v^-1+v)
sage: Cp[s].hash_involution()
-Cp[1] + (v^-1+v)

```

```

>>> from sage.all import *
>>> R = LaurentPolynomialRing(QQ, 'v', names=('v',)); (v,) = R._first_
↪ ngens(1)
>>> H = IwahoriHeckeAlgebra('A3', v**Integer(2))
>>> Cp=H.Cp()
>>> s=H.coxeter_group().simple_reflection(Integer(1))
>>> Cp.hash_involution_on_basis(s)
-Cp[1] + (v^-1+v)
>>> Cp[s].hash_involution()
-Cp[1] + (v^-1+v)

```

product_on_basis(w1, w2)

Return the expansion of $C'_{w_1} \cdot C'_{w_2}$ in the C' -basis.

If `coxeter3` is installed and the Iwahori-Hecke algebra is in the standard or normalized presentation, the product is computed directly using the method described in ALGORITHM. If not, the product is computed indirectly by converting the factors to the T -basis, computing the product there, and converting back.

The following formulas for products of the forms $C'_s \cdot C'_w$ and $C'_w \cdot C'_s$, where s is a generator of the Coxeter group and w an arbitrary element, are key to the direct computation method. The formulas are

valid for both the standard and normalized presentation of the Hecke algebra.

$$C'_s \cdot C'_w = \begin{cases} (q + q^{-1})C'_w, & \text{if } \ell(sw) = \ell(w) - 1, \\ C'_{sw} + \sum_{v \leq w, sv \leq v} \mu(v, w)C'_v, & \text{if } \ell(sw) = \ell(w) + 1. \end{cases}$$

$$C'_w \cdot C'_s = \begin{cases} (q + q^{-1})C'_w, & \text{if } \ell(ws) = \ell(w) - 1, \\ C'_{ws} + \sum_{v \leq w, vs \leq v} \mu(v, w)C'_v, & \text{if } \ell(ws) = \ell(w) + 1. \end{cases}$$

In the above, $\leq$ is the Bruhat order on the Coxeter group and $\mu(v, w)$ is the “leading coefficient of Kazhdan-Lusztig polynomials”; see [KL1979] and [Lus2013] for more details. The method designates the computation of the μ -coefficients to Sage’s interface to Fokko du Cloux’s `coxeter3` package, which is why the method requires the creation of the Coxeter group using the `'coxeter3'` implementation.

ALGORITHM:

The direct algorithm for computing $C'_x \cdot C'_y$ runs in two steps as follows.

If $\ell(x) \leq \ell(y)$, we first decompose C'_x into a polynomial in the generators $C'_s (s \in S)$ and then multiply that polynomial with C'_y . If $\ell(x) > \ell(y)$, we decompose C'_y into a polynomial in $C'_s (s \in S)$ and multiply that polynomial with C'_x . The second step (multiplication) is done by repeatedly applying the formulas displayed earlier directly. The first step (decomposition) is done by induction on the Bruhat order as follows: for every element $u \in W$ with length $\ell(u) > 1$, pick a left descent s of u and write $u = sw$ (so $w = su$), then note that

$$C'_u = C'_s \cdot C'_w - \sum_{v \leq u; sv < v} \mu(v, w)C'_v$$

by the earlier formulas, where the element w and all elements v ’s on the right side are lower than u in the Bruhat order; this allows us to finish the computation by decomposing the lower order terms C'_w and each C'_v . For example, for $u = 121, s = 1, w = 21$ in type A_3 we have $C'_{121} = C'_1 C'_{21} - C'_1$, where the lower order term C'_{21} further decomposes into $C'_2 C'_1$, therefore

$$C'_{121} = C'_1 C'_2 C'_1 - C'_1.$$

We note that the base cases $\ell(x) = 1$ or $\ell(x) = 0$ of the above induction occur when x is itself a Coxeter generator s or the group identity, respectively. The decomposition is trivial in these cases (we have $C'_x = C'_s$ or $C'_x = 1$, the unit of the Hecke algebra).

EXAMPLES:

```
sage: # optional - coxeter3
sage: R.<v> = LaurentPolynomialRing(ZZ, 'v')
sage: W = CoxeterGroup('A3', implementation='coxeter3')
sage: H = IwahoriHeckeAlgebra(W, v**2); Cp=H.Cp()
sage: Cp.product_on_basis(W([1,2,1]), W([3,1]))
(v^-1+v)*Cp[1,2,1,3]
sage: Cp.product_on_basis(W([1,2,1]), W([3,1,2]))
(v^-1+v)*Cp[1,2,1,3,2] + (v^-1+v)*Cp[1,2,1]
```

```
>>> from sage.all import *
>>> # optional - coxeter3
>>> R = LaurentPolynomialRing(ZZ, 'v', names=('v,)); (v,) = R._first_
    <-ngens(1)
>>> W = CoxeterGroup('A3', implementation='coxeter3')
>>> H = IwahoriHeckeAlgebra(W, v**Integer(2)); Cp=H.Cp()
```

(continues on next page)

(continued from previous page)

```

>>> Cp.product_on_basis(W([Integer(1), Integer(2), Integer(1)]),
↳ W([Integer(3), Integer(1)]))
(v^-1+v)*Cp[1,2,1,3]
>>> Cp.product_on_basis(W([Integer(1), Integer(2), Integer(1)]),
↳ W([Integer(3), Integer(1), Integer(2)]))
(v^-1+v)*Cp[1,2,1,3,2] + (v^-1+v)*Cp[1,2,1]

```

class `T` (*algebra*, *prefix=None*)

Bases: `_Basis`

The standard basis of Iwahori-Hecke algebra.

For every simple reflection s_i of the Coxeter group, there is a corresponding generator T_i of Iwahori-Hecke algebra. These are subject to the relations:

$$(T_i - q_1)(T_i - q_2) = 0$$

together with the braid relations:

$$T_i T_j T_i \cdots = T_j T_i T_j \cdots,$$

where the number of terms on each of the two sides is the order of $s_i s_j$ in the Coxeter group.

Weyl group elements form a basis of Iwahori-Hecke algebra H with the property that if w_1 and w_2 are Coxeter group elements such that $\ell(w_1 w_2) = \ell(w_1) + \ell(w_2)$ then $T_{w_1 w_2} = T_{w_1} T_{w_2}$.

With the default value $q_2 = -1$ and with $q_1 = q$ the generating relation may be written $T_i^2 = (q-1) \cdot T_i + q \cdot 1$ as in [Iwa1964].

EXAMPLES:

```

sage: H = IwahoriHeckeAlgebra("A3", 1)
sage: T = H.T()
sage: T1,T2,T3 = T.algebra_generators()
sage: T1*T2*T3*T1*T2*T1 == T3*T2*T1*T3*T2*T3
True
sage: w0 = T(H.coxeter_group().long_element())
sage: w0
T[1,2,3,1,2,1]
sage: T = H.T(prefix='s')
sage: T.an_element()
s[1,2,3] + 2*s[1] + 3*s[2] + 1

```

```

>>> from sage.all import *
>>> H = IwahoriHeckeAlgebra("A3", Integer(1))
>>> T = H.T()
>>> T1,T2,T3 = T.algebra_generators()
>>> T1*T2*T3*T1*T2*T1 == T3*T2*T1*T3*T2*T3
True
>>> w0 = T(H.coxeter_group().long_element())
>>> w0
T[1,2,3,1,2,1]
>>> T = H.T(prefix='s')
>>> T.an_element()
s[1,2,3] + 2*s[1] + 3*s[2] + 1

```

class ElementBases: `IndexedFreeModuleElement`A class for elements of an Iwahori-Hecke algebra in the T basis.**bar_on_basis(w)**Return the bar involution of T_w , which is $T_{w^{-1}}^{-1}$.

EXAMPLES:

```
sage: R.<v> = LaurentPolynomialRing(QQ)
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: W = H.coxeter_group()
sage: s1,s2,s3 = W.simple_reflections()
sage: T = H.T()
sage: b = T.bar_on_basis(s1*s2*s3); b
(v^-6)*T[1,2,3] + (v^-6-v^-4)*T[3,1]
+ (v^-6-v^-4)*T[1,2] + (v^-6-v^-4)*T[2,3]
+ (v^-6-2*v^-4+v^-2)*T[1] + (v^-6-2*v^-4+v^-2)*T[3]
+ (v^-6-2*v^-4+v^-2)*T[2] + (v^-6-3*v^-4+3*v^-2-1)
sage: b.bar()
T[1,2,3]
```

```
>>> from sage.all import *
>>> R = LaurentPolynomialRing(QQ, names=('v',)); (v,) = R._first_ngens(1)
>>> H = IwahoriHeckeAlgebra('A3', v**Integer(2))
>>> W = H.coxeter_group()
>>> s1,s2,s3 = W.simple_reflections()
>>> T = H.T()
>>> b = T.bar_on_basis(s1*s2*s3); b
(v^-6)*T[1,2,3] + (v^-6-v^-4)*T[3,1]
+ (v^-6-v^-4)*T[1,2] + (v^-6-v^-4)*T[2,3]
+ (v^-6-2*v^-4+v^-2)*T[1] + (v^-6-2*v^-4+v^-2)*T[3]
+ (v^-6-2*v^-4+v^-2)*T[2] + (v^-6-3*v^-4+3*v^-2-1)
>>> b.bar()
T[1,2,3]
```

goldman_involution_on_basis(w)Return the Goldman involution to the basis element indexed by w .

The goldman involution is the algebra involution of the Iwahori-Hecke algebra determined by

$$T_w \mapsto (-q_1 q_2)^{\ell(w)} T_{w^{-1}}^{-1},$$

where w is an element of the corresponding Coxeter group.

This map is defined in [Iwa1964] and it is used to define the alternating subalgebra of the Iwahori-Hecke algebra, which is the fixed-point subalgebra of the Goldman involution.

This function is not intended to be called directly. Instead, use `goldman_involution()`.

EXAMPLES:

```
sage: R.<v> = LaurentPolynomialRing(QQ, 'v')
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: T=H.T()
```

(continues on next page)

(continued from previous page)

```

sage: s=H.coxeter_group().simple_reflection(1)
sage: T.goldman_involution_on_basis(s)
-T[1] - (1-v^2)
sage: T[s].goldman_involution()
-T[1] - (1-v^2)
sage: h = T[1]*T[2] + (v^3 - v^-1 + 2)*T[3,1,2,3]
sage: h.goldman_involution()
-(v^-1-2-v^3)*T[1,2,3,2]
- (v^-1-2-v+2*v^2-v^3+v^5)*T[3,1,2]
- (v^-1-2-v+2*v^2-v^3+v^5)*T[1,2,3]
- (v^-1-2-v+2*v^2-v^3+v^5)*T[2,3,2]
- (v^-1-2-2*v+4*v^2-2*v^4+2*v^5-v^7)*T[3,1]
- (v^-1-3-2*v+4*v^2-2*v^4+2*v^5-v^7)*T[1,2]
- (v^-1-2-2*v+4*v^2-2*v^4+2*v^5-v^7)*T[3,2]
- (v^-1-2-2*v+4*v^2-2*v^4+2*v^5-v^7)*T[2,3]
- (v^-1-3-2*v+5*v^2+v^3-4*v^4+v^5+2*v^6-2*v^7+v^9)*T[1]
- (v^-1-2-3*v+6*v^2+2*v^3-6*v^4+2*v^5+2*v^6-3*v^7+v^9)*T[3]
- (v^-1-3-3*v+7*v^2+2*v^3-6*v^4+2*v^5+2*v^6-3*v^7+v^9)*T[2]
- (v^-1-3-3*v+8*v^2+3*v^3-9*v^4+6*v^6-3*v^7-2*v^8+3*v^9-v^11)
sage: h.goldman_involution().goldman_involution() == h
True

```

```

>>> from sage.all import *
>>> R = LaurentPolynomialRing(QQ, 'v', names=('v',)); (v,) = R._first_
    ↪ngens(1)
>>> H = IwahoriHeckeAlgebra('A3', v**Integer(2))
>>> T=H.T()
>>> s=H.coxeter_group().simple_reflection(Integer(1))
>>> T.goldman_involution_on_basis(s)
-T[1] - (1-v^2)
>>> T[s].goldman_involution()
-T[1] - (1-v^2)
>>> h = T[Integer(1)]*T[Integer(2)] + (v**Integer(3) - v**-Integer(1) +
    ↪Integer(2))*T[Integer(3), Integer(1), Integer(2), Integer(3)]
>>> h.goldman_involution()
-(v^-1-2-v^3)*T[1,2,3,2]
- (v^-1-2-v+2*v^2-v^3+v^5)*T[3,1,2]
- (v^-1-2-v+2*v^2-v^3+v^5)*T[1,2,3]
- (v^-1-2-v+2*v^2-v^3+v^5)*T[2,3,2]
- (v^-1-2-2*v+4*v^2-2*v^4+2*v^5-v^7)*T[3,1]
- (v^-1-3-2*v+4*v^2-2*v^4+2*v^5-v^7)*T[1,2]
- (v^-1-2-2*v+4*v^2-2*v^4+2*v^5-v^7)*T[3,2]
- (v^-1-2-2*v+4*v^2-2*v^4+2*v^5-v^7)*T[2,3]
- (v^-1-3-2*v+5*v^2+v^3-4*v^4+v^5+2*v^6-2*v^7+v^9)*T[1]
- (v^-1-2-3*v+6*v^2+2*v^3-6*v^4+2*v^5+2*v^6-3*v^7+v^9)*T[3]
- (v^-1-3-3*v+7*v^2+2*v^3-6*v^4+2*v^5+2*v^6-3*v^7+v^9)*T[2]
- (v^-1-3-3*v+8*v^2+3*v^3-9*v^4+6*v^6-3*v^7-2*v^8+3*v^9-v^11)
>>> h.goldman_involution().goldman_involution() == h
True

```

hash_involution_on_basis(w)

Return the hash involution on the basis element `self[w]`.

The hash involution α is a $\mathbf{Z}$ -algebra involution of the Iwahori-Hecke algebra determined by $q^{1/2} \mapsto q^{-1/2}$, and $T_w \mapsto (-q_1 q_2)^{-\ell(w)} T_w$, for w an element of the corresponding Coxeter group.

This map is defined in [KL1979] and it is used to change between the C and C' bases because $\alpha(C_w) = (-1)^{\ell(w)} C'_w$.

This function is not intended to be called directly. Instead, use `hash_involution()`.

EXAMPLES:

```
sage: R.<v> = LaurentPolynomialRing(QQ, 'v')
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: T=H.T()
sage: s=H.coxeter_group().simple_reflection(1)
sage: T.hash_involution_on_basis(s)
-(v^-2)*T[1]
sage: T[s].hash_involution()
-(v^-2)*T[1]
sage: h = T[1]*T[2] + (v^3 - v^-1 + 2)*T[3,1,2,3]
sage: h.hash_involution()
(v^-11+2*v^-8-v^-7)*T[1,2,3,2] + (v^-4)*T[1,2]
sage: h.hash_involution().hash_involution() == h
True
```

```
>>> from sage.all import *
>>> R = LaurentPolynomialRing(QQ, 'v', names=('v',)); (v,) = R._first_
↳ngens(1)
>>> H = IwahoriHeckeAlgebra('A3', v**Integer(2))
>>> T=H.T()
>>> s=H.coxeter_group().simple_reflection(Integer(1))
>>> T.hash_involution_on_basis(s)
-(v^-2)*T[1]
>>> T[s].hash_involution()
-(v^-2)*T[1]
>>> h = T[Integer(1)]*T[Integer(2)] + (v**Integer(3) - v**-Integer(1) +
↳Integer(2))*T[Integer(3), Integer(1), Integer(2), Integer(3)]
>>> h.hash_involution()
(v^-11+2*v^-8-v^-7)*T[1,2,3,2] + (v^-4)*T[1,2]
>>> h.hash_involution().hash_involution() == h
True
```

inverse_generator(*i*)

Return the inverse of the i -th generator, if it exists.

This method is only available if the Iwahori-Hecke algebra parameters q_1 and q_2 are both invertible. In this case, the algebra generators are also invertible and this method returns the inverse of `self.algebra_generator(i)`.

EXAMPLES:

```
sage: P.<q1, q2>=QQ[]
sage: F = Frac(P)
sage: H = IwahoriHeckeAlgebra("A2", q1, q2=q2, base_ring=F).T()
sage: H.base_ring()
Fraction Field of Multivariate Polynomial Ring in q1, q2 over Rational_
↳Field
```

(continues on next page)

(continued from previous page)

```

sage: H.inverse_generator(1)
-1/(q1*q2)*T[1] + ((q1+q2)/(q1*q2))
sage: H = IwahoriHeckeAlgebra("A2", q1, base_ring=F).T()
sage: H.inverse_generator(2)
-(1/(-q1))*T[2] + ((q1-1)/(-q1))
sage: P1.<r1, r2> = LaurentPolynomialRing(QQ)
sage: H1 = IwahoriHeckeAlgebra("B2", r1, q2=r2, base_ring=P1).T()
sage: H1.base_ring()
Multivariate Laurent Polynomial Ring in r1, r2 over Rational Field
sage: H1.inverse_generator(2)
(-r1^-1*r2^-1)*T[2] + (r2^-1+r1^-1)
sage: H2 = IwahoriHeckeAlgebra("C2", r1, base_ring=P1).T()
sage: H2.inverse_generator(2)
(r1^-1)*T[2] + (-1+r1^-1)

```

```

>>> from sage.all import *
>>> P = QQ['q1, q2']; (q1, q2,) = P._first_ngens(2)
>>> F = Frac(P)
>>> H = IwahoriHeckeAlgebra("A2", q1, q2=q2, base_ring=F).T()
>>> H.base_ring()
Fraction Field of Multivariate Polynomial Ring in q1, q2 over Rational_
↪Field
>>> H.inverse_generator(Integer(1))
-1/(q1*q2)*T[1] + ((q1+q2)/(q1*q2))
>>> H = IwahoriHeckeAlgebra("A2", q1, base_ring=F).T()
>>> H.inverse_generator(Integer(2))
-(1/(-q1))*T[2] + ((q1-1)/(-q1))
>>> P1 = LaurentPolynomialRing(QQ, names=('r1', 'r2',)); (r1, r2,) = P1._
↪first_ngens(2)
>>> H1 = IwahoriHeckeAlgebra("B2", r1, q2=r2, base_ring=P1).T()
>>> H1.base_ring()
Multivariate Laurent Polynomial Ring in r1, r2 over Rational Field
>>> H1.inverse_generator(Integer(2))
(-r1^-1*r2^-1)*T[2] + (r2^-1+r1^-1)
>>> H2 = IwahoriHeckeAlgebra("C2", r1, base_ring=P1).T()
>>> H2.inverse_generator(Integer(2))
(r1^-1)*T[2] + (-1+r1^-1)

```

inverse_generators()

Return the inverses of all the generators, if they exist.

This method is only available if q_1 and q_2 are invertible. In that case, the algebra generators are also invertible.

EXAMPLES:

```

sage: P.<q> = PolynomialRing(QQ)
sage: F = Frac(P)
sage: H = IwahoriHeckeAlgebra("A2", q, base_ring=F).T()
sage: T1,T2 = H.algebra_generators()
sage: U1,U2 = H.inverse_generators()
sage: U1*T1,T1*U1
(1, 1)

```

(continues on next page)

(continued from previous page)

```

sage: P1.<q> = LaurentPolynomialRing(QQ)
sage: H1 = IwahoriHeckeAlgebra("A2", q, base_ring=P1).T(prefix='V')
sage: V1,V2 = H1.algebra_generators()
sage: W1,W2 = H1.inverse_generators()
sage: [W1,W2]
[(q^-1)*V[1] + (q^-1-1), (q^-1)*V[2] + (q^-1-1)]
sage: V1*W1, W2*V2
(1, 1)

```

```

>>> from sage.all import *
>>> P = PolynomialRing(QQ, names=('q',)); (q,) = P._first_ngens(1)
>>> F = Frac(P)
>>> H = IwahoriHeckeAlgebra("A2", q, base_ring=F).T()
>>> T1,T2 = H.algebra_generators()
>>> U1,U2 = H.inverse_generators()
>>> U1*T1,T1*U1
(1, 1)
>>> P1 = LaurentPolynomialRing(QQ, names=('q',)); (q,) = P1._first_
↳ngens(1)
>>> H1 = IwahoriHeckeAlgebra("A2", q, base_ring=P1).T(prefix='V')
>>> V1,V2 = H1.algebra_generators()
>>> W1,W2 = H1.inverse_generators()
>>> [W1,W2]
[(q^-1)*V[1] + (q^-1-1), (q^-1)*V[2] + (q^-1-1)]
>>> V1*W1, W2*V2
(1, 1)

```

product_by_generator($x, i, \text{side}='right'$)

Return $T_i \cdot x$, where T_i is the i -th generator. This is coded individually for use in `x._mul_()`.

EXAMPLES:

```

sage: R.<q> = QQ[]; H = IwahoriHeckeAlgebra("A2", q).T()
sage: T1, T2 = H.algebra_generators()
sage: [H.product_by_generator(x, 1) for x in [T1,T2]]
[(q-1)*T[1] + q, T[2,1]]
sage: [H.product_by_generator(x, 1, side = "left") for x in [T1,T2]]
[(q-1)*T[1] + q, T[1,2]]

```

```

>>> from sage.all import *
>>> R = QQ['q']; (q,) = R._first_ngens(1); H = IwahoriHeckeAlgebra("A2",
↳q).T()
>>> T1, T2 = H.algebra_generators()
>>> [H.product_by_generator(x, Integer(1)) for x in [T1,T2]]
[(q-1)*T[1] + q, T[2,1]]
>>> [H.product_by_generator(x, Integer(1), side = "left") for x in [T1,
↳T2]]
[(q-1)*T[1] + q, T[1,2]]

```

product_by_generator_on_basis($w, i, \text{side}='right'$)

Return the product $T_w T_i$ (resp. $T_i T_w$) if `side` is `'right'` (resp. `'left'`).

If the quadratic relation is $(T_i - u)(T_i - v) = 0$, then we have

$$T_w T_i = \begin{cases} T_{ws_i} & \text{if } \ell(ws_i) = \ell(w) + 1, \\ (u + v)T_{ws_i} - uvT_w & \text{if } \ell(ws_i) = \ell(w) - 1. \end{cases}$$

The left action is similar.

INPUT:

- w – an element of the Coxeter group
- i – an element of the index set
- $side$ – 'right' (default) or 'left'

EXAMPLES:

```
sage: R.<q> = QQ[]; H = IwahoriHeckeAlgebra("A2", q)
sage: T = H.T()
sage: s1,s2 = H.coxeter_group().simple_reflections()
sage: [T.product_by_generator_on_basis(w, 1) for w in [s1,s2,s1*s2]]
[(q-1)*T[1] + q, T[2,1], T[1,2,1]]
sage: [T.product_by_generator_on_basis(w, 1, side='left') for w in [s1,s2,
↪s1*s2]]
[(q-1)*T[1] + q, T[1,2], (q-1)*T[1,2] + q*T[2]]
```

```
>>> from sage.all import *
>>> R = QQ['q']; (q,) = R._first_ngens(1); H = IwahoriHeckeAlgebra("A2", q)
↪q)
>>> T = H.T()
>>> s1,s2 = H.coxeter_group().simple_reflections()
>>> [T.product_by_generator_on_basis(w, Integer(1)) for w in [s1,s2,
↪s1*s2]]
[(q-1)*T[1] + q, T[2,1], T[1,2,1]]
>>> [T.product_by_generator_on_basis(w, Integer(1), side='left') for w in
↪[s1,s2,s1*s2]]
[(q-1)*T[1] + q, T[1,2], (q-1)*T[1,2] + q*T[2]]
```

product_on_basis($w1, w2$)

Return $T_{w_1}T_{w_2}$, where w_1 and w_2 are words in the Coxeter group.

EXAMPLES:

```
sage: R.<q> = QQ[]; H = IwahoriHeckeAlgebra("A2", q)
sage: T = H.T()
sage: s1,s2 = H.coxeter_group().simple_reflections()
sage: [T.product_on_basis(s1,x) for x in [s1,s2]]
[(q-1)*T[1] + q, T[1,2]]
```

```
>>> from sage.all import *
>>> R = QQ['q']; (q,) = R._first_ngens(1); H = IwahoriHeckeAlgebra("A2", q)
↪q)
>>> T = H.T()
>>> s1,s2 = H.coxeter_group().simple_reflections()
>>> [T.product_on_basis(s1,x) for x in [s1,s2]]
[(q-1)*T[1] + q, T[1,2]]
```

to_C_basis(w)

Return T_w as a linear combination of C -basis elements.

EXAMPLES:

```

sage: R = LaurentPolynomialRing(QQ, 'v')
sage: v = R.gen(0)
sage: H = IwahoriHeckeAlgebra('A2', v**2)
sage: s1,s2 = H.coxeter_group().simple_reflections()
sage: T = H.T()
sage: C = H.C()
sage: T.to_C_basis(s1)
v*T[1] + v^2
sage: C(T(s1))
v*C[1] + v^2
sage: C(v^-1*T(s1) - v)
C[1]
sage: C(T(s1*s2)+T(s1)+T(s2)+1)
v^2*C[1,2] + (v+v^3)*C[1] + (v+v^3)*C[2] + (1+2*v^2+v^4)
sage: C(T(s1*s2*s1))
v^3*C[1,2,1] + v^4*C[2,1] + v^4*C[1,2] + v^5*C[1] + v^5*C[2] + v^6

```

```

>>> from sage.all import *
>>> R = LaurentPolynomialRing(QQ, 'v')
>>> v = R.gen(Integer(0))
>>> H = IwahoriHeckeAlgebra('A2', v**Integer(2))
>>> s1,s2 = H.coxeter_group().simple_reflections()
>>> T = H.T()
>>> C = H.C()
>>> T.to_C_basis(s1)
v*T[1] + v^2
>>> C(T(s1))
v*C[1] + v^2
>>> C(v**-Integer(1)*T(s1) - v)
C[1]
>>> C(T(s1*s2)+T(s1)+T(s2)+Integer(1))
v^2*C[1,2] + (v+v^3)*C[1] + (v+v^3)*C[2] + (1+2*v^2+v^4)
>>> C(T(s1*s2*s1))
v^3*C[1,2,1] + v^4*C[2,1] + v^4*C[1,2] + v^5*C[1] + v^5*C[2] + v^6

```

to_Cp_basis(w)Return T_w as a linear combination of C' -basis elements.

EXAMPLES:

```

sage: R.<v> = LaurentPolynomialRing(QQ)
sage: H = IwahoriHeckeAlgebra('A2', v**2)
sage: s1,s2 = H.coxeter_group().simple_reflections()
sage: T = H.T()
sage: Cp = H.Cp()
sage: T.to_Cp_basis(s1)
v*Cp[1] - 1
sage: Cp(T(s1))
v*Cp[1] - 1
sage: Cp(T(s1)+1)
v*Cp[1]
sage: Cp(T(s1*s2)+T(s1)+T(s2)+1)

```

(continues on next page)

(continued from previous page)

```
v^2*Cp[1,2]
sage: Cp(T(s1*s2*s1))
v^3*Cp[1,2,1] - v^2*Cp[2,1] - v^2*Cp[1,2] + v*Cp[1] + v*Cp[2] - 1
```

```
>>> from sage.all import *
>>> R = LaurentPolynomialRing(QQ, names=('v',)); (v,) = R._first_ngens(1)
>>> H = IwahoriHeckeAlgebra('A2', v**Integer(2))
>>> s1,s2 = H.coxeter_group().simple_reflections()
>>> T = H.T()
>>> Cp = H.Cp()
>>> T.to_Cp_basis(s1)
v*Cp[1] - 1
>>> Cp(T(s1))
v*Cp[1] - 1
>>> Cp(T(s1)+Integer(1))
v*Cp[1]
>>> Cp(T(s1*s2)+T(s1)+T(s2)+Integer(1))
v^2*Cp[1,2]
>>> Cp(T(s1*s2*s1))
v^3*Cp[1,2,1] - v^2*Cp[2,1] - v^2*Cp[1,2] + v*Cp[1] + v*Cp[2] - 1
```

a_realization()

Return a particular realization of *self* (the *T*-basis).

EXAMPLES:

```
sage: H = IwahoriHeckeAlgebra("B2", 1)
sage: H.a_realization()
Iwahori-Hecke algebra of type B2 in 1,-1 over Integer Ring in the T-basis
```

```
>>> from sage.all import *
>>> H = IwahoriHeckeAlgebra("B2", Integer(1))
>>> H.a_realization()
Iwahori-Hecke algebra of type B2 in 1,-1 over Integer Ring in the T-basis
```

cartan_type()

Return the Cartan type of *self*.

EXAMPLES:

```
sage: IwahoriHeckeAlgebra("D4", 1).cartan_type()
['D', 4]
```

```
>>> from sage.all import *
>>> IwahoriHeckeAlgebra("D4", Integer(1)).cartan_type()
['D', 4]
```

coxeter_group()

Return the Coxeter group of *self*.

EXAMPLES:

```
sage: IwahoriHeckeAlgebra("B2", 1).coxeter_group()
Finite Coxeter group over Number Field in a with defining polynomial x^2 - 2
↳with a = 1.414213562373095? with Coxeter matrix:
[1 4]
[4 1]
```

```
>>> from sage.all import *
>>> IwahoriHeckeAlgebra("B2", Integer(1)).coxeter_group()
Finite Coxeter group over Number Field in a with defining polynomial x^2 - 2
↳with a = 1.414213562373095? with Coxeter matrix:
[1 4]
[4 1]
```

coxeter_type()

Return the Coxeter type of `self`.

EXAMPLES:

```
sage: IwahoriHeckeAlgebra("D4", 1).coxeter_type()
Coxeter type of ['D', 4]
```

```
>>> from sage.all import *
>>> IwahoriHeckeAlgebra("D4", Integer(1)).coxeter_type()
Coxeter type of ['D', 4]
```

q1()

Return the parameter q_1 of `self`.

EXAMPLES:

```
sage: H = IwahoriHeckeAlgebra("B2", 1)
sage: H.q1()
1
```

```
>>> from sage.all import *
>>> H = IwahoriHeckeAlgebra("B2", Integer(1))
>>> H.q1()
1
```

q2()

Return the parameter q_2 of `self`.

EXAMPLES:

```
sage: H = IwahoriHeckeAlgebra("B2", 1)
sage: H.q2()
-1
```

```
>>> from sage.all import *
>>> H = IwahoriHeckeAlgebra("B2", Integer(1))
>>> H.q2()
-1
```

standardalias of *T***class** sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_nonstandard(*W*)Bases: *IwahoriHeckeAlgebra*

This is a class which is used behind the scenes by *IwahoriHeckeAlgebra* to compute the Kazhdan-Lusztig bases. It is not meant to be used directly. It implements the slightly idiosyncratic (but convenient) Iwahori-Hecke algebra with two parameters which is defined over the Laurent polynomial ring $\mathbf{Z}[u, u^{-1}, v, v^{-1}]$ in two variables and has quadratic relations:

$$(T_r - u)(T_r + v^2/u) = 0.$$

The point of these relations is that the product of the two parameters is v^2 which is a square in $\mathbf{Z}[u, u^{-1}, v, v^{-1}]$. Consequently, the Kazhdan-Lusztig bases are defined for this algebra.

More generally, if we have a Iwahori-Hecke algebra with two parameters which has quadratic relations of the form:

$$(T_r - q_1)(T_r - q_2) = 0$$

where $-q_1q_2$ is a square then the Kazhdan-Lusztig bases are well-defined for this algebra. Moreover, these bases be computed by specialization from the generic Iwahori-Hecke algebra using the specialization which sends $u \mapsto q_1$ and $v \mapsto \sqrt{-q_1q_2}$, so that $v^2/u \mapsto -q_2$.

For example, if $q_1 = q = Q^2$ and $q_2 = -1$ then $u \mapsto q$ and $v \mapsto \sqrt{q} = Q$; this is the standard presentation of the Iwahori-Hecke algebra with $(T_r - q)(T_r + 1) = 0$. On the other hand, when $q_1 = q$ and $q_2 = -q^{-1}$ then $u \mapsto q$ and $v \mapsto 1$. This is the normalized presentation with $(T_r - v)(T_r + v^{-1}) = 0$.

Warning

This class uses non-standard parameters for the Iwahori-Hecke algebra and are related to the standard parameters by an outer automorphism that is non-trivial on the *T*-basis.

class C(*IHAlgebra*, *prefix=None*)Bases: *C*

The Kazhdan-Lusztig *C*-basis for the generic Iwahori-Hecke algebra.

to_T_basis(*w*)

Return C_w as a linear combination of *T*-basis elements.

EXAMPLES:

```
sage: H = sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_
      ↪nonstandard("A3")
sage: s1,s2,s3 = H.coxeter_group().simple_reflections()
sage: T = H.T()
sage: C = H.C()
sage: C.to_T_basis(s1)
(v^-1)*T[1] + (-u*v^-1)
sage: C.to_T_basis(s1*s2)
(v^-2)*T[1,2] + (-u*v^-2)*T[1] + (-u*v^-2)*T[2] + (u^2*v^-2)
sage: C.to_T_basis(s1*s2*s1)
(v^-3)*T[1,2,1] + (-u*v^-3)*T[2,1] + (-u*v^-3)*T[1,2]
      + (u^2*v^-3)*T[1] + (u^2*v^-3)*T[2] + (-u^3*v^-3)
```

(continues on next page)

(continued from previous page)

```

sage: T(C(s1*s2*s1))
(v^-3)*T[1,2,1] + (-u*v^-3)*T[2,1] + (-u*v^-3)*T[1,2]
+ (u^2*v^-3)*T[1] + (u^2*v^-3)*T[2] + (-u^3*v^-3)
sage: T(C(s2*s1*s3*s2))
(v^-4)*T[2,3,1,2] + (-u*v^-4)*T[2,3,1] + (-u*v^-4)*T[1,2,1]
+ (-u*v^-4)*T[3,1,2] + (-u*v^-4)*T[2,3,2] + (u^2*v^-4)*T[2,1]
+ (u^2*v^-4)*T[3,1] + (u^2*v^-4)*T[1,2] + (u^2*v^-4)*T[3,2]
+ (u^2*v^-4)*T[2,3] + (-u^3*v^-4)*T[1] + (-u^3*v^-4)*T[3]
+ (-u^3*v^-4-u*v^-2)*T[2] + (u^4*v^-4+u^2*v^-2)

```

```

>>> from sage.all import *
>>> H = sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_
↳nonstandard("A3")
>>> s1,s2,s3 = H.coxeter_group().simple_reflections()
>>> T = H.T()
>>> C = H.C()
>>> C.to_T_basis(s1)
(v^-1)*T[1] + (-u*v^-1)
>>> C.to_T_basis(s1*s2)
(v^-2)*T[1,2] + (-u*v^-2)*T[1] + (-u*v^-2)*T[2] + (u^2*v^-2)
>>> C.to_T_basis(s1*s2*s1)
(v^-3)*T[1,2,1] + (-u*v^-3)*T[2,1] + (-u*v^-3)*T[1,2]
+ (u^2*v^-3)*T[1] + (u^2*v^-3)*T[2] + (-u^3*v^-3)
>>> T(C(s1*s2*s1))
(v^-3)*T[1,2,1] + (-u*v^-3)*T[2,1] + (-u*v^-3)*T[1,2]
+ (u^2*v^-3)*T[1] + (u^2*v^-3)*T[2] + (-u^3*v^-3)
>>> T(C(s2*s1*s3*s2))
(v^-4)*T[2,3,1,2] + (-u*v^-4)*T[2,3,1] + (-u*v^-4)*T[1,2,1]
+ (-u*v^-4)*T[3,1,2] + (-u*v^-4)*T[2,3,2] + (u^2*v^-4)*T[2,1]
+ (u^2*v^-4)*T[3,1] + (u^2*v^-4)*T[1,2] + (u^2*v^-4)*T[3,2]
+ (u^2*v^-4)*T[2,3] + (-u^3*v^-4)*T[1] + (-u^3*v^-4)*T[3]
+ (-u^3*v^-4-u*v^-2)*T[2] + (u^4*v^-4+u^2*v^-2)

```

C_primealias of *Cp***class Cp** (*IHAlgebra*, *prefix=None*)Bases: *Cp*The Kazhdan-Lusztig C' -basis for the generic Iwahori-Hecke algebra.**to_T_basis** (*w*)Return C'_w as a linear combination of T -basis elements.

EXAMPLES:

```

sage: H = sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_
↳nonstandard("A3")
sage: s1,s2,s3 = H.coxeter_group().simple_reflections()
sage: T = H.T()
sage: Cp = H.Cp()
sage: Cp.to_T_basis(s1)
(v^-1)*T[1] + (u^-1*v)

```

(continues on next page)

(continued from previous page)

```

sage: Cp.to_T_basis(s1*s2)
(v^-2)*T[1,2] + (u^-1)*T[1] + (u^-1)*T[2] + (u^-2*v^2)
sage: Cp.to_T_basis(s1*s2*s1)
(v^-3)*T[1,2,1] + (u^-1*v^-1)*T[2,1] + (u^-1*v^-1)*T[1,2]
+ (u^-2*v)*T[1] + (u^-2*v)*T[2] + (u^-3*v^3)
sage: T(Cp(s1*s2*s1))
(v^-3)*T[1,2,1] + (u^-1*v^-1)*T[2,1] + (u^-1*v^-1)*T[1,2]
+ (u^-2*v)*T[1] + (u^-2*v)*T[2] + (u^-3*v^3)
sage: T(Cp(s2*s1*s3*s2))
(v^-4)*T[2,3,1,2] + (u^-1*v^-2)*T[2,3,1] + (u^-1*v^-2)*T[1,2,1]
+ (u^-1*v^-2)*T[3,1,2] + (u^-1*v^-2)*T[2,3,2] + (u^-2)*T[2,1]
+ (u^-2)*T[3,1] + (u^-2)*T[1,2] + (u^-2)*T[3,2]
+ (u^-2)*T[2,3] + (u^-3*v^2)*T[1] + (u^-3*v^2)*T[3]
+ (u^-1+u^-3*v^2)*T[2] + (u^-2*v^2+u^-4*v^4)

```

```

>>> from sage.all import *
>>> H = sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_
↳nonstandard("A3")
>>> s1,s2,s3 = H.coxeter_group().simple_reflections()
>>> T = H.T()
>>> Cp = H.Cp()
>>> Cp.to_T_basis(s1)
(v^-1)*T[1] + (u^-1*v)
>>> Cp.to_T_basis(s1*s2)
(v^-2)*T[1,2] + (u^-1)*T[1] + (u^-1)*T[2] + (u^-2*v^2)
>>> Cp.to_T_basis(s1*s2*s1)
(v^-3)*T[1,2,1] + (u^-1*v^-1)*T[2,1] + (u^-1*v^-1)*T[1,2]
+ (u^-2*v)*T[1] + (u^-2*v)*T[2] + (u^-3*v^3)
>>> T(Cp(s1*s2*s1))
(v^-3)*T[1,2,1] + (u^-1*v^-1)*T[2,1] + (u^-1*v^-1)*T[1,2]
+ (u^-2*v)*T[1] + (u^-2*v)*T[2] + (u^-3*v^3)
>>> T(Cp(s2*s1*s3*s2))
(v^-4)*T[2,3,1,2] + (u^-1*v^-2)*T[2,3,1] + (u^-1*v^-2)*T[1,2,1]
+ (u^-1*v^-2)*T[3,1,2] + (u^-1*v^-2)*T[2,3,2] + (u^-2)*T[2,1]
+ (u^-2)*T[3,1] + (u^-2)*T[1,2] + (u^-2)*T[3,2]
+ (u^-2)*T[2,3] + (u^-3*v^2)*T[1] + (u^-3*v^2)*T[3]
+ (u^-1+u^-3*v^2)*T[2] + (u^-2*v^2+u^-4*v^4)

```

class `T`(*algebra*, *prefix=None*)

Bases: `T`

The T -basis for the generic Iwahori-Hecke algebra.

to_C_basis(*w*)

Return T_w as a linear combination of C -basis elements.

To compute this we piggy back off the C' -basis conversion using the observation that the hash involution sends T_w to $(-q_1 q_1)^{\ell(w)} T_w$ and C_w to $(-1)^{\ell(w)} C'_w$. Therefore, if

$$T_w = \sum_v a_{vw} C'_v$$

then

$$T_w = (-q_1 q_2)^{\ell(w)} \left(\sum_v a_{vw} C'_v \right)^{\#} = \sum_v (-1)^{\ell(v)} \overline{a_{vw}} C_v$$

Note that we cannot just apply `hash_involution()` here because this involution always returns the answer with respect to the same basis.

EXAMPLES:

```
sage: H = sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_
      ↪nonstandard("A2")
sage: s1,s2 = H.coxeter_group().simple_reflections()
sage: T = H.T()
sage: C = H.C()
sage: T.to_C_basis(s1)
v*T[1] + u
sage: C(T(s1))
v*C[1] + u
sage: C(T( C[1] ))
C[1]
sage: C(T(s1*s2)+T(s1)+T(s2)+1)
v^2*C[1,2] + (u*v+v)*C[1] + (u*v+v)*C[2] + (u^2+2*u+1)
sage: C(T(s1*s2*s1))
v^3*C[1,2,1] + u*v^2*C[2,1] + u*v^2*C[1,2] + u^2*v*C[1] + u^2*v*C[2] + u^3
```

```
>>> from sage.all import *
>>> H = sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_
      ↪nonstandard("A2")
>>> s1,s2 = H.coxeter_group().simple_reflections()
>>> T = H.T()
>>> C = H.C()
>>> T.to_C_basis(s1)
v*T[1] + u
>>> C(T(s1))
v*C[1] + u
>>> C(T( C[Integer(1)] ))
C[1]
>>> C(T(s1*s2)+T(s1)+T(s2)+Integer(1))
v^2*C[1,2] + (u*v+v)*C[1] + (u*v+v)*C[2] + (u^2+2*u+1)
>>> C(T(s1*s2*s1))
v^3*C[1,2,1] + u*v^2*C[2,1] + u*v^2*C[1,2] + u^2*v*C[1] + u^2*v*C[2] + u^3
```

to_Cp_basis(w)

Return T_w as a linear combination of C' -basis elements.

EXAMPLES:

```
sage: H = sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_
      ↪nonstandard("A2")
sage: s1,s2 = H.coxeter_group().simple_reflections()
sage: T = H.T()
sage: Cp = H.Cp()
sage: T.to_Cp_basis(s1)
v*Cp[1] + (-u^-1*v^2)
sage: Cp(T(s1))
v*Cp[1] + (-u^-1*v^2)
sage: Cp(T(s1)+1)
v*Cp[1] + (-u^-1*v^2+1)
```

(continues on next page)

(continued from previous page)

```

sage: Cp(T(s1*s2)+T(s1)+T(s2)+1)
v^2*Cp[1,2] + (-u^-1*v^3+v)*Cp[1] + (-u^-1*v^3+v)*Cp[2]
+ (u^-2*v^4-2*u^-1*v^2+1)
sage: Cp(T(s1*s2*s1))
v^3*Cp[1,2,1] + (-u^-1*v^4)*Cp[2,1] + (-u^-1*v^4)*Cp[1,2]
+ (u^-2*v^5)*Cp[1] + (u^-2*v^5)*Cp[2] + (-u^-3*v^6)

```

```

>>> from sage.all import *
>>> H = sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_
↳nonstandard("A2")
>>> s1,s2 = H.coxeter_group().simple_reflections()
>>> T = H.T()
>>> Cp = H.Cp()
>>> T.to_Cp_basis(s1)
v*Cp[1] + (-u^-1*v^2)
>>> Cp(T(s1))
v*Cp[1] + (-u^-1*v^2)
>>> Cp(T(s1)+Integer(1))
v*Cp[1] + (-u^-1*v^2+1)
>>> Cp(T(s1*s2)+T(s1)+T(s2)+Integer(1))
v^2*Cp[1,2] + (-u^-1*v^3+v)*Cp[1] + (-u^-1*v^3+v)*Cp[2]
+ (u^-2*v^4-2*u^-1*v^2+1)
>>> Cp(T(s1*s2*s1))
v^3*Cp[1,2,1] + (-u^-1*v^4)*Cp[2,1] + (-u^-1*v^4)*Cp[1,2]
+ (u^-2*v^5)*Cp[1] + (u^-2*v^5)*Cp[2] + (-u^-3*v^6)

```

`sage.algebras.iwahori_hecke_algebra.index_cmp(x, y)`

Compare two term indices x and y by Bruhat order, then by word length, and then by the generic comparison.

EXAMPLES:

```

sage: from sage.algebras.iwahori_hecke_algebra import index_cmp
sage: W = WeylGroup(['A', 2, 1])
sage: x = W.from_reduced_word([0, 1])
sage: y = W.from_reduced_word([0, 2, 1])
sage: x.bruhat_le(y)
True
sage: index_cmp(x, y)
1

```

```

>>> from sage.all import *
>>> from sage.algebras.iwahori_hecke_algebra import index_cmp
>>> W = WeylGroup(['A', Integer(2), Integer(1)])
>>> x = W.from_reduced_word([Integer(0), Integer(1)])
>>> y = W.from_reduced_word([Integer(0), Integer(2), Integer(1)])
>>> x.bruhat_le(y)
True
>>> index_cmp(x, y)
1

```

`sage.algebras.iwahori_hecke_algebra.normalized_laurent_polynomial(R, p)`

Return a normalized version of the (Laurent polynomial) p in the ring R .

Various ring operations in `sage` return an element of the field of fractions of the parent ring even though the element is “known” to belong to the base ring. This function is a hack to recover from this. This occurs somewhat haphazardly with Laurent polynomial rings:

```
sage: R.<q> = LaurentPolynomialRing(ZZ)
sage: [type(c) for c in (q**(-1)).coefficients()]
[<class 'sage.rings.integer.Integer'>]
```

```
>>> from sage.all import *
>>> R = LaurentPolynomialRing(ZZ, names=('q',)); (q,) = R._first_ngens(1)
>>> [type(c) for c in (q**(-Integer(1))).coefficients()]
[<class 'sage.rings.integer.Integer'>]
```

It also happens in any ring when dividing by units:

```
sage: type(3/1)
<class 'sage.rings.rational.Rational'>
sage: type(-1/-1)
<class 'sage.rings.rational.Rational'>
```

```
>>> from sage.all import *
>>> type(Integer(3)/Integer(1))
<class 'sage.rings.rational.Rational'>
>>> type(-Integer(1)/-Integer(1))
<class 'sage.rings.rational.Rational'>
```

This function is a variation on a suggested workaround of Nils Bruin.

EXAMPLES:

```
sage: from sage.algebras.iwahori_hecke_algebra import normalized_laurent_
      ↪ polynomial
sage: type(normalized_laurent_polynomial(ZZ, 3/1))
<class 'sage.rings.integer.Integer'>
sage: R.<q> = LaurentPolynomialRing(ZZ)
sage: [type(c) for c in normalized_laurent_polynomial(R, q**(-1)).coefficients()]
[<class 'sage.rings.integer.Integer'>]
sage: R.<u,v> = LaurentPolynomialRing(ZZ, 2)
sage: p = normalized_laurent_polynomial(R, 2*u**(-1)*v**(-1) + u*v)
sage: ui = normalized_laurent_polynomial(R, u^(-1))
sage: vi = normalized_laurent_polynomial(R, v^(-1))
sage: p(ui, vi)
2*u*v + u^(-1)*v^(-1)
sage: q = u+v+ui
sage: q(ui, vi)
u + v^(-1) + u^(-1)
```

```
>>> from sage.all import *
>>> from sage.algebras.iwahori_hecke_algebra import normalized_laurent_polynomial
>>> type(normalized_laurent_polynomial(ZZ, Integer(3)/Integer(1)))
<class 'sage.rings.integer.Integer'>
>>> R = LaurentPolynomialRing(ZZ, names=('q',)); (q,) = R._first_ngens(1)
>>> [type(c) for c in normalized_laurent_polynomial(R, q**(-Integer(1))).
```

(continues on next page)

(continued from previous page)

```

->coefficients()]
[<class 'sage.rings.integer.Integer'>]
>>> R = LaurentPolynomialRing(ZZ,Integer(2), names=('u', 'v',)); (u, v,) = R._
->first_ngens(2)
>>> p = normalized_laurent_polynomial(R, Integer(2)*u**-Integer(1)*v**-Integer(1)
->+ u*v)
>>> ui = normalized_laurent_polynomial(R, u**-Integer(1))
>>> vi = normalized_laurent_polynomial(R, v**-Integer(1))
>>> p(ui, vi)
2*u*v + u^-1*v^-1
>>> q = u+v+ui
>>> q(ui, vi)
u + v^-1 + u^-1

```

6.4 Nil-Coxeter Algebra

class sage.algebras.nil_coxeter_algebra.**NilCoxeterAlgebra** (*W*, *base_ring*=*Rational Field*,
prefix='u')

Bases: T

Construct the Nil-Coxeter algebra of given type.

This is the algebra with generators u_i for every node i of the corresponding Dynkin diagram. It has the usual braid relations (from the Weyl group) as well as the quadratic relation $u_i^2 = 0$.

INPUT:

- *W* – a Weyl group
- *base_ring* – a ring (default: rational numbers)
- *prefix* – a label for the generators (default: 'u')

EXAMPLES:

```

sage: U = NilCoxeterAlgebra(WeylGroup(['A', 3, 1]))
sage: u0, u1, u2, u3 = U.algebra_generators()
sage: u1*u1
0
sage: u2*u1*u2 == u1*u2*u1
True
sage: U.an_element()
u[0,1,2,3] + 2*u[0] + 3*u[1] + 1

```

```

>>> from sage.all import *
>>> U = NilCoxeterAlgebra(WeylGroup(['A', Integer(3), Integer(1)]))
>>> u0, u1, u2, u3 = U.algebra_generators()
>>> u1*u1
0
>>> u2*u1*u2 == u1*u2*u1
True
>>> U.an_element()
u[0,1,2,3] + 2*u[0] + 3*u[1] + 1

```

homogeneous_generator_noncommutative_variables(*r*)

Give the r -th homogeneous function inside the Nil-Coxeter algebra. In finite type A this is the sum of all decreasing elements of length r . In affine type A this is the sum of all cyclically decreasing elements of length r . This is only defined in finite type A , B and affine types $A^{(1)}$, $B^{(1)}$, $C^{(1)}$, $D^{(1)}$.

INPUT:

- r – positive integer at most the rank of the Weyl group

EXAMPLES:

```
sage: U = NilCoxeterAlgebra(WeylGroup(['A', 3, 1]))
sage: U.homogeneous_generator_noncommutative_variables(2)
u[1,0] + u[2,0] + u[0,3] + u[3,2] + u[3,1] + u[2,1]

sage: U = NilCoxeterAlgebra(WeylGroup(['B', 4]))
sage: U.homogeneous_generator_noncommutative_variables(2)
u[1,2] + u[2,1] + u[3,1] + u[4,1] + u[2,3] + u[3,2] + u[4,2] + u[3,4] + u[4,3]

sage: U = NilCoxeterAlgebra(WeylGroup(['C', 3]))
sage: U.homogeneous_generator_noncommutative_variables(2)
Traceback (most recent call last):
...
AssertionError: Analogue of symmetric functions in noncommutative variables_
↳ is not defined in type ['C', 3]
```

```
>>> from sage.all import *
>>> U = NilCoxeterAlgebra(WeylGroup(['A', Integer(3), Integer(1)]))
>>> U.homogeneous_generator_noncommutative_variables(Integer(2))
u[1,0] + u[2,0] + u[0,3] + u[3,2] + u[3,1] + u[2,1]

>>> U = NilCoxeterAlgebra(WeylGroup(['B', Integer(4)]))
>>> U.homogeneous_generator_noncommutative_variables(Integer(2))
u[1,2] + u[2,1] + u[3,1] + u[4,1] + u[2,3] + u[3,2] + u[4,2] + u[3,4] + u[4,3]

>>> U = NilCoxeterAlgebra(WeylGroup(['C', Integer(3)]))
>>> U.homogeneous_generator_noncommutative_variables(Integer(2))
Traceback (most recent call last):
...
AssertionError: Analogue of symmetric functions in noncommutative variables_
↳ is not defined in type ['C', 3]
```

homogeneous_noncommutative_variables(*la*)

Give the homogeneous function indexed by la , viewed inside the Nil-Coxeter algebra.

This is only defined in finite type A , B and affine types $A^{(1)}$, $B^{(1)}$, $C^{(1)}$, $D^{(1)}$.

INPUT:

- la – a partition with first part bounded by the rank of the Weyl group

EXAMPLES:

```
sage: U = NilCoxeterAlgebra(WeylGroup(['B', 2, 1]))
sage: U.homogeneous_noncommutative_variables([2, 1])
u[1,2,0] + 2*u[2,1,0] + u[0,2,0] + u[0,2,1] + u[1,2,1] + u[2,1,2] + u[2,0,2]_
↳ + u[1,0,2]
```

```
>>> from sage.all import *
>>> U = NilCoxeterAlgebra(WeylGroup(['B', Integer(2), Integer(1)]))
>>> U.homogeneous_noncommutative_variables([Integer(2), Integer(1)])
u[1,2,0] + 2*u[2,1,0] + u[0,2,0] + u[0,2,1] + u[1,2,1] + u[2,1,2] + u[2,0,2]
↪ + u[1,0,2]
```

k_schur_noncommutative_variables (*la*)

In type $A^{(1)}$ this is the k -Schur function in noncommutative variables defined by Thomas Lam [Lam2005].

This function is currently only defined in type $A^{(1)}$.

INPUT:

- *la* – a partition with first part bounded by the rank of the Weyl group

EXAMPLES:

```
sage: A = NilCoxeterAlgebra(WeylGroup(['A', 3, 1]))
sage: A.k_schur_noncommutative_variables([2, 2])
u[0,3,1,0] + u[3,1,2,0] + u[1,2,0,1] + u[3,2,0,3] + u[2,0,3,1] + u[2,3,1,2]
```

```
>>> from sage.all import *
>>> A = NilCoxeterAlgebra(WeylGroup(['A', Integer(3), Integer(1)]))
>>> A.k_schur_noncommutative_variables([Integer(2), Integer(2)])
u[0,3,1,0] + u[3,1,2,0] + u[1,2,0,1] + u[3,2,0,3] + u[2,0,3,1] + u[2,3,1,2]
```

6.5 Yokonuma-Hecke Algebras

AUTHORS:

- Travis Scrimshaw (2015-11): initial version

class sage.algebras.yokonuma_hecke_algebra.YokonumaHeckeAlgebra (*d, n, q, R*)

Bases: CombinatorialFreeModule

The Yokonuma-Hecke algebra $Y_{d,n}(q)$.

Let R be a commutative ring and q be a unit in R . The *Yokonuma-Hecke algebra* $Y_{d,n}(q)$ is the associative, unital R -algebra generated by $t_1, t_2, \dots, t_n, g_1, g_2, \dots, g_{n-1}$ and subject to the relations:

- $g_i g_j = g_j g_i$ for all $|i - j| > 1$,
- $g_i g_{i+1} g_i = g_{i+1} g_i g_{i+1}$,
- $t_i t_j = t_j t_i$,
- $t_j g_i = g_i t_{j s_i}$, and
- $t_j^d = 1$,

where s_i is the simple transposition $(i, i + 1)$, along with the quadratic relation

$$g_i^2 = 1 + \frac{(q - q^{-1})}{d} \left(\sum_{s=0}^{d-1} t_i^s t_{i+1}^{-s} \right) g_i.$$

Thus the Yokonuma-Hecke algebra can be considered a quotient of the framed braid group $(\mathbf{Z}/d\mathbf{Z}) \wr B_n$, where B_n is the classical braid group on n strands, by the quadratic relations. Moreover, all of the algebra generators are

invertible. In particular, we have

$$g_i^{-1} = g_i - (q - q^{-1})e_i.$$

When we specialize $q = \pm 1$, we obtain the group algebra of the complex reflection group $G(d, 1, n) = (\mathbf{Z}/d\mathbf{Z}) \wr S_n$. Moreover for $d = 1$, the Yokonuma-Hecke algebra is equal to the *Iwahori-Hecke* of type A_{n-1} .

INPUT:

- d – the maximum power of t
- n – the number of generators
- q – (optional) an invertible element in a commutative ring; the default is $q \in \mathbf{Q}[q, q^{-1}]$
- R – (optional) a commutative ring containing q ; the default is the parent of q

EXAMPLES:

We construct $Y_{4,3}$ and do some computations:

```
sage: Y = algebras.YokonumaHecke(4, 3)
sage: g1, g2, t1, t2, t3 = Y.algebra_generators()
sage: g1 * g2
g[1,2]
sage: t1 * g1
t1*g[1]
sage: g2 * t2
t3*g[2]
sage: g2 * t3
t2*g[2]
sage: (g2 + t1) * (g1 + t2*t3)
g[2,1] + t2*t3*g[2] + t1*g[1] + t1*t2*t3
sage: g1 * g1
1 - (1/4*q^-1-1/4*q)*g[1] - (1/4*q^-1-1/4*q)*t1*t2^3*g[1]
- (1/4*q^-1-1/4*q)*t1^2*t2^2*g[1] - (1/4*q^-1-1/4*q)*t1^3*t2*g[1]
sage: g2 * g1 * t1
t3*g[2,1]
```

```
>>> from sage.all import *
>>> Y = algebras.YokonumaHecke(Integer(4), Integer(3))
>>> g1, g2, t1, t2, t3 = Y.algebra_generators()
>>> g1 * g2
g[1,2]
>>> t1 * g1
t1*g[1]
>>> g2 * t2
t3*g[2]
>>> g2 * t3
t2*g[2]
>>> (g2 + t1) * (g1 + t2*t3)
g[2,1] + t2*t3*g[2] + t1*g[1] + t1*t2*t3
>>> g1 * g1
1 - (1/4*q^-1-1/4*q)*g[1] - (1/4*q^-1-1/4*q)*t1*t2^3*g[1]
- (1/4*q^-1-1/4*q)*t1^2*t2^2*g[1] - (1/4*q^-1-1/4*q)*t1^3*t2*g[1]
>>> g2 * g1 * t1
t3*g[2,1]
```

We construct the elements e_i and show that they are idempotents:

```
sage: e1 = Y.e(1); e1
1/4 + 1/4*t1*t2^3 + 1/4*t1^2*t2^2 + 1/4*t1^3*t2
sage: e1 * e1 == e1
True
sage: e2 = Y.e(2); e2
1/4 + 1/4*t2*t3^3 + 1/4*t2^2*t3^2 + 1/4*t2^3*t3
sage: e2 * e2 == e2
True
```

```
>>> from sage.all import *
>>> e1 = Y.e(Integer(1)); e1
1/4 + 1/4*t1*t2^3 + 1/4*t1^2*t2^2 + 1/4*t1^3*t2
>>> e1 * e1 == e1
True
>>> e2 = Y.e(Integer(2)); e2
1/4 + 1/4*t2*t3^3 + 1/4*t2^2*t3^2 + 1/4*t2^3*t3
>>> e2 * e2 == e2
True
```

REFERENCES:

- [CL2013]
- [CPdA2014]
- [ERH2015]
- [JPdA15]

class Element

Bases: `IndexedFreeModuleElement`

algebra_generators()

Return the algebra generators of `self`.

EXAMPLES:

```
sage: Y = algebras.YokonumaHecke(5, 3)
sage: dict(Y.algebra_generators())
{'g1': g[1], 'g2': g[2], 't1': t1, 't2': t2, 't3': t3}
```

```
>>> from sage.all import *
>>> Y = algebras.YokonumaHecke(Integer(5), Integer(3))
>>> dict(Y.algebra_generators())
{'g1': g[1], 'g2': g[2], 't1': t1, 't2': t2, 't3': t3}
```

$e(i)$

Return the element e_i .

EXAMPLES:

```
sage: Y = algebras.YokonumaHecke(4, 3)
sage: Y.e(1)
1/4 + 1/4*t1*t2^3 + 1/4*t1^2*t2^2 + 1/4*t1^3*t2
```

(continues on next page)

(continued from previous page)

```
sage: Y.e(2)
1/4 + 1/4*t2*t3^3 + 1/4*t2^2*t3^2 + 1/4*t2^3*t3
```

```
>>> from sage.all import *
>>> Y = algebras.YokonumaHecke(Integer(4), Integer(3))
>>> Y.e(Integer(1))
1/4 + 1/4*t1*t2^3 + 1/4*t1^2*t2^2 + 1/4*t1^3*t2
>>> Y.e(Integer(2))
1/4 + 1/4*t2*t3^3 + 1/4*t2^2*t3^2 + 1/4*t2^3*t3
```

g (*i=None*)Return the generator(s) g_i .

INPUT:

- *i* – (default: None) the generator g_i or if None, then the list of all generators g_i

EXAMPLES:

```
sage: Y = algebras.YokonumaHecke(8, 3)
sage: Y.g(1)
g[1]
sage: Y.g()
[g[1], g[2]]
```

```
>>> from sage.all import *
>>> Y = algebras.YokonumaHecke(Integer(8), Integer(3))
>>> Y.g(Integer(1))
g[1]
>>> Y.g()
[g[1], g[2]]
```

gens ()

Return the generators of self.

EXAMPLES:

```
sage: Y = algebras.YokonumaHecke(5, 3)
sage: Y.gens()
(g[1], g[2], t1, t2, t3)
```

```
>>> from sage.all import *
>>> Y = algebras.YokonumaHecke(Integer(5), Integer(3))
>>> Y.gens()
(g[1], g[2], t1, t2, t3)
```

inverse_g (*i*)Return the inverse of the generator g_i .

From the quadratic relation, we have

$$g_i^{-1} = g_i - (q - q^{-1})e_i.$$

EXAMPLES:


```

sage: Y = algebras.YokonumaHecke(2, 4)
sage: [2*Y.inverse_g(i) for i in range(1, 4)]
[(q^-1-q) + 2*g[1] + (q^-1-q)*t1*t2,
 (q^-1-q) + 2*g[2] + (q^-1-q)*t2*t3,
 (q^-1-q) + 2*g[3] + (q^-1-q)*t3*t4]
sage: all(Y.inverse_g(i) * Y.g(i) == Y.one() for i in range(1, 4))
True
sage: all(Y.g(i) * Y.inverse_g(i) == Y.one() for i in range(1, 4))
True

```

```

>>> from sage.all import *
>>> Y = algebras.YokonumaHecke(Integer(2), Integer(4))
>>> [Integer(2)*Y.inverse_g(i) for i in range(Integer(1), Integer(4))]
[(q^-1-q) + 2*g[1] + (q^-1-q)*t1*t2,
 (q^-1-q) + 2*g[2] + (q^-1-q)*t2*t3,
 (q^-1-q) + 2*g[3] + (q^-1-q)*t3*t4]
>>> all(Y.inverse_g(i) * Y.g(i) == Y.one() for i in range(Integer(1),
↳Integer(4)))
True
>>> all(Y.g(i) * Y.inverse_g(i) == Y.one() for i in range(Integer(1),
↳Integer(4)))
True

```

one_basis()

Return the index of the basis element of 1.

EXAMPLES:

```

sage: Y = algebras.YokonumaHecke(5, 3)
sage: Y.one_basis()
((0, 0, 0), [1, 2, 3])

```

```

>>> from sage.all import *
>>> Y = algebras.YokonumaHecke(Integer(5), Integer(3))
>>> Y.one_basis()
((0, 0, 0), [1, 2, 3])

```

product_on_basis(m1, m2)

Return the product of the basis elements indexed by m1 and m2.

EXAMPLES:

```

sage: Y = algebras.YokonumaHecke(4, 3)
sage: m = ((1, 0, 2), Permutations(3)([2, 1, 3]))
sage: 4 * Y.product_on_basis(m, m)
-(q^-1-q)*t2^2*g[1] + 4*t1*t2 - (q^-1-q)*t1*t2*g[1]
- (q^-1-q)*t1^2*g[1] - (q^-1-q)*t1^3*t2^3*g[1]

```

```

>>> from sage.all import *
>>> Y = algebras.YokonumaHecke(Integer(4), Integer(3))
>>> m = ((Integer(1), Integer(0), Integer(2)),
↳Permutations(Integer(3))([Integer(2), Integer(1), Integer(3)]))
>>> Integer(4) * Y.product_on_basis(m, m)

```

(continues on next page)

(continued from previous page)

```

-(q^-1-q)*t2^2*g[1] + 4*t1*t2 - (q^-1-q)*t1*t2*g[1]
- (q^-1-q)*t1^2*g[1] - (q^-1-q)*t1^3*t2^3*g[1]

```

Check that we apply the permutation correctly on t_i :

```

sage: Y = algebras.YokonumaHecke(4, 3)
sage: g1, g2, t1, t2, t3 = Y.algebra_generators()
sage: g21 = g2 * g1
sage: g21 * t1
t3*g[2,1]

```

```

>>> from sage.all import *
>>> Y = algebras.YokonumaHecke(Integer(4), Integer(3))
>>> g1, g2, t1, t2, t3 = Y.algebra_generators()
>>> g21 = g2 * g1
>>> g21 * t1
t3*g[2,1]

```

t (*i*=None)

Return the generator(s) t_i .

INPUT:

- *i* – (default: None) the generator t_i or if None, then the list of all generators t_i

EXAMPLES:

```

sage: Y = algebras.YokonumaHecke(8, 3)
sage: Y.t(2)
t2
sage: Y.t()
[t1, t2, t3]

```

```

>>> from sage.all import *
>>> Y = algebras.YokonumaHecke(Integer(8), Integer(3))
>>> Y.t(Integer(2))
t2
>>> Y.t()
[t1, t2, t3]

```

6.6 Cubic Hecke Algebras

6.6.1 Cubic Hecke Algebras

We consider the factors of the group algebra of the Artin braid groups such that the images s_i of the braid generators satisfy a cubic equation:

$$s_i^3 = us_i^2 - vs_i + w.$$

Here u, v, w are elements in an arbitrary integral domain and i is a positive integer less than n , the number of the braid group's strands. By the analogue to the *Iwahori Hecke algebras* (see [IwahoriHeckeAlgebra](#)), in which the braid generators satisfy a quadratic relation these algebras have been called *cubic Hecke algebras*. The relations inherited from

the braid group are:

$$s_i s_{i+1} s_i = s_{i+1} s_i s_{i+1} \text{ for } 1 \leq i < n-1 \text{ and } s_i s_j = s_j s_i \text{ for } 1 \leq i < j-1 < n-1.$$

The algebra epimorphism from the braid group algebra over the same base ring is realized inside the element constructor of the present class, for example in the case of the 3 strand cubic Hecke algebra:

```
sage: CHA3 = algebras.CubicHecke(3)
sage: BG3 = CHA3.braid_group()
sage: braid = BG3((1,2,-1,2,2,-1)); braid
c0*c1*c0^-1*c1^2*c0^-1
sage: braid_image = CHA3(braid); braid_image
u*c1*c0^-1*c1 + u*v*c0*c1^-1*c0^-1 - u^2*c0^-1*c1
+ ((u^2*v-v^2)/w)*c0*c1*c0^-1 + ((u^2-v)/w)*c0*c1*c0
- ((u^3-u*v)/w)*c0*c1 - (u*v-w)*c1^-1
```

```
>>> from sage.all import *
>>> CHA3 = algebras.CubicHecke(Integer(3))
>>> BG3 = CHA3.braid_group()
>>> braid = BG3((Integer(1), Integer(2), -Integer(1), Integer(2), Integer(2), -
↳ Integer(1))); braid
c0*c1*c0^-1*c1^2*c0^-1
>>> braid_image = CHA3(braid); braid_image
u*c1*c0^-1*c1 + u*v*c0*c1^-1*c0^-1 - u^2*c0^-1*c1
+ ((u^2*v-v^2)/w)*c0*c1*c0^-1 + ((u^2-v)/w)*c0*c1*c0
- ((u^3-u*v)/w)*c0*c1 - (u*v-w)*c1^-1
```

If the ring elements u, v, w (which will be called the *cubic equation parameters* in the sequel) are taken to be $u = v = 0, w = 1$ the cubic Hecke algebra specializes to the group algebra of the *cubic braid group*, which is the factor group of the Artin braid group under setting the generators order to be three. These groups can be obtained by `CubicHeckeAlgebra.cubic_braid_group()`.

It is well known, that these algebras are free of finite rank as long as the number of braid generators is less than six and infinite dimensional else wise. In the former (non trivial) cases they are also known as *cyclotomic Hecke algebras* corresponding to the complex reflection groups having Shepard-Todd number 4, 25 and 32.

Since the *Broué, Malle, Rouquier* conjecture has been proved (for references of these cases see [Mar2012]) there exists a finite free basis of the cubic Hecke algebra which is in bijection to the cubic braid group and compatible with the specialization to the cubic braid group algebra as explained above.

For the algebras corresponding to braid groups of less than five strands such a basis has been calculated by Ivan Marin. This one is used here. In the case of 5 strands such a basis is not available, right now. Instead the elements of the cubic braid group class themselves are used as basis elements. This is also the case when the cubic braid group is infinite, even though it is not known if these elements span all of the cubic Hecke algebra.

Accordingly, be aware that the module embedding of the group algebra of the cubic braid groups is known to be an isomorphism of free modules only in the cases of less than five strands.

EXAMPLES:

Consider the obstruction b of the *triple quadratic algebra* from Section 2.6 of [Mar2018]. We verify that the third power of it is a scalar multiple of itself (explicitly $2*w^2$ times the *Schur element* of the three dimensional irreducible representation):

```
sage: CHA3 = algebras.CubicHecke(3)
sage: c1, c2 = CHA3.gens()
sage: b = c1^2*c2 - c2*c1^2 - c1*c2^2 + c2^2*c1; b
```

(continues on next page)

(continued from previous page)

```

w*c0^-1*c1 - w*c0*c1^-1 - w*c1*c0^-1 + w*c1^-1*c0
sage: b2 = b*b
sage: b3 = b2*b
sage: BR = CHA3.base_ring()
sage: ER = CHA3.extension_ring()
sage: u, v, w = BR.gens()
sage: f = BR(b3.coefficients()[0]/w)
sage: try:
.....: sh = CHA3.schur_element(CH A3.irred_repr.W3_111)
.....: except NotImplementedError: # for the case GAP3 / CHEVIE not available
.....: sh = ER(f/(2*w^2))
sage: ER(f/(2*w^2)) == sh
True
sage: b3 == f*b
True

```

```

>>> from sage.all import *
>>> CHA3 = algebras.CubicHecke(Integer(3))
>>> c1, c2 = CHA3.gens()
>>> b = c1**Integer(2)*c2 - c2*c1**Integer(2) - c1*c2**Integer(2) + c2**Integer(2)*c1;
↪ b
w*c0^-1*c1 - w*c0*c1^-1 - w*c1*c0^-1 + w*c1^-1*c0
>>> b2 = b*b
>>> b3 = b2*b
>>> BR = CHA3.base_ring()
>>> ER = CHA3.extension_ring()
>>> u, v, w = BR.gens()
>>> f = BR(b3.coefficients()[Integer(0)]/w)
>>> try:
... sh = CHA3.schur_element(CH A3.irred_repr.W3_111)
... except NotImplementedError: # for the case GAP3 / CHEVIE not available
... sh = ER(f/(Integer(2)*w**Integer(2)))
>>> ER(f/(Integer(2)*w**Integer(2))) == sh
True
>>> b3 == f*b
True

```

Defining the cubic Hecke algebra on 6 strands will need some seconds for initializing. However, you can do calculations inside the infinite algebra as well:

```

sage: # optional - database_cubic_hecke
sage: CHA6 = algebras.CubicHecke(6)
sage: CHA6.inject_variables()
Defining c0, c1, c2, c3, c4
sage: s = c0*c1*c2*c3*c4; s
c0*c1*c2*c3*c4
sage: s^2
(c0*c1*c2*c3*c4)^2
sage: t = CHA6.an_element() * c4; t
(-w)*c0*c1^-1*c4 + v*c0*c2^-1*c4 + u*c2*c1*c4 + ((-v*w+u)/w)*c4

```

```

>>> from sage.all import *
>>> # optional - database_cubic_hecke
>>> CHA6 = algebras.CubicHecke(Integer(6))
>>> CHA6.inject_variables()
Defining c0, c1, c2, c3, c4
>>> s = c0*c1*c2*c3*c4; s
c0*c1*c2*c3*c4
>>> s**Integer(2)
(c0*c1*c2*c3*c4)^2
>>> t = CHA6.an_element() * c4; t
(-w)*c0*c1^-1*c4 + v*c0*c2^-1*c4 + u*c2*c1*c4 + ((-v*w+u)/w)*c4

```

REFERENCES:

- [Mar2012]
- [Mar2018]
- [CM2012]

AUTHORS:

- Sebastian Oehms May 2020: initial version

```

class sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra(names, cu-
                                                                    bic_equation_pa-
                                                                    rameters=None,
                                                                    cubic_equa-
                                                                    tion_roots=None)

```

Bases: `CombinatorialFreeModule`

Return the Cubic-Hecke algebra with respect to the Artin braid group on n strands.

This is a quotient of the group algebra of the Artin braid group, such that the images s_i ($1 \leq i < n$) of the braid generators satisfy a cubic equation (see [cubic_hecke_algebra](#) for more information, in a session type `sage.algebras.hecke_algebras.cubic_hecke_algebra?`):

$$s_i^3 = us_i^2 - vs_i + w.$$

The base ring of this algebra can be specified by giving optional keywords described below. If no keywords are given, the base ring will be a `CubicHeckeRingOfDefinition`, which is constructed as the polynomial ring in u, v, w over the integers localized at w . This ring will be called the *ring of definition* or sometimes for short *generic base ring*. However note, that in this context the word *generic* should not remind in a generic point of the corresponding scheme.

In addition to the base ring, another ring containing the roots (a , b and c) of the cubic equation will be needed to handle the split irreducible representations. This ring will be called the *extension ring*. Generically, the extension ring will be a `CubicHeckeExtensionRing`, which is constructed as the Laurent polynomial ring in a, b and c over the integers adjoined with a primitive third root of unity. A special form of this *generic extension ring* is constructed as a `SplittingAlgebra` for the roots of the cubic equation and a primitive third root of unity over the ring of definition. This ring will be called the *default extension ring*.

This class uses a static and a dynamic data library. The first one is defined as instance of `CubicHeckeDataBase` and contains the complete basis for the algebras with less than 5 strands and various types of representation matrices of the generators. These data have been calculated by [Ivan Marin](#) and have been imported from his corresponding [web page](#).

Note that just the data for the cubic Hecke algebras on less than four strands is available in Sage by default. To deal with four strands and more you need to install the optional package `database_cubic_hecke` by typing

- `sage -i database_cubic_hecke` (first time installation) or
- `sage -f database_cubic_hecke` (reinstallation) respective
- `sage -i -c database_cubic_hecke` (for running all test in concern)
- `sage -f -c database_cubic_hecke`

This will add a [Python wrapper](#) around Ivan Marin's data to the Sage library. For more installation hints see the documentation of this wrapper.

Furthermore, representation matrices can be obtained from the CHEVIE package of GAP3 via the GAP3 interface if GAP3 is installed inside Sage. For more information on how to obtain representation matrices to elements of this class, see the documentation of the element class [CubicHeckeElement](#) or its method `matrix()`:

```
algebras.CubicHecke.Element? or algebras.CubicHecke.Element.matrix?
```

The second library is created as instance of [CubicHeckeFileCache](#) and used while working with the class to achieve a better performance. This file cache contains images of braids and representation matrices of basis elements from former calculations. A refresh of the file cache can be done using the `reset_filecache()`.

INPUT:

- `names` – string containing the names of the generators as images of the braid group generators
- `cubic_equation_parameters` – tuple (u, v, w) of three elements in an integral domain used as coefficients in the cubic equation. If this argument is given the base ring will be set to the common parent of u, v, w . In addition a conversion map from the generic base ring is supplied. This keyword can also be used to change the variable names of the generic base ring (see example 3 below)
- `cubic_equation_roots` – tuple (a, b, c) of three elements in an integral domain which stand for the roots of the cubic equation. If this argument is given the extension ring will be set to the common parent of a, b, c . In addition a conversion map from the generic extension ring and the generic base ring is supplied. This keyword can also be used to change the variable names of the generic extension ring (see example 3 below)

EXAMPLES:

Cubic Hecke algebra over the ring of definition:

```
sage: CHA3 = algebras.CubicHecke('s1, s2'); CHA3
Cubic Hecke algebra on 3 strands over Multivariate Polynomial Ring
in u, v, w
over Integer Ring localized at (w,)
with cubic equation: h^3 - u*h^2 + v*h - w = 0
sage: CHA3.gens()
(s1, s2)
sage: GER = CHA3.extension_ring(generic=True); GER
Multivariate Laurent Polynomial Ring in a, b, c
over Splitting Algebra of x^2 + x + 1
with roots [e3, -e3 - 1] over Integer Ring
sage: ER = CHA3.extension_ring(); ER
Splitting Algebra of T^2 + T + 1 with roots [E3, -E3 - 1]
over Splitting Algebra of h^3 - u*h^2 + v*h - w
with roots [a, b, -b - a + u]
over Multivariate Polynomial Ring in u, v, w
over Integer Ring localized at (w,)
```

```
>>> from sage.all import *
>>> CHA3 = algebras.CubicHecke('s1, s2'); CHA3
```

(continues on next page)

(continued from previous page)

```

Cubic Hecke algebra on 3 strands over Multivariate Polynomial Ring
  in u, v, w
  over Integer Ring localized at (w,)
  with cubic equation: h^3 - u*h^2 + v*h - w = 0
>>> CHA3.gens()
(s1, s2)
>>> GER = CHA3.extension_ring(generic=True); GER
Multivariate Laurent Polynomial Ring in a, b, c
  over Splitting Algebra of x^2 + x + 1
  with roots [e3, -e3 - 1] over Integer Ring
>>> ER = CHA3.extension_ring(); ER
Splitting Algebra of T^2 + T + 1 with roots [E3, -E3 - 1]
  over Splitting Algebra of h^3 - u*h^2 + v*h - w
  with roots [a, b, -b - a + u]
  over Multivariate Polynomial Ring in u, v, w
  over Integer Ring localized at (w,)

```

Element construction:

```

sage: ele = CHA3.an_element(); ele
-w*s1*s2^-1 + v*s1 + u*s2 - ((v*w-u)/w)
sage: ele2 = ele**2; ele2
w^2*(s1^-1*s2)^2 - u*w^2*s1^-1*s2*s1^-1 - v*w*s2*s1^-1*s2
- v*w^2*s1^-1*s2^-1 + u*w*s1*s2*s1^-1*s2 - u*w*s1^-1*s2*s1
- (u*v*w-2*v*w+2*u)*s1*s2^-1 + u*v*w*s2*s1^-1 + u*v*s2*s1 + v^2*w*s1^-1
- u^2*w*s1*s2*s1^-1 + ((u*v^2*w-2*v^2*w-u*w^2+2*u*v)/w)*s1 + u*v*s1*s2
+ (u^2*w+v^2*w)*s2^-1 + ((u^3*w-2*u*v*w+2*u^2)/w)*s2
- ((u^2*v*w^2+v^3*w^2-v^2*w^2+2*u*v*w-u^2)/w^2)
sage: B3 = CHA3.braid_group()
sage: braid = B3((2,-1, 2, 1)); braid
s2*s1^-1*s2*s1
sage: ele3 = CHA3(braid); ele3
s1*s2*s1^-1*s2 + u*s1^-1*s2*s1 - v*s1*s2^-1 + v*s2^-1*s1 - u*s1*s2*s1^-1
sage: ele3t = CHA3((2,-1, 2, 1))
sage: ele3 == ele3t
True
sage: CHA4 = algebras.CubicHecke(4) # optional database_cubic_hecke
sage: ele4 = CHA4(ele3); ele4 # optional database_cubic_hecke
c0*c1*c0^-1*c1 + u*c0^-1*c1*c0 + (-v)*c0*c1^-1 + v*c1^-1*c0 + (-u)*c0*c1*c0^-1

```

```

>>> from sage.all import *
>>> ele = CHA3.an_element(); ele
-w*s1*s2^-1 + v*s1 + u*s2 - ((v*w-u)/w)
>>> ele2 = ele**Integer(2); ele2
w^2*(s1^-1*s2)^2 - u*w^2*s1^-1*s2*s1^-1 - v*w*s2*s1^-1*s2
- v*w^2*s1^-1*s2^-1 + u*w*s1*s2*s1^-1*s2 - u*w*s1^-1*s2*s1
- (u*v*w-2*v*w+2*u)*s1*s2^-1 + u*v*w*s2*s1^-1 + u*v*s2*s1 + v^2*w*s1^-1
- u^2*w*s1*s2*s1^-1 + ((u*v^2*w-2*v^2*w-u*w^2+2*u*v)/w)*s1 + u*v*s1*s2
+ (u^2*w+v^2*w)*s2^-1 + ((u^3*w-2*u*v*w+2*u^2)/w)*s2
- ((u^2*v*w^2+v^3*w^2-v^2*w^2+2*u*v*w-u^2)/w^2)
>>> B3 = CHA3.braid_group()
>>> braid = B3((Integer(2),-Integer(1), Integer(2), Integer(1))); braid

```

(continues on next page)

(continued from previous page)

```

s2*s1^-1*s2*s1
>>> ele3 = CHA3(braid); ele3
s1*s2*s1^-1*s2 + u*s1^-1*s2*s1 - v*s1*s2^-1 + v*s2^-1*s1 - u*s1*s2*s1^-1
>>> ele3t = CHA3((Integer(2), -Integer(1), Integer(2), Integer(1)))
>>> ele3 == ele3t
True
>>> CHA4 = algebras.CubicHecke(Integer(4))      # optional database_cubic_hecke
>>> ele4 = CHA4(ele3); ele4                      # optional database_cubic_hecke
c0*c1*c0^-1*c1 + u*c0^-1*c1*c0 + (-v)*c0*c1^-1 + v*c1^-1*c0 + (-u)*c0*c1*c0^-1

```

Cubic Hecke algebra over the ring of definition using different variable names:

```

sage: algebras.CubicHecke(3, cubic_equation_parameters='u, v, w', cubic_equation_
↪roots='p, q, r')
Cubic Hecke algebra on 3 strands over Multivariate Polynomial Ring
in u, v, w
over Integer Ring localized at (w,)
with cubic equation: h^3 - u*h^2 + v*h - w = 0
sage: _.extension_ring()
Splitting Algebra of T^2 + T + 1 with roots [E3, -E3 - 1]
over Splitting Algebra of h^3 - u*h^2 + v*h - w
with roots [p, q, -q - p + u]
over Multivariate Polynomial Ring in u, v, w
over Integer Ring localized at (w,)

```

```

>>> from sage.all import *
>>> algebras.CubicHecke(Integer(3), cubic_equation_parameters='u, v, w', cubic_
↪equation_roots='p, q, r')
Cubic Hecke algebra on 3 strands over Multivariate Polynomial Ring
in u, v, w
over Integer Ring localized at (w,)
with cubic equation: h^3 - u*h^2 + v*h - w = 0
>>> _.extension_ring()
Splitting Algebra of T^2 + T + 1 with roots [E3, -E3 - 1]
over Splitting Algebra of h^3 - u*h^2 + v*h - w
with roots [p, q, -q - p + u]
over Multivariate Polynomial Ring in u, v, w
over Integer Ring localized at (w,)

```

Cubic Hecke algebra over a special base ring with respect to a special cubic equation:

```

sage: algebras.CubicHecke('s1, s2', cubic_equation_parameters=(QQ(1), 3, 1))
Cubic Hecke algebra on 3 strands over Rational Field
with cubic equation: h^3 - h^2 + 3*h - 1 = 0
sage: CHA3 = _
sage: ER = CHA3.extension_ring(); ER
Number Field in T with defining polynomial T^12 + 4*T^11 + 51*T^10
+ 154*T^9 + 855*T^8 + 1880*T^7 + 5805*T^6 + 8798*T^5 + 15312*T^4
+ 14212*T^3 + 13224*T^2 + 5776*T + 1444
sage: CHA3.cubic_equation_roots()[0]
-4321/1337904*T^11 - 4181/445968*T^10 - 4064/27873*T^9 - 51725/167238*T^8
- 2693189/1337904*T^7 - 1272907/445968*T^6 - 704251/74328*T^5

```

(continues on next page)

(continued from previous page)

```

- 591488/83619*T^4 - 642145/83619*T^3 + 252521/111492*T^2 + 45685/5868*T
+ 55187/17604

sage: F = GF(25, 'u')
sage: algebras.CubicHecke('s1, s2', cubic_equation_parameters=(F(1), F.gen(), u
↪F(3)))
Cubic Hecke algebra on 3 strands over Finite Field in u of size 5^2
  with cubic equation: h^3 + 4*h^2 + u*h + 2 = 0
sage: CHA3 = _
sage: ER = CHA3.extension_ring(); ER
Finite Field in S of size 5^4
sage: CHA3.cubic_equation_roots()
[2*S^3 + 2*S^2 + 2*S + 1, 2*S^3 + 3*S^2 + 3*S + 2, S^3 + 3]

```

```

>>> from sage.all import *
>>> algebras.CubicHecke('s1, s2', cubic_equation_parameters=(QQ(Integer(1)),
↪Integer(3), Integer(1)))
Cubic Hecke algebra on 3 strands over Rational Field
  with cubic equation: h^3 - h^2 + 3*h - 1 = 0
>>> CHA3 = _
>>> ER = CHA3.extension_ring(); ER
Number Field in T with defining polynomial T^12 + 4*T^11 + 51*T^10
+ 154*T^9 + 855*T^8 + 1880*T^7 + 5805*T^6 + 8798*T^5 + 15312*T^4
+ 14212*T^3 + 13224*T^2 + 5776*T + 1444
>>> CHA3.cubic_equation_roots()[Integer(0)]
-4321/1337904*T^11 - 4181/445968*T^10 - 4064/27873*T^9 - 51725/167238*T^8
- 2693189/1337904*T^7 - 1272907/445968*T^6 - 704251/74328*T^5
- 591488/83619*T^4 - 642145/83619*T^3 + 252521/111492*T^2 + 45685/5868*T
+ 55187/17604

>>> F = GF(Integer(25), 'u')
>>> algebras.CubicHecke('s1, s2', cubic_equation_parameters=(F(Integer(1)), F.
↪gen(), F(Integer(3))))
Cubic Hecke algebra on 3 strands over Finite Field in u of size 5^2
  with cubic equation: h^3 + 4*h^2 + u*h + 2 = 0
>>> CHA3 = _
>>> ER = CHA3.extension_ring(); ER
Finite Field in S of size 5^4
>>> CHA3.cubic_equation_roots()
[2*S^3 + 2*S^2 + 2*S + 1, 2*S^3 + 3*S^2 + 3*S + 2, S^3 + 3]

```

Cubic Hecke algebra over a special extension ring with respect to special roots of the cubic equation:

```

sage: UCF = UniversalCyclotomicField()
sage: e3=UCF.gen(3); e5=UCF.gen(5)
sage: algebras.CubicHecke('s1, s2', cubic_equation_roots=(1, e5, e3))
Cubic Hecke algebra on 3 strands over Universal Cyclotomic Field
  with cubic equation:
h^3 + (-E(15) - E(15)^4 - E(15)^7 + E(15)^8)*h^2 + (-E(15)^2 - E(15)^8
- E(15)^11 - E(15)^13 - E(15)^14)*h - E(15)^8 = 0

```

```

>>> from sage.all import *
>>> UCF = UniversalCyclotomicField()
>>> e3=UCF.gen(Integer(3)); e5=UCF.gen(Integer(5))
>>> algebras.CubicHecke('s1, s2', cubic_equation_roots=(Integer(1), e5, e3))
Cubic Hecke algebra on 3 strands over Universal Cyclotomic Field
with cubic equation:
h^3 + (-E(15) - E(15)^4 - E(15)^7 + E(15)^8)*h^2 + (-E(15)^2 - E(15)^8
- E(15)^11 - E(15)^13 - E(15)^14)*h - E(15)^8 = 0

```

Element

alias of *CubicHeckeElement*

algebra_generators()

Return the algebra generators of self.

EXAMPLES:

```

sage: CHA2 = algebras.CubicHecke(2)
sage: CHA2.algebra_generators()
Finite family {c: c}

```

```

>>> from sage.all import *
>>> CHA2 = algebras.CubicHecke(Integer(2))
>>> CHA2.algebra_generators()
Finite family {c: c}

```

base_ring (generic=False)

Return the base ring of self.

INPUT:

- **generic** – boolean (default: False); if True the ring of definition (here often called the generic base ring) is returned

EXAMPLES:

```

sage: CHA2 = algebras.CubicHecke(2, cubic_equation_roots=(3, 4, 5))
sage: CHA2.base_ring()
Integer Ring localized at (2, 3, 5)
sage: CHA2.base_ring(generic=True)
Multivariate Polynomial Ring in u, v, w
over Integer Ring localized at (w,)

```

```

>>> from sage.all import *
>>> CHA2 = algebras.CubicHecke(Integer(2), cubic_equation_roots=(Integer(3), Integer(4), Integer(5)))
>>> CHA2.base_ring()
Integer Ring localized at (2, 3, 5)
>>> CHA2.base_ring(generic=True)
Multivariate Polynomial Ring in u, v, w
over Integer Ring localized at (w,)

```

braid_group()

Return the braid group attached to self.

EXAMPLES:

```
sage: CHA2 = algebras.CubicHecke(2)
sage: CHA2.braid_group()
Braid group on 2 strands
```

```
>>> from sage.all import *
>>> CHA2 = algebras.CubicHecke(Integer(2))
>>> CHA2.braid_group()
Braid group on 2 strands
```

braid_group_algebra()

Return the group algebra of braid group attached to *self* over the base ring of *self*.

EXAMPLES:

```
sage: CHA2 = algebras.CubicHecke(2)
sage: CHA2.braid_group_algebra()
Algebra of Braid group on 2 strands
over Multivariate Polynomial Ring in u, v, w
over Integer Ring localized at (w,)
```

```
>>> from sage.all import *
>>> CHA2 = algebras.CubicHecke(Integer(2))
>>> CHA2.braid_group_algebra()
Algebra of Braid group on 2 strands
over Multivariate Polynomial Ring in u, v, w
over Integer Ring localized at (w,)
```

characters (irr=None, original=True)

Return the irreducible characters of *self*.

By default the values are given in the generic extension ring. Setting the keyword *original* to *False* you can obtain the values in the (non generic) extension ring (compare the same keyword for *CubicHeckeElement.matrix()*).

INPUT:

- *irr* – (optional) instance of *AbsIrreducibleRep* selecting the irreducible representation corresponding to the character; if not given a list of all characters is returned
- *original* – boolean (default: *True*); see description above

OUTPUT:

Function or list of Functions from the element class of *self* to the (generic or non generic) extension ring depending on the given keyword arguments.

EXAMPLES:

```
sage: CHA3 = algebras.CubicHecke(3)
sage: ch = CHA3.characters()
sage: e = CHA3.an_element()
sage: ch[0](e)
a^2*b + a^2*c + a^2 - b*c + b^-1*c^-1 + a^-1*c^-1 + a^-1*b^-1
sage: _.parent()
Multivariate Laurent Polynomial Ring in a, b, c
over Splitting Algebra of x^2 + x + 1 with roots [e3, -e3 - 1]
```

(continues on next page)

(continued from previous page)

```

over Integer Ring
sage: ch_w3_100 = CHA3.characters(irr=CHA3.irred_repr.W3_100)
sage: ch_w3_100(e) == ch[0](e)
True
sage: ch_x = CHA3.characters(original=False)
sage: ch_x[0](e)
(u + v)*a + (-v*w - w^2 + u)/w
sage: _.parent()
Splitting Algebra of T^2 + T + 1 with roots [E3, -E3 - 1]
over Splitting Algebra of h^3 - u*h^2 + v*h - w
with roots [a, b, -b - a + u]
over Multivariate Polynomial Ring in u, v, w
over Integer Ring localized at (w,)

```

```

>>> from sage.all import *
>>> CHA3 = algebras.CubicHecke(Integer(3))
>>> ch = CHA3.characters()
>>> e = CHA3.an_element()
>>> ch[Integer(0)](e)
a^2*b + a^2*c + a^2 - b*c + b^-1*c^-1 + a^-1*c^-1 + a^-1*b^-1
>>> _.parent()
Multivariate Laurent Polynomial Ring in a, b, c
over Splitting Algebra of x^2 + x + 1 with roots [e3, -e3 - 1]
over Integer Ring
>>> ch_w3_100 = CHA3.characters(irr=CHA3.irred_repr.W3_100)
>>> ch_w3_100(e) == ch[Integer(0)](e)
True
>>> ch_x = CHA3.characters(original=False)
>>> ch_x[Integer(0)](e)
(u + v)*a + (-v*w - w^2 + u)/w
>>> _.parent()
Splitting Algebra of T^2 + T + 1 with roots [E3, -E3 - 1]
over Splitting Algebra of h^3 - u*h^2 + v*h - w
with roots [a, b, -b - a + u]
over Multivariate Polynomial Ring in u, v, w
over Integer Ring localized at (w,)

```

chevie()

Return the GAP3-CHEVIE realization of the corresponding cyclotomic Hecke algebra in the finite-dimensional case.

EXAMPLES:

```

sage: CHA3 = algebras.CubicHecke(3) # optional gap3
sage: CHA3.chevie()                 # optional gap3
Hecke(G4, [[a,b,c]])

```

```

>>> from sage.all import *
>>> CHA3 = algebras.CubicHecke(Integer(3)) # optional gap3
>>> CHA3.chevie()                       # optional gap3
Hecke(G4, [[a,b,c]])

```

cubic_braid_group()

Return the cubic braid group attached to `self`.

EXAMPLES:

```
sage: CHA2 = algebras.CubicHecke(2)
sage: CHA2.cubic_braid_group()
Cubic Braid group on 2 strands
```

```
>>> from sage.all import *
>>> CHA2 = algebras.CubicHecke(Integer(2))
>>> CHA2.cubic_braid_group()
Cubic Braid group on 2 strands
```

cubic_braid_group_algebra()

Return the group algebra of cubic braid group attached to `self` over the base ring of `self`.

EXAMPLES:

```
sage: CHA2 = algebras.CubicHecke(2)
sage: CHA2.cubic_braid_group_algebra()
Algebra of Cubic Braid group on 2 strands
over Multivariate Polynomial Ring in u, v, w
over Integer Ring localized at (w,)
```

```
>>> from sage.all import *
>>> CHA2 = algebras.CubicHecke(Integer(2))
>>> CHA2.cubic_braid_group_algebra()
Algebra of Cubic Braid group on 2 strands
over Multivariate Polynomial Ring in u, v, w
over Integer Ring localized at (w,)
```

cubic_equation(`var='h'`, `as_coefficients=False`, `generic=False`)

Return the cubic equation attached to `self`.

INPUT:

- `var` – string (default: 'h'); setting the indeterminate of the equation
- `as_coefficients` – boolean (default: False); if set to True the list of coefficients is returned
- `generic` – boolean (default: False); if set to True the cubic equation will be given over the generic base ring

OUTPUT:

A polynomial over the base ring (resp. generic base ring if `generic` is set to True). In case `as_coefficients` is set to True a list of them is returned.

EXAMPLES:

```
sage: CHA2 = algebras.CubicHecke(2, cubic_equation_roots=(E(3), ~E(3), 1))
sage: CHA2.cubic_equation()
h^3 - 1
sage: CHA2.cubic_equation(generic=True)
h^3 - u*h^2 + v*h - w
sage: CHA2.cubic_equation(as_coefficients=True, generic=True)
```

(continues on next page)

(continued from previous page)

```
[-w, v, -u, 1]
sage: CHA2.cubic_equation(as_coefficients=True)
[-1, 0, 0, 1]
```

```
>>> from sage.all import *
>>> CHA2 = algebras.CubicHecke(Integer(2), cubic_equation_
↳ roots=(E(Integer(3)), ~E(Integer(3)), Integer(1)))
>>> CHA2.cubic_equation()
h^3 - 1
>>> CHA2.cubic_equation(generic=True)
h^3 - u*h^2 + v*h - w
>>> CHA2.cubic_equation(as_coefficients=True, generic=True)
[-w, v, -u, 1]
>>> CHA2.cubic_equation(as_coefficients=True)
[-1, 0, 0, 1]
```

cubic_equation_parameters (*generic=False*)

Return the coefficients of the underlying cubic equation.

INPUT:

- *generic* – boolean (default: False); if set to True the coefficients are returned as elements of the generic base ring

OUTPUT: a triple consisting of the coefficients

EXAMPLES:

```
sage: CHA2 = algebras.CubicHecke(2, cubic_equation_roots=(3, 4, 5))
sage: CHA2.cubic_equation()
h^3 - 12*h^2 + 47*h - 60
sage: CHA2.cubic_equation_parameters()
[12, 47, 60]
sage: CHA2.cubic_equation_parameters(generic=True)
[u, v, w]
```

```
>>> from sage.all import *
>>> CHA2 = algebras.CubicHecke(Integer(2), cubic_equation_roots=(Integer(3),
↳ Integer(4), Integer(5)))
>>> CHA2.cubic_equation()
h^3 - 12*h^2 + 47*h - 60
>>> CHA2.cubic_equation_parameters()
[12, 47, 60]
>>> CHA2.cubic_equation_parameters(generic=True)
[u, v, w]
```

cubic_equation_roots (*generic=False*)

Return the roots of the underlying cubic equation.

INPUT:

- *generic* – boolean (default: False); if set to True the roots are returned as elements of the generic extension ring

OUTPUT: a triple consisting of the roots

EXAMPLES:

```

sage: CHA2 = algebras.CubicHecke(2, cubic_equation_roots=(3, 4, 5))
sage: CHA2.cubic_equation()
h^3 - 12*h^2 + 47*h - 60
sage: CHA2.cubic_equation_roots()
[3, 4, 5]
sage: CHA2.cubic_equation_roots(generic=True)
[a, b, c]

```

```

>>> from sage.all import *
>>> CHA2 = algebras.CubicHecke(Integer(2), cubic_equation_roots=(Integer(3), Integer(4), Integer(5)))
>>> CHA2.cubic_equation()
h^3 - 12*h^2 + 47*h - 60
>>> CHA2.cubic_equation_roots()
[3, 4, 5]
>>> CHA2.cubic_equation_roots(generic=True)
[a, b, c]

```

cubic_hecke_subalgebra (*nstrands=None*)

Return a *CubicHeckeAlgebra* that realizes a sub-algebra of *self* on the first *n_strands* strands.

INPUT:

- *nstrands* – integer at least 1 and at most *strands()* giving the number of strands for the subgroup; the default is one strand less than *self* has

OUTPUT: an instance of this class realizing the sub-algebra

EXAMPLES:

```

sage: CHA3 = algebras.CubicHecke(3, cubic_equation_roots=(3, 4, 5))
sage: CHA3.cubic_hecke_subalgebra()
Cubic Hecke algebra on 2 strands
over Integer Ring localized at (2, 3, 5)
with cubic equation: h^3 - 12*h^2 + 47*h - 60 = 0

```

```

>>> from sage.all import *
>>> CHA3 = algebras.CubicHecke(Integer(3), cubic_equation_roots=(Integer(3), Integer(4), Integer(5)))
>>> CHA3.cubic_hecke_subalgebra()
Cubic Hecke algebra on 2 strands
over Integer Ring localized at (2, 3, 5)
with cubic equation: h^3 - 12*h^2 + 47*h - 60 = 0

```

cyclotomic_generator (*generic=False*)

Return the third root of unity as element of the extension ring.

The only thing where this is needed is in the nine dimensional irreducible representations of the cubic Hecke algebra on four strands (see the examples of *CubicHeckeElement.matrix()* for instance).

INPUT:

- *generic* – boolean (default: *False*); if *True* the cyclotomic generator is returned as an element extension ring of definition

EXAMPLES:

```
sage: CHA2 = algebras.CubicHecke(2, cubic_equation_roots=(3, 4, 5))
sage: CHA2.cyclotomic_generator()
E3
sage: CHA2.cyclotomic_generator(generic=True)
e3
```

```
>>> from sage.all import *
>>> CHA2 = algebras.CubicHecke(Integer(2), cubic_equation_roots=(Integer(3), Integer(4), Integer(5)))
>>> CHA2.cyclotomic_generator()
E3
>>> CHA2.cyclotomic_generator(generic=True)
e3
```

extension_ring(generic=False)

Return the extension ring of self.

This is an extension of its base ring containing the roots of the cubic equation.

INPUT:

- generic – boolean (default: False); if True the extension ring of definition (here often called the generic extension ring) is returned

EXAMPLES:

```
sage: CHA2 = algebras.CubicHecke(2, cubic_equation_roots=(3, 4, 5))
sage: CHA2.extension_ring()
Splitting Algebra of T^2 + T + 1 with roots [E3, -E3 - 1]
over Integer Ring localized at (2, 3, 5)
sage: CHA2.extension_ring(generic=True)
Multivariate Laurent Polynomial Ring in a, b, c
over Splitting Algebra of x^2 + x + 1
with roots [e3, -e3 - 1] over Integer Ring
```

```
>>> from sage.all import *
>>> CHA2 = algebras.CubicHecke(Integer(2), cubic_equation_roots=(Integer(3), Integer(4), Integer(5)))
>>> CHA2.extension_ring()
Splitting Algebra of T^2 + T + 1 with roots [E3, -E3 - 1]
over Integer Ring localized at (2, 3, 5)
>>> CHA2.extension_ring(generic=True)
Multivariate Laurent Polynomial Ring in a, b, c
over Splitting Algebra of x^2 + x + 1
with roots [e3, -e3 - 1] over Integer Ring
```

filecache_section()

Return the enum to select a section in the file cache.

EXAMPLES:

```
sage: CHA2 = algebras.CubicHecke(2)
sage: list(CH A2.filecache_section())
```

(continues on next page)

(continued from previous page)

```
[<section.matrix_representations: 'matrix_representations'>,
 <section.braid_images: 'braid_images'>,
 <section.basis_extensions: 'basis_extensions'>,
 <section.markov_trace: 'markov_trace'>]
```

```
>>> from sage.all import *
>>> CHA2 = algebras.CubicHecke(Integer(2))
>>> list(CHA2.filecache_section())
[<section.matrix_representations: 'matrix_representations'>,
 <section.braid_images: 'braid_images'>,
 <section.basis_extensions: 'basis_extensions'>,
 <section.markov_trace: 'markov_trace'>]
```

garside_involution(*element*)

Return the image of the given element of *self* under the extension of the Garside involution of braids to *self*.

This method may be invoked by the `revert_garside` method of the element class of *self*, alternatively.

INPUT:

- *element* – instance of the element class of *self*

OUTPUT:

Instance of the element class of *self* representing the image of *element* under the extension of the Garside involution to *self*.

EXAMPLES:

```
sage: CHA3 = algebras.CubicHecke(3)
sage: ele = CHA3.an_element()
sage: ele_gar = CHA3.garside_involution(ele); ele_gar
-w*c1*c0^-1 + u*c0 + v*c1 - ((v*w-u)/w)
sage: ele == CHA3.garside_involution(ele_gar)
True
```

```
>>> from sage.all import *
>>> CHA3 = algebras.CubicHecke(Integer(3))
>>> ele = CHA3.an_element()
>>> ele_gar = CHA3.garside_involution(ele); ele_gar
-w*c1*c0^-1 + u*c0 + v*c1 - ((v*w-u)/w)
>>> ele == CHA3.garside_involution(ele_gar)
True
```

gen(*i*)

The *i*-th generator of the algebra.

EXAMPLES:

```
sage: CHA2 = algebras.CubicHecke(2)
sage: CHA2.gen(0)
c
```

```
>>> from sage.all import *
>>> CHA2 = algebras.CubicHecke(Integer(2))
>>> CHA2.gen(Integer(0))
c
```

gens()

Return the generators of self.

EXAMPLES:

```
sage: CHA2 = algebras.CubicHecke(2)
sage: CHA2.gens()
(c,)
```

```
>>> from sage.all import *
>>> CHA2 = algebras.CubicHecke(Integer(2))
>>> CHA2.gens()
(c,)
```

get_order()

Return an ordering of the basis of self.

EXAMPLES:

```
sage: CHA3 = algebras.CubicHecke(3)
sage: len(CH A3.get_order())
24
```

```
>>> from sage.all import *
>>> CHA3 = algebras.CubicHecke(Integer(3))
>>> len(CH A3.get_order())
24
```

irred_repr

alias of *AbsIrreducibeRep*

is_filecache_empty(*section=None*)

Return True if the file cache of the given section is empty. If no section is given the answer is given for the complete file cache.

INPUT:

- *section* – (default: all sections) an element of enum *section* that can be selected using *file-cache_section()*

EXAMPLES:

```
sage: CHA2 = algebras.CubicHecke(2)
sage: CHA2.is_filecache_empty()
False
```

```
>>> from sage.all import *
>>> CHA2 = algebras.CubicHecke(Integer(2))
>>> CHA2.is_filecache_empty()
False
```

mirror_image()

Return a copy of `self` with the mirrored cubic equation, that is: the cubic equation has the inverse roots to the roots with respect to `self`.

This is needed since the mirror involution of the braid group does not factor through `self` (considered as an algebra over the base ring, just considered as $\mathbf{Z}$ -algebra). Therefore, the mirror involution of an element of `self` belongs to `mirror_image`.

OUTPUT:

A cubic Hecke algebra over the same base and extension ring, but whose cubic equation is transformed by the mirror involution applied to its coefficients and roots.

EXAMPLES:

```
sage: CHA2 = algebras.CubicHecke(2)
sage: ce = CHA2.cubic_equation(); ce
h^3 - u*h^2 + v*h - w
sage: CHA2m = CHA2.mirror_image()
sage: cem = CHA2m.cubic_equation(); cem
h^3 + ((-v)/w)*h^2 + u/w*h + (-1)/w
sage: mi = CHA2.base_ring().mirror_involution(); mi
Ring endomorphism of Multivariate Polynomial Ring in u, v, w
      over Integer Ring localized at (w,)
Defn: u |--> v/w
      v |--> u/w
      w |--> 1/w
sage: cem == cem.parent()([mi(cf) for cf in ce.coefficients()])
True
```

```
>>> from sage.all import *
>>> CHA2 = algebras.CubicHecke(Integer(2))
>>> ce = CHA2.cubic_equation(); ce
h^3 - u*h^2 + v*h - w
>>> CHA2m = CHA2.mirror_image()
>>> cem = CHA2m.cubic_equation(); cem
h^3 + ((-v)/w)*h^2 + u/w*h + (-1)/w
>>> mi = CHA2.base_ring().mirror_involution(); mi
Ring endomorphism of Multivariate Polynomial Ring in u, v, w
      over Integer Ring localized at (w,)
Defn: u |--> v/w
      v |--> u/w
      w |--> 1/w
>>> cem == cem.parent()([mi(cf) for cf in ce.coefficients()])
True
```

Note that both cubic Hecke algebras have the same ring of definition and identical generic cubic equation:

```
sage: cemg = CHA2m.cubic_equation(generic=True)
sage: CHA2.cubic_equation(generic=True) == cemg
True
sage: CHA2.cubic_equation() == cemg
True
sage: a, b, c = CHA2.cubic_equation_roots()
sage: CHA2m.cubic_equation_roots(generic=True) == [a, b, c]
```

(continues on next page)

(continued from previous page)

```

True
sage: CHA2m.cubic_equation_roots()
[((-1)/(-w))*a^2 + (u/(-w))*a + (-v)/(-w),
 ((1/(-w))*a)*b + (1/(-w))*a^2 + ((-u)/(-w))*a,
 (((-1)/(-w))*a)*b]
sage: ai, bi, ci = _
sage: ai == ~a, bi == ~b, ci == ~c
(True, True, True)
sage: CHA2.extension_ring(generic=True).mirror_involution()
Ring endomorphism of Multivariate Laurent Polynomial Ring in a, b, c
      over Splitting Algebra of x^2 + x + 1
      with roots [e3, -e3 - 1] over Integer Ring

Defn: a |--> a^-1
      b |--> b^-1
      c |--> c^-1
      with map of base ring

```

```

>>> from sage.all import *
>>> cemg = CHA2m.cubic_equation(generic=True)
>>> CHA2.cubic_equation(generic=True) == cemg
True
>>> CHA2.cubic_equation() == cemg
True
>>> a, b, c = CHA2.cubic_equation_roots()
>>> CHA2m.cubic_equation_roots(generic=True) == [a, b, c]
True
>>> CHA2m.cubic_equation_roots()
[((-1)/(-w))*a^2 + (u/(-w))*a + (-v)/(-w),
 ((1/(-w))*a)*b + (1/(-w))*a^2 + ((-u)/(-w))*a,
 (((-1)/(-w))*a)*b]
>>> ai, bi, ci = _
>>> ai == ~a, bi == ~b, ci == ~c
(True, True, True)
>>> CHA2.extension_ring(generic=True).mirror_involution()
Ring endomorphism of Multivariate Laurent Polynomial Ring in a, b, c
      over Splitting Algebra of x^2 + x + 1
      with roots [e3, -e3 - 1] over Integer Ring

Defn: a |--> a^-1
      b |--> b^-1
      c |--> c^-1
      with map of base ring

```

The mirror image can not be obtained for specialized cubic Hecke algebras if the specialization does not factor through the mirror involution on the ring if definition:

```

sage: CHA2s = algebras.CubicHecke(2, cubic_equation_roots=(3, 4, 5))
sage: CHA2s
Cubic Hecke algebra on 2 strands
      over Integer Ring localized at (2, 3, 5)
      with cubic equation: h^3 - 12*h^2 + 47*h - 60 = 0

```

```
>>> from sage.all import *
>>> CHA2s = algebras.CubicHecke(Integer(2), cubic_equation_roots=(Integer(3), Integer(4), Integer(5)))
>>> CHA2s
Cubic Hecke algebra on 2 strands
over Integer Ring localized at (2, 3, 5)
with cubic equation: h^3 - 12*h^2 + 47*h - 60 = 0
```

In the next example it is not clear what the mirror image of 7 should be:

```
sage: CHA2s.mirror_image()
Traceback (most recent call last):
...
RuntimeError: base ring Integer Ring localized at (2, 3, 5)
does not factor through mirror involution
```

```
>>> from sage.all import *
>>> CHA2s.mirror_image()
Traceback (most recent call last):
...
RuntimeError: base ring Integer Ring localized at (2, 3, 5)
does not factor through mirror involution
```

mirror_isomorphism(*element*)

Return the image of the given element of *self* under the extension of the mirror involution of braids to *self*. The mirror involution of a braid is given by inverting all generators in the braid word. It does not factor through *self* over the base ring but it factors through *self* considered as a $\mathbf{Z}$ -module relative to the mirror automorphism of the generic base ring. Considering *self* as algebra over its base ring this involution defines an isomorphism of *self* onto a different cubic Hecke algebra with a different cubic equation. This is defined over a different base (and extension) ring than *self*. It can be obtained by the method `mirror_image` or as parent of the output of this method.

This method may be invoked by the `CubicHeckeElement.revert_mirror` method of the element class of *self*, alternatively.

INPUT:

- *element* – instance of the element class of *self*

OUTPUT:

Instance of the element class of the mirror image of *self* representing the image of *element* under the extension of the braid mirror involution to *self*.

EXAMPLES:

```
sage: CHA3 = algebras.CubicHecke(3)
sage: ele = CHA3.an_element()
sage: ele_mirr = CHA3.mirror_isomorphism(ele); ele_mirr
-1/w*c0^-1*c1 + u/w*c0^-1 + v/w*c1^-1 + ((v*w-u)/w)
sage: ele_mirr2 = ele.revert_mirror() # indirect doctest
sage: ele_mirr == ele_mirr2
True
sage: par_mirr = ele_mirr.parent()
sage: par_mirr == CHA3
```

(continues on next page)

(continued from previous page)

```
False
sage: par_mirr == CHA3.mirror_image()
True
sage: ele == par_mirr.mirror_isomorphism(ele_mirr)
True
```

```
>>> from sage.all import *
>>> CHA3 = algebras.CubicHecke(Integer(3))
>>> ele = CHA3.an_element()
>>> ele_mirr = CHA3.mirror_isomorphism(ele); ele_mirr
-1/w*c0^-1*c1 + u/w*c0^-1 + v/w*c1^-1 + ((v*w-u)/w)
>>> ele_mirr2 = ele.revert_mirror() # indirect doctest
>>> ele_mirr == ele_mirr2
True
>>> par_mirr = ele_mirr.parent()
>>> par_mirr == CHA3
False
>>> par_mirr == CHA3.mirror_image()
True
>>> ele == par_mirr.mirror_isomorphism(ele_mirr)
True
```

ngens()

The number of generators of the algebra.

EXAMPLES:

```
sage: CHA2 = algebras.CubicHecke(2)
sage: CHA2.ngens()
1
```

```
>>> from sage.all import *
>>> CHA2 = algebras.CubicHecke(Integer(2))
>>> CHA2.ngens()
1
```

one_basis()

Return the index of the basis element for the identity element in the cubic braid group.

EXAMPLES:

```
sage: CHA2 = algebras.CubicHecke(2)
sage: CHA2.one_basis()
1
```

```
>>> from sage.all import *
>>> CHA2 = algebras.CubicHecke(Integer(2))
>>> CHA2.one_basis()
1
```

orientation_antiinvolution(element)

Return the image of the given element of `self` under the extension of the orientation anti involution of braids

to `self`. The orientation anti involution of a braid is given by reversing the order of generators in the braid word.

This method may be invoked by the `revert_orientation` method of the element class of `self`, alternatively.

INPUT:

- `element` – instance of the element class of `self`

OUTPUT:

Instance of the element class of `self` representing the image of `element` under the extension of the orientation reversing braid involution to `self`.

EXAMPLES:

```
sage: CHA3 = algebras.CubicHecke(3)
sage: ele = CHA3.an_element()
sage: ele_ori = CHA3.orientation_antiinvolution(ele); ele_ori
-w*c1^-1*c0 + v*c0 + u*c1 - ((v*w-u)/w)
sage: ele == CHA3.orientation_antiinvolution(ele_ori)
True
```

```
>>> from sage.all import *
>>> CHA3 = algebras.CubicHecke(Integer(3))
>>> ele = CHA3.an_element()
>>> ele_ori = CHA3.orientation_antiinvolution(ele); ele_ori
-w*c1^-1*c0 + v*c0 + u*c1 - ((v*w-u)/w)
>>> ele == CHA3.orientation_antiinvolution(ele_ori)
True
```

product_on_basis (*g1*, *g2*)

Return product on basis elements indexed by *g1* and *g2*.

EXAMPLES:

```
sage: CHA3 = algebras.CubicHecke(3)
sage: g = CHA3.basis().keys().an_element(); g
c0*c1
sage: CHA3.product_on_basis(g, ~g)
1
sage: CHA3.product_on_basis(g, g)
w*c0^-1*c1*c0 - v*c1*c0 + u*c0*c1*c0
```

```
>>> from sage.all import *
>>> CHA3 = algebras.CubicHecke(Integer(3))
>>> g = CHA3.basis().keys().an_element(); g
c0*c1
>>> CHA3.product_on_basis(g, ~g)
1
>>> CHA3.product_on_basis(g, g)
w*c0^-1*c1*c0 - v*c1*c0 + u*c0*c1*c0
```

repr_type

alias of *RepresentationType*

reset_filecache (*section=None, commit=True*)

Reset the file cache of the given *section* resp. the complete file cache if no *section* is given.

INPUT:

- *section* – (default: all sections) an element of enum *section* that can be selected using *filecache_section()*
- *commit* – boolean (default: True); if set to False the reset is not written to the filesystem

EXAMPLES:

```
sage: # optional - database_cubic_hecke
sage: CHA5 = algebras.CubicHecke(5)
sage: be = CHA5.filecache_section().basis_extensions
sage: CHA5.is_filecache_empty(be)
False
sage: CHA5.reset_filecache(be)
sage: CHA5.is_filecache_empty(be)
True
```

```
>>> from sage.all import *
>>> # optional - database_cubic_hecke
>>> CHA5 = algebras.CubicHecke(Integer(5))
>>> be = CHA5.filecache_section().basis_extensions
>>> CHA5.is_filecache_empty(be)
False
>>> CHA5.reset_filecache(be)
>>> CHA5.is_filecache_empty(be)
True
```

schur_element (*item, generic=False*)

Return a single Schur element of *self* as elements of the extension ring of *self*.

Note

This method needs GAP3 installed with package CHEVIE.

INPUT:

- *item* – an element of *AbsIrreducibeRep* to give the irreducible representation of *self* to which the Schur element should be returned
- *generic* – boolean (default: False); if True, the element is returned as element of the generic extension ring

EXAMPLES:

```
sage: CHA3 = algebras.CubicHecke(3) # optional gap3
sage: CHA3.schur_element(CHA3.irred_repr.W3_111) # optional gap3
(u^3*w + v^3 - 6*u*v*w + 8*w^2)/w^2
```

```
>>> from sage.all import *
>>> CHA3 = algebras.CubicHecke(Integer(3)) # optional gap3
>>> CHA3.schur_element(CHA3.irred_repr.W3_111) # optional gap3
(u^3*w + v^3 - 6*u*v*w + 8*w^2)/w^2
```


schur_elements (*generic=False*)

Return the list of Schur elements of `self` as elements of the extension ring of `self`.

Note

This method needs GAP3 installed with package CHEVIE.

INPUT:

- `generic` – boolean (default: `False`); if `True`, the element is returned as element of the generic extension ring

EXAMPLES:

```
sage: CHA3 = algebras.CubicHecke(3)          # optional gap3
sage: sch_eles = CHA3.schur_elements()       # optional gap3
sage: sch_eles[6]                           # optional gap3
(u^3*w + v^3 - 6*u*v*w + 8*w^2)/w^2
```

```
>>> from sage.all import *
>>> CHA3 = algebras.CubicHecke(Integer(3))   # optional gap3
>>> sch_eles = CHA3.schur_elements()         # optional gap3
>>> sch_eles[Integer(6)]                     # optional gap3
(u^3*w + v^3 - 6*u*v*w + 8*w^2)/w^2
```

strands ()

Return the number of strands of the braid group whose group algebra image is `self`.

EXAMPLES:

```
sage: CHA4 = algebras.CubicHecke(2)
sage: CHA4.strands()
2
```

```
>>> from sage.all import *
>>> CHA4 = algebras.CubicHecke(Integer(2))
>>> CHA4.strands()
2
```

class `sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeElement`

Bases: `IndexedFreeModuleElement`

An element of a `CubicHeckeAlgebra`.

For more information see `CubicHeckeAlgebra`.

EXAMPLES:

```
sage: CHA3s = algebras.CubicHecke('s1, s2'); CHA3s.an_element()
-w*s1*s2^-1 + v*s1 + u*s2 - ((v*w-u)/w)
sage: CHA3.<c1, c2> = algebras.CubicHecke(3)
sage: c1**3*c2
u*w*c1^-1*c2^-1 + (u^2-v)*c1*c2^-1 - (u*v-w)*c2^-1
```

```

>>> from sage.all import *
>>> CHA3s = algebras.CubicHecke('s1, s2'); CHA3s.an_element()
-w*s1*s2^-1 + v*s1 + u*s2 - ((v*w-u)/w)
>>> CHA3 = algebras.CubicHecke(Integer(3), names=('c1', 'c2',)); (c1, c2,) = CHA3.
↳_first_ngens(2)
>>> c1**Integer(3)*~c2
u*w*c1^-1*c2^-1 + (u^2-v)*c1*c2^-1 - (u*v-w)*c2^-1

```

Tietze()

Return the Tietze presentation of `self` if `self` belongs to the basis of its parent and `None` otherwise.

OUTPUT:

A tuple representing the pre image braid of `self` if `self` is a monomial from the basis `None` else-wise

EXAMPLES:

```

sage: CHA3 = algebras.CubicHecke(3)
sage: ele = CHA3.an_element(); ele
-w*c0*c1^-1 + v*c0 + u*c1 - ((v*w-u)/w)
sage: ele.Tietze() is None
True
sage: [CHA3(sp).Tietze() for sp in ele.support()]
[(), (1,), (1, -2), (2,)]

```

```

>>> from sage.all import *
>>> CHA3 = algebras.CubicHecke(Integer(3))
>>> ele = CHA3.an_element(); ele
-w*c0*c1^-1 + v*c0 + u*c1 - ((v*w-u)/w)
>>> ele.Tietze() is None
True
>>> [CHA3(sp).Tietze() for sp in ele.support()]
[(), (1,), (1, -2), (2,)]

```

braid_group_algebra_pre_image()

Return a pre image of `self` in the group algebra of the braid group (with respect to the basis given by Ivan Marin).

OUTPUT:

The pre image of `self` as instance of the element class of the group algebra of the BraidGroup

EXAMPLES:

```

sage: CHA3 = algebras.CubicHecke(3)
sage: ele = CHA3.an_element(); ele
-w*c0*c1^-1 + v*c0 + u*c1 - ((v*w-u)/w)
sage: b_ele = ele.braid_group_algebra_pre_image(); b_ele
-((v*w-u)/w) + v*c0 + u*c1 - w*c0*c1^-1
sage: ele in CHA3
True
sage: b_ele in CHA3
False
sage: b_ele in CHA3.braid_group_algebra()
True

```

```

>>> from sage.all import *
>>> CHA3 = algebras.CubicHecke(Integer(3))
>>> ele = CHA3.an_element(); ele
-w*c0*c1^-1 + v*c0 + u*c1 - ((v*w-u)/w)
>>> b_ele = ele.braid_group_algebra_pre_image(); b_ele
-((v*w-u)/w) + v*c0 + u*c1 - w*c0*c1^-1
>>> ele in CHA3
True
>>> b_ele in CHA3
False
>>> b_ele in CHA3.braid_group_algebra()
True

```

`cubic_braid_group_algebra_pre_image()`

Return a pre image of `self` in the group algebra of the cubic braid group.

OUTPUT:

The pre image of `self` as instance of the element class of the group algebra of the `CubicBraidGroup`.

EXAMPLES:

```

sage: CHA3 = algebras.CubicHecke(3)
sage: ele = CHA3.an_element(); ele
-w*c0*c1^-1 + v*c0 + u*c1 - ((v*w-u)/w)
sage: cb_ele = ele.cubic_braid_group_algebra_pre_image(); cb_ele
-w*c0*c1^-1 + v*c0 + u*c1 - ((v*w-u)/w)
sage: ele in CHA3
True
sage: cb_ele in CHA3
False
sage: cb_ele in CHA3.cubic_braid_group_algebra()
True

```

```

>>> from sage.all import *
>>> CHA3 = algebras.CubicHecke(Integer(3))
>>> ele = CHA3.an_element(); ele
-w*c0*c1^-1 + v*c0 + u*c1 - ((v*w-u)/w)
>>> cb_ele = ele.cubic_braid_group_algebra_pre_image(); cb_ele
-w*c0*c1^-1 + v*c0 + u*c1 - ((v*w-u)/w)
>>> ele in CHA3
True
>>> cb_ele in CHA3
False
>>> cb_ele in CHA3.cubic_braid_group_algebra()
True

```

`formal_markov_trace` (*extended=False, field_embedding=False*)

Return a formal expression which can be specialized to Markov traces which factor through the cubic Hecke algebra.

This covers Markov traces corresponding to the

- HOMFLY-PT polynomial,
- Kauffman polynomial,

- Links-Gould polynomial.

These expressions are elements of a sub-module of the module of linear forms on `self` the base ring of which is an extension of the generic base ring of `self` by an additional variable `s` representing the writhe factor. All variables of this base ring extension are invertible.

A Markov trace is a family of class functions tr_n on the family of braid groups B_n into some commutative ring R depending on a unit $s \in R$ such that for all $b \in B_n$ the following two conditions are satisfied (see [Kau1991], section 7):

$$\begin{aligned} tr_{n+1}(bg_n) &= str_n(b), \\ tr_{n+1}(bg_n^{-1}) &= s^{-1}tr_n(b). \end{aligned}$$

The unit s is often called the writhe factor and corresponds to the additional variable mentioned above.

Note

Currently it is not known if all linear forms of this sub-module belong to a Markov trace, i.e. can be extended to the full tower of cubic Hecke algebras. Anyway, at least the four basis elements (U_1 , U_2 , U_3 and K_4) can be reconstructed from the HOMFLY-PT and Kauffman polynomial.

INPUT:

- `extended` – boolean (default: `False`); if set to `True` the base ring of the Markov trace module is constructed as an extension of generic extension ring of `self`; per default it is constructed upon the generic base ring
- `field_embedding` – boolean (default: `False`); if set to `True` the base ring of the module is the smallest field containing the generic extension ring of `self`; ignored if `extended=False`

EXAMPLES:

```
sage: from sage.knots.knotinfo import KnotInfo
sage: CHA2 = algebras.CubicHecke(2)
sage: K3_1 = KnotInfo.K3_1
sage: b3_1 = CHA2(K3_1.braid())
sage: mt3_1 = b3_1.formal_markov_trace(); mt3_1
((u^2*s^2-v*s^2+u*w)/s)*B[U1] - (u*v-w)*B[U2]
sage: mt3_1.parent()
Free module generated by {U1, U2}
over Multivariate Polynomial Ring in u, v, w, s
over Integer Ring localized at (s, w, v, u)

sage: f = b3_1.formal_markov_trace(extended=True); f
(a^2*b*c*s^-1+a*b^2*c*s^-1+a*b*c^2*s^-1+a^2*s+a*b*s+b^2*s+a*c*s+b*c*s+c^
↪2*s)*B[U1]
+ (-a^2*b-a*b^2-a^2*c+(-2)*a*b*c-b^2*c-a*c^2-b*c^2)*B[U2]
sage: f.parent().base_ring()
Multivariate Laurent Polynomial Ring in a, b, c, s
over Splitting Algebra of x^2 + x + 1 with roots [e3, -e3 - 1]
over Integer Ring

sage: f = b3_1.formal_markov_trace(extended=True, field_embedding=True); f
((a^2*b*c+a*b^2*c+a*b*c^2+a^2*s^2+a*b*s^2+b^2*s^2+a*c*s^2+b*c*s^2+c^2*s^2)/
↪s)*B[U1]
- (a^2*b+a*b^2+a^2*c+2*a*b*c+b^2*c+a*c^2+b*c^2)*B[U2]
```

(continues on next page)

(continued from previous page)

```

sage: f.parent().base_ring()
Fraction Field of Multivariate Polynomial Ring in a, b, c, s
over Cyclotomic Field of order 3 and degree 2

>>> from sage.all import *
>>> from sage.knots.knotinfo import KnotInfo
>>> CHA2 = algebras.CubicHecke(Integer(2))
>>> K3_1 = KnotInfo.K3_1
>>> b3_1 = CHA2(K3_1.braid())
>>> mt3_1 = b3_1.formal_markov_trace(); mt3_1
((u^2*s^2-v*s^2+u*w)/s)*B[U1] - (u*v-w)*B[U2]
>>> mt3_1.parent()
Free module generated by {U1, U2}
over Multivariate Polynomial Ring in u, v, w, s
over Integer Ring localized at (s, w, v, u)

>>> f = b3_1.formal_markov_trace(extended=True); f
(a^2*b*c*s^-1+a*b^2*c*s^-1+a*b*c^2*s^-1+a^2*s+a*b*s+b^2*s+a*c*s+b*c*s+c^
↪2*s)*B[U1]
+ (-a^2*b-a*b^2-a^2*c+(-2)*a*b*c-b^2*c-a*c^2-b*c^2)*B[U2]
>>> f.parent().base_ring()
Multivariate Laurent Polynomial Ring in a, b, c, s
over Splitting Algebra of x^2 + x + 1 with roots [e3, -e3 - 1]
over Integer Ring

>>> f = b3_1.formal_markov_trace(extended=True, field_embedding=True); f
((a^2*b*c+a*b^2*c+a*b*c^2+a^2*s^2+a*b*s^2+b^2*s^2+a*c*s^2+b*c*s^2+c^2*s^2)/
↪s)*B[U1]
- (a^2*b+a*b^2+a^2*c+2*a*b*c+b^2*c+a*c^2+b*c^2)*B[U2]
>>> f.parent().base_ring()
Fraction Field of Multivariate Polynomial Ring in a, b, c, s
over Cyclotomic Field of order 3 and degree 2

```

Obtaining the well known link invariants from it:

```

sage: MT = mt3_1.base_ring()
sage: sup = mt3_1.support()
sage: u, v, w, s = mt3_1.base_ring().gens()
sage: LK3_1 = mt3_1*s**(-3) # since the writhe of K3_1 is 3
sage: f = MT.specialize_homfly()
sage: g = sum(f(LK3_1.coefficient(b)) * b.regular_homfly_polynomial() for b_
↪in sup); g
L^-2*M^2 - 2*L^-2 - L^-4
sage: g == K3_1.link().homfly_polynomial()
True

sage: f = MT.specialize_kauffman()
sage: g = sum(f(LK3_1.coefficient(b)) * b.regular_kauffman_polynomial() for b_
↪in sup); g
a^-2*z^2 - 2*a^-2 + a^-3*z + a^-4*z^2 - a^-4 + a^-5*z
sage: g == K3_1.kauffman_polynomial()
True

```

(continues on next page)

(continued from previous page)

```

sage: f = MT.specialize_links_gould()
sage: g = sum(f(LK3_1.coefficient(b)) * b.links_gould_polynomial() for b in_
↳sup); g
-t0^2*t1 - t0*t1^2 + t0^2 + 2*t0*t1 + t1^2 - t0 - t1 + 1
sage: g == K3_1.link().links_gould_polynomial()
True

```

```

>>> from sage.all import *
>>> MT = mt3_1.base_ring()
>>> sup = mt3_1.support()
>>> u, v, w, s = mt3_1.base_ring().gens()
>>> LK3_1 = mt3_1*s*-Integer(3) # since the writhe of K3_1 is 3
>>> f = MT.specialize_homfly()
>>> g = sum(f(LK3_1.coefficient(b)) * b.regular_homfly_polynomial() for b in_
↳sup); g
L^-2*M^2 - 2*L^-2 - L^-4
>>> g == K3_1.link().homfly_polynomial()
True

>>> f = MT.specialize_kauffman()
>>> g = sum(f(LK3_1.coefficient(b)) * b.regular_kauffman_polynomial() for b_
↳in sup); g
a^-2*z^2 - 2*a^-2 + a^-3*z + a^-4*z^2 - a^-4 + a^-5*z
>>> g == K3_1.kauffman_polynomial()
True

>>> f = MT.specialize_links_gould()
>>> g = sum(f(LK3_1.coefficient(b)) * b.links_gould_polynomial() for b in_
↳sup); g
-t0^2*t1 - t0*t1^2 + t0^2 + 2*t0*t1 + t1^2 - t0 - t1 + 1
>>> g == K3_1.link().links_gould_polynomial()
True

```

matrix (*subdivide=False, representation_type=None, original=False*)

Return certain types of matrix representations of *self*.

The absolutely irreducible representations of the cubic Hecke algebra are constructed using the GAP3 interface and the CHEVIE package if GAP3 and CHEVIE are installed on the system. Furthermore, the representations given on [Ivan Marin's homepage](#) are used:

INPUT:

- *subdivide* – boolean (default: `False`); this boolean is passed to the `block_matrix` function
- *representation_type* – instance of enum `RepresentationType`; this can be obtained by the attribute `CubicHeckeAlgebra.repr_type` of *self*; the following values are possible:
 - `RegularLeft` – (regular left repr. from the above URL)
 - `RegularRight` – (regular right repr. from the above URL)
 - `SplitIrredChevie` – (split irred. repr. via CHEVIE)
 - `SplitIrredMarin` – (split irred. repr. from the above URL)

- default: `SplitIrredChevie` taken if GAP3 and CHEVIE are installed on the system, otherwise the default will be `SplitIrredMarin`
- `original` – boolean (default: `False`); if set to `True` the base ring of the matrix will be the generic `base_ring` resp. generic extension ring (for the split versions) of the parent of `self`

OUTPUT:

An instance of `CubicHeckeMatrixRep`, which is inherited from `Matrix_generic_dense`. In the case of the irreducible representations the matrix is given as a block matrix. Each single irreducible can be obtained as item indexed by the members of the enum `AbsIrreducibleRep` available via `CubicHeckeAlgebra.irred_repr`. For details type: `CubicHeckeAlgebra.irred_repr?`.

EXAMPLES:

```
sage: CHA3 = algebras.CubicHecke(3)
sage: CHA3.inject_variables()
Defining c0, c1
sage: c0m = c0.matrix()
sage: c0m[CHA3.irred_repr.W3_111]
[
      -b - a + u      0      0]
[(-2*a + u)*b - 2*a^2 + 2*u*a - v      b      0]
[
                        b      1      a]
```

```
>>> from sage.all import *
>>> CHA3 = algebras.CubicHecke(Integer(3))
>>> CHA3.inject_variables()
Defining c0, c1
>>> c0m = c0.matrix()
>>> c0m[CHA3.irred_repr.W3_111]
[
      -b - a + u      0      0]
[(-2*a + u)*b - 2*a^2 + 2*u*a - v      b      0]
[
                        b      1      a]
```

using the `representation_type` option:

```
sage: CHA3.<c0, c1> = algebras.CubicHecke(3) # optional gap3
sage: chevie = CHA3.repr_type.SplitIrredChevie # optional gap3
sage: c0m_ch = c0.matrix(representation_type=chevie) # optional gap3
sage: c0m_ch[CHA3.irred_repr.W3_011] # optional gap3
[
      b      0]
[
    -b -b - a + u]
sage: c0m[CHA3.irred_repr.W3_011]
[
      b      0]
[a^2 - u*a + v      -b - a + u]
```

```
>>> from sage.all import *
>>> CHA3 = algebras.CubicHecke(Integer(3), names=('c0', 'c1',)); (c0, c1,) = CHA3._first_ngens(2) # optional gap3
>>> chevie = CHA3.repr_type.SplitIrredChevie # optional gap3
>>> c0m_ch = c0.matrix(representation_type=chevie) # optional gap3
>>> c0m_ch[CHA3.irred_repr.W3_011] # optional gap3
[
      b      0]
[
    -b -b - a + u]
>>> c0m[CHA3.irred_repr.W3_011]
```

(continues on next page)

(continued from previous page)

```
[      b      0]
[a^2 - u*a + v  -b - a + u]
```

using the original option:

```
sage: c0mo = c0.matrix(original=True)
sage: c0mo_ch = c0.matrix(representation_type=chevie, original=True) # ↵
↪optional gap3
sage: c0mo[CHA3.irred_repr.W3_011]
[  b  0]
[b*c  c]
sage: c0mo_ch[CHA3.irred_repr.W3_011] # optional gap3
[  b  0]
[-b  c]
```

```
>>> from sage.all import *
>>> c0mo = c0.matrix(original=True)
>>> c0mo_ch = c0.matrix(representation_type=chevie, original=True) # ↵
↪optional gap3
>>> c0mo[CHA3.irred_repr.W3_011]
[  b  0]
[b*c  c]
>>> c0mo_ch[CHA3.irred_repr.W3_011] # optional gap3
[  b  0]
[-b  c]
```

specialized matrices:

```
sage: t = (3,7,11)
sage: CHA4 = algebras.CubicHecke(4, cubic_equation_roots=t) # optional ↵
↪database_cubic_hecke
sage: e = CHA4.an_element(); e # optional database_cubic_
↪hecke
-231*c0*c1^-1 + 131*c0*c2^-1 + 21*c2*c1 - 1440/11
sage: em = e.matrix() # optional database_cubic_
↪hecke
sage: em.base_ring() # optional database_cubic_
↪hecke
Splitting Algebra of T^2 + T + 1 with roots [E3, -E3 - 1]
over Integer Ring localized at (3, 7, 11)
sage: em.dimensions() # optional database_cubic_
↪hecke
(108, 108)
sage: em_irr24 = em[23] # optional database_cubic_
↪hecke
sage: em_irr24.dimensions() # optional database_cubic_
↪hecke
(9, 9)
sage: em_irr24[3,2] # optional database_cubic_
↪hecke
-131*E3 - 393/7
sage: emg = e.matrix(representation_type=chevie) # optional gap3 database_
```

(continues on next page)

(continued from previous page)

```

↪cubic_hecke
sage: emg_irr24 = emg[23] # optional gap3 database_
↪cubic_hecke
sage: emg_irr24[3,2] # optional gap3 database_
↪cubic_hecke
-131*E3 - 393/7

```

```

>>> from sage.all import *
>>> t = (Integer(3), Integer(7), Integer(11))
>>> CHA4 = algebras.CubicHecke(Integer(4), cubic_equation_roots=t) #_
↪optional_database_cubic_hecke
>>> e = CHA4.an_element(); e # optional_database_cubic_
↪hecke
-231*c0*c1^-1 + 131*c0*c2^-1 + 21*c2*c1 - 1440/11
>>> em = e.matrix() # optional_database_cubic_
↪hecke
>>> em.base_ring() # optional_database_cubic_
↪hecke
Splitting Algebra of T^2 + T + 1 with roots [E3, -E3 - 1]
over Integer Ring localized at (3, 7, 11)
>>> em.dimensions() # optional_database_cubic_
↪hecke
(108, 108)
>>> em_irr24 = em[Integer(23)] # optional_database_
↪cubic_hecke
>>> em_irr24.dimensions() # optional_database_cubic_
↪hecke
(9, 9)
>>> em_irr24[Integer(3), Integer(2)] # optional_
↪database_cubic_hecke
-131*E3 - 393/7
>>> emg = e.matrix(representation_type=chevie) # optional_gap3_database_
↪cubic_hecke
>>> emg_irr24 = emg[Integer(23)] # optional_gap3_
↪database_cubic_hecke
>>> emg_irr24[Integer(3), Integer(2)] # optional_
↪gap3_database_cubic_hecke
-131*E3 - 393/7

```

max_len()

Return the maximum of the length of Tietze expressions among the support of self.

EXAMPLES:

```

sage: CHA3 = algebras.CubicHecke(3)
sage: ele = CHA3.an_element(); ele
-w*c0*c1^-1 + v*c0 + u*c1 - ((v*w-u)/w)
sage: ele.max_len()
2

```

```

>>> from sage.all import *
>>> CHA3 = algebras.CubicHecke(Integer(3))

```

(continues on next page)

(continued from previous page)

```
>>> ele = CHA3.an_element(); ele
-w*c0*c1^-1 + v*c0 + u*c1 - ((v*w-u)/w)
>>> ele.max_len()
2
```

revert_garside()

Return the image of `self` under the Garside involution.

See also

`CubicHeckeAlgebra.garside_involution()`

EXAMPLES:

```
sage: roots = (E(3), ~E(3), 1)
sage: CHA3.<c1, c2> = algebras.CubicHecke(3, cubic_equation_roots=roots)
sage: e = CHA3.an_element(); e
-c1*c2^-1
sage: _.revert_garside()
-c2*c1^-1
sage: _.revert_garside()
-c1*c2^-1
```

```
>>> from sage.all import *
>>> roots = (E(Integer(3)), ~E(Integer(3)), Integer(1))
>>> CHA3 = algebras.CubicHecke(Integer(3), cubic_equation_roots=roots, names=(
↳ 'c1', 'c2',)); (c1, c2,) = CHA3._first_ngens(2)
>>> e = CHA3.an_element(); e
-c1*c2^-1
>>> _.revert_garside()
-c2*c1^-1
>>> _.revert_garside()
-c1*c2^-1
```

revert_mirror()

Return the image of `self` under the mirror isomorphism.

See also

`CubicHeckeAlgebra.mirror_isomorphism()`

EXAMPLES:

```
sage: CHA3.<c1, c2> = algebras.CubicHecke(3)
sage: e = CHA3.an_element()
sage: e.revert_mirror()
-1/w*c0^-1*c1 + u/w*c0^-1 + v/w*c1^-1 + ((v*w-u)/w)
sage: _.revert_mirror() == e
True
```

```

>>> from sage.all import *
>>> CHA3 = algebras.CubicHecke(Integer(3), names=('c1', 'c2')); (c1, c2,) = _
↳CHA3._first_ngens(2)
>>> e = CHA3.an_element()
>>> e.revert_mirror()
-1/w*c0^-1*c1 + u/w*c0^-1 + v/w*c1^-1 + ((v*w-u)/w)
>>> _.revert_mirror() == e
True

```

revert_orientation()

Return the image of self under the anti involution reverting the orientation of braids.

See also

CubicHeckeAlgebra.orientation_antiinvolution()

EXAMPLES:

```

sage: CHA3.<c1, c2> = algebras.CubicHecke(3)
sage: e = CHA3.an_element()
sage: e.revert_orientation()
-w*c2^-1*c1 + v*c1 + u*c2 - ((v*w-u)/w)
sage: _.revert_orientation() == e
True

```

```

>>> from sage.all import *
>>> CHA3 = algebras.CubicHecke(Integer(3), names=('c1', 'c2')); (c1, c2,) = _
↳CHA3._first_ngens(2)
>>> e = CHA3.an_element()
>>> e.revert_orientation()
-w*c2^-1*c1 + v*c1 + u*c2 - ((v*w-u)/w)
>>> _.revert_orientation() == e
True

```

6.6.2 Cubic Hecke Base Rings

This module contains special classes of polynomial rings (*CubicHeckeRingOfDefinition* and *CubicHeckeExtensionRing*) used in the context of *cubic Hecke algebras*.

AUTHORS:

- Sebastian Oehms May 2020: initial version

```

class sage.algebras.hecke_algebras.cubic_hecke_base_ring.CubicHeckeExtensionRing(names,
or-
der='de-
grevlex',
ring_of_def-
ini-
tion=None,
third_unity_root_name='e3',
markov_trace_ver-
sion=False)

```

Bases: `LaurentPolynomialRing_mpair`

The generic splitting algebra for the irreducible representations of the cubic Hecke algebra.

This ring must contain three invertible indeterminates (representing the roots of the cubic equation) together with a third root of unity (needed for the 18-dimensional irreducibles of the cubic Hecke algebra on 4 strands).

Therefore this ring is constructed as a multivariate Laurent polynomial ring in three indeterminates over a polynomial quotient ring over the integers with respect to the minimal polynomial of a third root of unity.

The polynomial quotient ring is constructed as instance of `SplittingAlgebra`.

INPUT:

- `names` – (default: `'u,v,w'`) string containing the names of the indeterminates separated by `,` or a triple of strings each of which are the names of one of the three indeterminates
- `order` – string (default: `'degrevlex'`); the term order; see also `LaurentPolynomialRing_mpair`
- `ring_of_definition` – (optional) a `CubicHeckeRingOfDefinition` to specify the generic cubic Hecke base ring over which `self` may be realized as splitting ring via the `as_splitting_algebra` method
- `third_unity_root_name` – string (default: `'e3'`); for setting the name of the third root of unity of `self`
- `markov_trace_version` – boolean (default: `False`); if this is set to `True` then `self` contains one invertible indeterminate in addition which is meant to represent the writhe factor of a Markov trace on the cubic Hecke algebra and which default name is `s`

EXAMPLES:

```
sage: from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
sage: chbr.CubicHeckeExtensionRing('a, b, c')
Multivariate Laurent Polynomial Ring in a, b, c
over Splitting Algebra of x^2 + x + 1
with roots [e3, -e3 - 1]
over Integer Ring
sage: _.an_element()
b^2*c^-1 + e3*a
```

```
>>> from sage.all import *
>>> from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
>>> chbr.CubicHeckeExtensionRing('a, b, c')
Multivariate Laurent Polynomial Ring in a, b, c
over Splitting Algebra of x^2 + x + 1
with roots [e3, -e3 - 1]
over Integer Ring
>>> _.an_element()
b^2*c^-1 + e3*a
```

`as_splitting_algebra()`

Return `self` as a `SplittingAlgebra`; that is as an extension ring of the corresponding cubic Hecke algebra base ring (`self._ring_of_definition`, as a `CubicHeckeRingOfDefinition`) splitting its cubic equation into linear factors, such that the roots are images of the generators of `self`.

EXAMPLES:

```
sage: from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
sage: GBR = chbr.CubicHeckeRingOfDefinition()
```

(continues on next page)

(continued from previous page)

```

sage: GER = GBR.extension_ring()
sage: ER = GER.as_splitting_algebra(); ER
Splitting Algebra of  $T^2 + T + 1$  with roots  $[E3, -E3 - 1]$ 
  over Splitting Algebra of  $h^3 - u*h^2 + v*h - w$ 
    with roots  $[a, b, -b - a + u]$ 
  over Multivariate Polynomial Ring in  $u, v, w$ 
  over Integer Ring localized at  $(w,)$ 
sage: ER(GER.an_element())
a*E3 + ((u/(-w))*a^2 + ((u^2 - v)/w)*a)*b + a - u
sage: ER(GBR.an_element())
(u^2 + v*w)/w

sage: MBR = chbr.CubicHeckeRingOfDefinition(markov_trace_version=True)
sage: MER = MBR.extension_ring()
sage: ES = MER.as_splitting_algebra(); ES
Splitting Algebra of  $T^2 + T + 1$  with roots  $[E3, -E3 - 1]$ 
  over Splitting Algebra of  $h^3 - u*h^2 + v*h - w$ 
    with roots  $[a, b, -b - a + u]$ 
  over Multivariate Polynomial Ring in  $u, v, w, s$ 
  over Integer Ring localized at  $(s, w, v, u)$ 
sage: ES(MER.an_element())
((-1)/(-s))*a*E3 + ((u/(-w))*a^2 + ((u^2 - v)/w)*a)*b + a - u
sage: ES(MBR.an_element())
(u^2*s + v*w)/(w*s)

```

```

>>> from sage.all import *
>>> from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
>>> GBR = chbr.CubicHeckeRingOfDefinition()
>>> GER = GBR.extension_ring()
>>> ER = GER.as_splitting_algebra(); ER
Splitting Algebra of  $T^2 + T + 1$  with roots  $[E3, -E3 - 1]$ 
  over Splitting Algebra of  $h^3 - u*h^2 + v*h - w$ 
    with roots  $[a, b, -b - a + u]$ 
  over Multivariate Polynomial Ring in  $u, v, w$ 
  over Integer Ring localized at  $(w,)$ 
>>> ER(GER.an_element())
a*E3 + ((u/(-w))*a^2 + ((u^2 - v)/w)*a)*b + a - u
>>> ER(GBR.an_element())
(u^2 + v*w)/w

>>> MBR = chbr.CubicHeckeRingOfDefinition(markov_trace_version=True)
>>> MER = MBR.extension_ring()
>>> ES = MER.as_splitting_algebra(); ES
Splitting Algebra of  $T^2 + T + 1$  with roots  $[E3, -E3 - 1]$ 
  over Splitting Algebra of  $h^3 - u*h^2 + v*h - w$ 
    with roots  $[a, b, -b - a + u]$ 
  over Multivariate Polynomial Ring in  $u, v, w, s$ 
  over Integer Ring localized at  $(s, w, v, u)$ 
>>> ES(MER.an_element())
((-1)/(-s))*a*E3 + ((u/(-w))*a^2 + ((u^2 - v)/w)*a)*b + a - u
>>> ES(MBR.an_element())
(u^2*s + v*w)/(w*s)

```

conjugation()

Return an involution that performs *complex conjugation* with respect to base ring considered as order in the complex field.

EXAMPLES:

```
sage: from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
sage: ER = chbr.CubicHeckeExtensionRing('x, y, z')
sage: conj = ER.conjugation()
sage: conj(ER.an_element())
y^2*z^-1 + (-e3 - 1)*x
sage: MER = chbr.CubicHeckeExtensionRing('x, y, z, s', markov_trace_
↳version=True)
sage: conj = MER.conjugation()
sage: conj(MER.an_element())
y^2*z^-1 + (-e3 - 1)*x*s^-1
```

```
>>> from sage.all import *
>>> from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
>>> ER = chbr.CubicHeckeExtensionRing('x, y, z')
>>> conj = ER.conjugation()
>>> conj(ER.an_element())
y^2*z^-1 + (-e3 - 1)*x
>>> MER = chbr.CubicHeckeExtensionRing('x, y, z, s', markov_trace_
↳version=True)
>>> conj = MER.conjugation()
>>> conj(MER.an_element())
y^2*z^-1 + (-e3 - 1)*x*s^-1
```

construction()

Return None since this construction is not functorial.

EXAMPLES:

```
sage: from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
sage: ER = chbr.CubicHeckeExtensionRing('a, b, c')
sage: ER._test_category() # indirect doctest
```

```
>>> from sage.all import *
>>> from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
>>> ER = chbr.CubicHeckeExtensionRing('a, b, c')
>>> ER._test_category() # indirect doctest
```

create_specialization(*im_cubic_equation_roots*, *im_writhe_parameter*=None, *var*='T',
third_unity_root_name='E3')

Return an appropriate ring containing the elements from the list *im_cubic_equation_roots* defining a conversion map from *self* mapping the cubic equation roots of *self* to *im_cubic_equation_roots*.

INPUT:

- *im_cubic_equation_roots* – list or tuple of three ring elements such that there exists a ring homomorphism from the corresponding elements of *self* to them

OUTPUT:

A common parent containing the elements of *im_cubic_equation_roots* together with their inverses.

EXAMPLES:

```

sage: from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
sage: ER = chbr.CubicHeckeExtensionRing('a, b, c')
sage: t = ER.an_element(); t
b^2*c^-1 + e3*a
sage: Sp1 = ER.create_specialization([E(5), E(7), E(3)]); Sp1
Universal Cyclotomic Field
sage: Sp1(t)
-E(105)^11 - E(105)^16 - E(105)^26 - E(105)^37 - E(105)^41
- E(105)^58 - E(105)^71 - E(105)^79 - E(105)^86 - E(105)^101
sage: MER = chbr.CubicHeckeExtensionRing('a, b, c, s', markov_trace_
↳version=True)
sage: MER.create_specialization([E(5), E(7), E(3)], im_writhe_parameter=E(4))
Universal Cyclotomic Field
sage: a, b, c, s = MER.gens()
sage: Sp1(MER(t)/s)
E(420) + E(420)^29 + E(420)^89 + E(420)^149 + E(420)^169 + E(420)^209
+ E(420)^253 + E(420)^269 + E(420)^337 + E(420)^389

sage: Z3 = CyclotomicField(3); E3=Z3.gen()
sage: Sp2 = ER.create_specialization([E3, E3**2, Z3(1)])
sage: Sp2(t)
-1
sage: MER.create_specialization([E3, E3**2, 1], im_writhe_parameter=2)
Cyclotomic Field of order 3 and degree 2
sage: Sp2(MER(t)*s)
-2

sage: Sp3 = ER.create_specialization([5, 7, 11])
sage: Sp3(t)
5*E3 + 49/11

```

```

>>> from sage.all import *
>>> from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
>>> ER = chbr.CubicHeckeExtensionRing('a, b, c')
>>> t = ER.an_element(); t
b^2*c^-1 + e3*a
>>> Sp1 = ER.create_specialization([E(Integer(5)), E(Integer(7)),
↳E(Integer(3))]); Sp1
Universal Cyclotomic Field
>>> Sp1(t)
-E(105)^11 - E(105)^16 - E(105)^26 - E(105)^37 - E(105)^41
- E(105)^58 - E(105)^71 - E(105)^79 - E(105)^86 - E(105)^101
>>> MER = chbr.CubicHeckeExtensionRing('a, b, c, s', markov_trace_
↳version=True)
>>> MER.create_specialization([E(Integer(5)), E(Integer(7)), E(Integer(3))],
↳im_writhe_parameter=E(Integer(4)))
Universal Cyclotomic Field
>>> a, b, c, s = MER.gens()
>>> Sp1(MER(t)/s)
E(420) + E(420)^29 + E(420)^89 + E(420)^149 + E(420)^169 + E(420)^209
+ E(420)^253 + E(420)^269 + E(420)^337 + E(420)^389

```

(continues on next page)

(continued from previous page)

```

>>> Z3 = CyclotomicField(Integer(3)); E3=Z3.gen()
>>> Sp2 = ER.create_specialization([E3, E3**Integer(2), Z3(Integer(1))])
>>> Sp2(t)
-1
>>> MER.create_specialization([E3, E3**Integer(2), Integer(1)], im_writhe_
↳parameter=Integer(2))
Cyclotomic Field of order 3 and degree 2
>>> Sp2(MER(t)*s)
-2

>>> Sp3 = ER.create_specialization([Integer(5), Integer(7), Integer(11)])
>>> Sp3(t)
5*E3 + 49/11

```

cubic_equation_galois_group()

Return the Galois group of the cubic equation, which is the permutation group on the three generators together with its action on `self`.

EXAMPLES:

```

sage: from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
sage: ER = chbr.CubicHeckeExtensionRing('a, b, c')
sage: G = ER.cubic_equation_galois_group()
sage: t = ER.an_element()
sage: [(g, g*t) for g in G]
[((), b^2*c^-1 + e3*a),
 (1,3,2), a^2*b^-1 + e3*c),
 (1,2,3), e3*b + a^-1*c^2),
 (2,3), e3*a + b^-1*c^2),
 (1,3), a^-1*b^2 + e3*c),
 (1,2), a^2*c^-1 + e3*b)]

```

```

>>> from sage.all import *
>>> from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
>>> ER = chbr.CubicHeckeExtensionRing('a, b, c')
>>> G = ER.cubic_equation_galois_group()
>>> t = ER.an_element()
>>> [(g, g*t) for g in G]
[((), b^2*c^-1 + e3*a),
 (1,3,2), a^2*b^-1 + e3*c),
 (1,2,3), e3*b + a^-1*c^2),
 (2,3), e3*a + b^-1*c^2),
 (1,3), a^-1*b^2 + e3*c),
 (1,2), a^2*c^-1 + e3*b)]

```

cyclotomic_generator()

Return the third root of unity as generator of the base ring of `self`.

EXAMPLES:

```

sage: from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
sage: ER = chbr.CubicHeckeExtensionRing('a, b, c')

```

(continues on next page)

(continued from previous page)

```
sage: ER.cyclotomic_generator()
e3
sage: _**3 == 1
True
```

```
>>> from sage.all import *
>>> from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
>>> ER = chbr.CubicHeckeExtensionRing('a, b, c')
>>> ER.cyclotomic_generator()
e3
>>> _**Integer(3) == Integer(1)
True
```

field_embedding(characteristic=0)

Return a field embedding of self.

INPUT:

- characteristic – integer (default: 0); the characteristic of the field

EXAMPLES:

```
sage: from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
sage: BR = chbr.CubicHeckeRingOfDefinition()
sage: ER = BR.extension_ring()
sage: ER.field_embedding()
Ring morphism:
From: Multivariate Laurent Polynomial Ring in a, b, c
      over Splitting Algebra of x^2 + x + 1
      with roots [e3, -e3 - 1]
      over Integer Ring
To:    Fraction Field of Multivariate Polynomial Ring in a, b, c
      over Cyclotomic Field of order 3 and degree 2
Defn: a |--> a
      b |--> b
      c |--> c
with map of base ring

sage: ER.field_embedding(characteristic=5)
Ring morphism:
From: Multivariate Laurent Polynomial Ring in a, b, c
      over Splitting Algebra of x^2 + x + 1
      with roots [e3, -e3 - 1]
      over Integer Ring
To:    Fraction Field of Multivariate Polynomial Ring in a, b, c
      over Finite Field in a of size 5^2
Defn: a |--> a
      b |--> b
      c |--> c
with map of base ring

sage: MER = ER.markov_trace_version()
sage: MER.field_embedding()
```

(continues on next page)

(continued from previous page)

```

Ring morphism:
From: Multivariate Laurent Polynomial Ring in a, b, c, s
      over Splitting Algebra of  $x^2 + x + 1$ 
      with roots [e3, -e3 - 1]
      over Integer Ring
To:   Fraction Field of Multivariate Polynomial Ring in a, b, c, s
      over Cyclotomic Field of order 3 and degree 2
Defn: a |--> a
      b |--> b
      c |--> c
      s |--> s
with map of base ring

```

```

>>> from sage.all import *
>>> from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
>>> BR = chbr.CubicHeckeRingOfDefinition()
>>> ER = BR.extension_ring()
>>> ER.field_embedding()
Ring morphism:
From: Multivariate Laurent Polynomial Ring in a, b, c
      over Splitting Algebra of  $x^2 + x + 1$ 
      with roots [e3, -e3 - 1]
      over Integer Ring
To:   Fraction Field of Multivariate Polynomial Ring in a, b, c
      over Cyclotomic Field of order 3 and degree 2
Defn: a |--> a
      b |--> b
      c |--> c
with map of base ring

>>> ER.field_embedding(characteristic=Integer(5))
Ring morphism:
From: Multivariate Laurent Polynomial Ring in a, b, c
      over Splitting Algebra of  $x^2 + x + 1$ 
      with roots [e3, -e3 - 1]
      over Integer Ring
To:   Fraction Field of Multivariate Polynomial Ring in a, b, c
      over Finite Field in a of size  $5^2$ 
Defn: a |--> a
      b |--> b
      c |--> c
with map of base ring

>>> MER = ER.markov_trace_version()
>>> MER.field_embedding()
Ring morphism:
From: Multivariate Laurent Polynomial Ring in a, b, c, s
      over Splitting Algebra of  $x^2 + x + 1$ 
      with roots [e3, -e3 - 1]
      over Integer Ring
To:   Fraction Field of Multivariate Polynomial Ring in a, b, c, s
      over Cyclotomic Field of order 3 and degree 2

```

(continues on next page)

(continued from previous page)

```

Defn: a |--> a
      b |--> b
      c |--> c
      s |--> s
with map of base ring

```

hom(*im_gens*, *codomain=None*, *check=True*, *base_map=None*)

Return a homomorphism of *self*.

INPUT:

- *im_gens* – tuple for the image of the generators of *self*
- *codomain* – (optional) the codomain of the homomorphism

EXAMPLES:

```

sage: from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
sage: ER = chbr.CubicHeckeExtensionRing('a, b, c')
sage: UCF = UniversalCyclotomicField()
sage: map = ER.hom((UCF.gen(3),) + (UCF(3), UCF(4), UCF(5)))
sage: ER.an_element()
b^2*c^-1 + e3*a
sage: map(_)
-1/5*E(3) - 16/5*E(3)^2

```

```

>>> from sage.all import *
>>> from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
>>> ER = chbr.CubicHeckeExtensionRing('a, b, c')
>>> UCF = UniversalCyclotomicField()
>>> map = ER.hom((UCF.gen(Integer(3)),) + (UCF(Integer(3)), UCF(Integer(4)),
↳ UCF(Integer(5))))
>>> ER.an_element()
b^2*c^-1 + e3*a
>>> map(_)
-1/5*E(3) - 16/5*E(3)^2

```

markov_trace_version()

Return the Markov trace version of *self*.

EXAMPLES:

```

sage: from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
sage: ER = chbr.CubicHeckeExtensionRing('a, b, c')
sage: ER.markov_trace_version()
Multivariate Laurent Polynomial Ring in a, b, c, s
over Splitting Algebra of x^2 + x + 1
with roots [e3, -e3 - 1] over Integer Ring

```

```

>>> from sage.all import *
>>> from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
>>> ER = chbr.CubicHeckeExtensionRing('a, b, c')
>>> ER.markov_trace_version()
Multivariate Laurent Polynomial Ring in a, b, c, s

```

(continues on next page)

(continued from previous page)

```
over Splitting Algebra of  $x^2 + x + 1$ 
with roots  $[e3, -e3 - 1]$  over Integer Ring
```

mirror_involution()

Return the involution of `self` corresponding to the involution of the cubic Hecke algebra (with the same name).

This means that it maps the generators of `self` to their inverses.

Note

The mirror involution of the braid group does not factor through the cubic Hecke algebra over its base ring, but it does if it is considered as $\mathbf{Z}$ -algebra. The base ring elements are transformed by this automorphism.

OUTPUT: the involution as automorphism of `self`

EXAMPLES:

```
sage: from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
sage: ER = chbr.CubicHeckeExtensionRing('p, q, r')
sage: ER.mirror_involution()
Ring endomorphism of Multivariate Laurent Polynomial Ring in p, q, r
      over Splitting Algebra of  $x^2 + x + 1$ 
      with roots  $[e3, -e3 - 1]$ 
      over Integer Ring

Defn: p |--> p^-1
      q |--> q^-1
      r |--> r^-1
      with map of base ring
sage: _(ER.an_element())
e3*p^-1 + q^-2*r

sage: MER = chbr.CubicHeckeExtensionRing('p, q, r, s', markov_trace_
      ↪version=True)
sage: MER.mirror_involution()
Ring endomorphism of Multivariate Laurent Polynomial Ring in p, q, r, s
      over Splitting Algebra of  $x^2 + x + 1$ 
      with roots  $[e3, -e3 - 1]$  over Integer Ring
Defn: p |--> p^-1
      q |--> q^-1
      r |--> r^-1
      s |--> s^-1
      with map of base ring
sage: _(MER.an_element())
e3*p^-1*s + q^-2*r
```

```
>>> from sage.all import *
>>> from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
>>> ER = chbr.CubicHeckeExtensionRing('p, q, r')
>>> ER.mirror_involution()
Ring endomorphism of Multivariate Laurent Polynomial Ring in p, q, r
      over Splitting Algebra of  $x^2 + x + 1$ 
```

(continues on next page)

(continued from previous page)

```

                with roots [e3, -e3 - 1]
                over Integer Ring
Defn: p |--> p^-1
      q |--> q^-1
      r |--> r^-1
      with map of base ring
>>> _(ER.an_element())
e3*p^-1 + q^-2*r

>>> MER = chbr.CubicHeckeExtensionRing('p, q, r, s', markov_trace_
↳version=True)
>>> MER.mirror_involution()
Ring endomorphism of Multivariate Laurent Polynomial Ring in p, q, r, s
  over Splitting Algebra of x^2 + x + 1
    with roots [e3, -e3 - 1] over Integer Ring
Defn: p |--> p^-1
      q |--> q^-1
      r |--> r^-1
      s |--> s^-1
      with map of base ring
>>> _(MER.an_element())
e3*p^-1*s + q^-2*r

```

```

class sage.algebras.hecke_algebras.cubic_hecke_base_ring.CubicHeckeRingOfDefinition (names=('u',
                                                                                               'v',
                                                                                               'w',
                                                                                               's'),
                                                                                               or-
                                                                                               der='de-
                                                                                               grevlex',
                                                                                               markov_trace_ver-
                                                                                               sion=False)

```

Bases: `Localization`

The *ring of definition* of the cubic Hecke algebra.

It contains one invertible indeterminate (representing the product of the roots of the cubic equation) and two non invertible indeterminates.

Note

We follow a suggestion by Ivan Marin in the name *ring of definition*. We avoid alternative names like *generic* or *universal* base ring as these have some issues. The first option could be misleading in view of the term *generic point* used in algebraic geometry, which would mean the function field in u, v, w , here.

The second option is problematic since the base ring itself is not a universal object. Rather, the universal object is the cubic Hecke algebra considered as a $\mathbb{Z}$ -algebra including u, v, w as pairwise commuting indeterminates. From this point of view the base ring appears to be a subalgebra of this universal object generated by u, v, w .

INPUT:

- `names` – (default: `'u, v, w'`) string containing the names of the indeterminates separated by `,` or a triple of strings each of which are the names of one of the three indeterminates

- `order` – string (default: 'degrevlex'); the term order; see also `LaurentPolynomialRing_mpair`
- `ring_of_definition` – (optional) a `CubicHeckeRingOfDefinition` to specify the generic cubic Hecke base ring over which `self` may be realized as splitting ring via the `as_splitting_algebra` method
- `markov_trace_version` – boolean (default: False); if this is set to True then `self` contains one invertible indeterminate in addition which is meant to represent the writhe factor of a Markov trace on the cubic Hecke algebra and which default name is `s`

EXAMPLES:

```
sage: from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
sage: BR = chbr.CubicHeckeRingOfDefinition()
sage: u, v, w = BR.gens()
sage: ele = 3*u*v-5*w**(-2)
sage: ER = BR.extension_ring()
sage: ER(ele)
3*a^2*b + 3*a*b^2 + 3*a^2*c + 9*a*b*c + 3*b^2*c
+ 3*a*c^2 + 3*b*c^2 + (-5)*a^-2*b^-2*c^-2
sage: phi1 = BR.hom( [4,3,1/1] )
sage: phi1(ele)
31

sage: LL.<t> = LaurentPolynomialRing(ZZ)
sage: phi2=BR.hom( [LL(4),LL(3),t] )
sage: phi2(ele)
-5*t^-2 + 36

sage: BR.create_specialization( [E(5), E(7), E(3)] )
Universal Cyclotomic Field
sage: _(ele)
-3*E(105) - 5*E(105)^2 - 5*E(105)^8 - 5*E(105)^11 - 5*E(105)^17
- 5*E(105)^23 - 5*E(105)^26 - 5*E(105)^29 - 5*E(105)^32 - 5*E(105)^38
- 5*E(105)^41 - 5*E(105)^44 - 5*E(105)^47 - 5*E(105)^53 - 5*E(105)^59
- 5*E(105)^62 - 5*E(105)^68 - 8*E(105)^71 - 5*E(105)^74 - 5*E(105)^83
- 5*E(105)^86 - 5*E(105)^89 - 5*E(105)^92 - 5*E(105)^101 - 5*E(105)^104
```

```
>>> from sage.all import *
>>> from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
>>> BR = chbr.CubicHeckeRingOfDefinition()
>>> u, v, w = BR.gens()
>>> ele = Integer(3)*u*v-Integer(5)*w**(-Integer(2))
>>> ER = BR.extension_ring()
>>> ER(ele)
3*a^2*b + 3*a*b^2 + 3*a^2*c + 9*a*b*c + 3*b^2*c
+ 3*a*c^2 + 3*b*c^2 + (-5)*a^-2*b^-2*c^-2
>>> phi1 = BR.hom( [Integer(4),Integer(3),Integer(1)/Integer(1)] )
>>> phi1(ele)
31

>>> LL = LaurentPolynomialRing(ZZ, names=('t',)); (t,) = LL._first_ngens(1)
>>> phi2=BR.hom( [LL(Integer(4)),LL(Integer(3)),t] )
>>> phi2(ele)
-5*t^-2 + 36
```

(continues on next page)

(continued from previous page)

```

>>> BR.create_specialization( [E(Integer(5)), E(Integer(7)), E(Integer(3))] )
Universal Cyclotomic Field
>>> _(ele)
-3*E(105) - 5*E(105)^2 - 5*E(105)^8 - 5*E(105)^11 - 5*E(105)^17
- 5*E(105)^23 - 5*E(105)^26 - 5*E(105)^29 - 5*E(105)^32 - 5*E(105)^38
- 5*E(105)^41 - 5*E(105)^44 - 5*E(105)^47 - 5*E(105)^53 - 5*E(105)^59
- 5*E(105)^62 - 5*E(105)^68 - 8*E(105)^71 - 5*E(105)^74 - 5*E(105)^83
- 5*E(105)^86 - 5*E(105)^89 - 5*E(105)^92 - 5*E(105)^101 - 5*E(105)^104

```

create_specialization(*im_cubic_equation_parameters*, *im_writhe_parameter*=None)

Return an appropriate Ring containing the elements from the list *im_cubic_equation_parameters* having a conversion map from self mapping the cubic equation parameters of self to *im_cubic_equation_parameters*.

INPUT:

- *im_cubic_equation_parameters* – list or tuple of three ring elements such that there exists a ring homomorphism from the corresponding elements of self to them

OUTPUT:

A common parent containing the elements of *im_cubic_equation_parameters* together with an inverse of the third element.

EXAMPLES:

```

sage: from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
sage: BR = chbr.CubicHeckeRingOfDefinition()
sage: t = BR.an_element(); t
(u^2 + v*w)/w
sage: Sp1 = BR.create_specialization([E(5), E(7), E(3)]); Sp1
Universal Cyclotomic Field
sage: Sp1(t)
E(105) + E(105)^8 + E(105)^29 - E(105)^37 + E(105)^43 - E(105)^52
+ E(105)^64 - E(105)^67 + E(105)^71 - E(105)^82 + E(105)^92
- E(105)^97

sage: MBR = chbr.CubicHeckeRingOfDefinition(markov_trace_version=True)
sage: MBR.create_specialization([E(5), E(7), E(3)], im_writhe_parameter=E(4))
Universal Cyclotomic Field
sage: u, v, w, s = MBR.gens()
sage: Sp1(MBR(t)/s)
E(420)^13 - E(420)^53 + E(420)^73 - E(420)^109 - E(420)^137
- E(420)^221 + E(420)^253 - E(420)^277 + E(420)^313 - E(420)^361
+ E(420)^373 - E(420)^389

sage: Z3 = CyclotomicField(3); E3=Z3.gen()
sage: Sp2 = BR.create_specialization([E3, E3**2, 1]); Sp2
Cyclotomic Field of order 3 and degree 2
sage: Sp2(t)
-2*zeta3 - 2
sage: MBR.create_specialization([E3, E3**2, 1], im_writhe_parameter=2)
Cyclotomic Field of order 3 and degree 2
sage: Sp2(MBR(t)/s)

```

(continues on next page)

(continued from previous page)

```
-zeta3 - 1
```

```
sage: Sp3 = BR.create_specialization([5, 7, 11]); Sp3
Integer Ring localized at (11,)
sage: Sp3(t)
102/11
```

```
>>> from sage.all import *
>>> from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
>>> BR = chbr.CubicHeckeRingOfDefinition()
>>> t = BR.an_element(); t
(u^2 + v*w)/w
>>> Sp1 = BR.create_specialization([E(Integer(5)), E(Integer(7)),
↳ E(Integer(3))]); Sp1
Universal Cyclotomic Field
>>> Sp1(t)
E(105) + E(105)^8 + E(105)^29 - E(105)^37 + E(105)^43 - E(105)^52
+ E(105)^64 - E(105)^67 + E(105)^71 - E(105)^82 + E(105)^92
- E(105)^97

>>> MBR = chbr.CubicHeckeRingOfDefinition(markov_trace_version=True)
>>> MBR.create_specialization([E(Integer(5)), E(Integer(7)), E(Integer(3))],
↳ im_writhe_parameter=E(Integer(4)))
Universal Cyclotomic Field
>>> u, v, w, s = MBR.gens()
>>> Sp1(MBR(t)/s)
E(420)^13 - E(420)^53 + E(420)^73 - E(420)^109 - E(420)^137
- E(420)^221 + E(420)^253 - E(420)^277 + E(420)^313 - E(420)^361
+ E(420)^373 - E(420)^389

>>> Z3 = CyclotomicField(Integer(3)); E3=Z3.gen()
>>> Sp2 = BR.create_specialization([E3, E3**Integer(2), Integer(1)]); Sp2
Cyclotomic Field of order 3 and degree 2
>>> Sp2(t)
-2*zeta3 - 2
>>> MBR.create_specialization([E3, E3**Integer(2), Integer(1)], im_writhe_
↳ parameter=Integer(2))
Cyclotomic Field of order 3 and degree 2
>>> Sp2(MBR(t)/s)
-zeta3 - 1

>>> Sp3 = BR.create_specialization([Integer(5), Integer(7), Integer(11)]); Sp3
Integer Ring localized at (11,)
>>> Sp3(t)
102/11
```

cubic_equation (var='h', as_coefficients=False)

Return the cubic equation over which the cubic Hecke algebra is defined.

EXAMPLES:

```
sage: from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
```

(continues on next page)

(continued from previous page)

```

sage: BR = chbr.CubicHeckeRingOfDefinition()
sage: BR.cubic_equation()
h^3 - u*h^2 + v*h - w
sage: BR.cubic_equation(var='t')
t^3 - u*t^2 + v*t - w
sage: BR.cubic_equation(as_coefficients=True)
[-w, v, -u, 1]

```

```

>>> from sage.all import *
>>> from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
>>> BR = chbr.CubicHeckeRingOfDefinition()
>>> BR.cubic_equation()
h^3 - u*h^2 + v*h - w
>>> BR.cubic_equation(var='t')
t^3 - u*t^2 + v*t - w
>>> BR.cubic_equation(as_coefficients=True)
[-w, v, -u, 1]

```

extension_ring(names=('a', 'b', 'c', 's'))

Return the generic extension ring attached to self.

EXAMPLES:

```

sage: from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
sage: BR = chbr.CubicHeckeRingOfDefinition()
sage: BR.extension_ring()
Multivariate Laurent Polynomial Ring in a, b, c
over Splitting Algebra of x^2 + x + 1
with roots [e3, -e3 - 1]
over Integer Ring

```

```

>>> from sage.all import *
>>> from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
>>> BR = chbr.CubicHeckeRingOfDefinition()
>>> BR.extension_ring()
Multivariate Laurent Polynomial Ring in a, b, c
over Splitting Algebra of x^2 + x + 1
with roots [e3, -e3 - 1]
over Integer Ring

```

markov_trace_version()

Return the extension of the ring of definition needed to treat the formal Markov traces.

This appends an additional variable s to measure the writhe of knots and makes u and v invertible.

EXAMPLES:

```

sage: from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
sage: GBR = chbr.CubicHeckeRingOfDefinition()
sage: GBR.markov_trace_version()
Multivariate Polynomial Ring in u, v, w, s
over Integer Ring localized at (s, w, v, u)

```

```
>>> from sage.all import *
>>> from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
>>> GBR = chbr.CubicHeckeRingOfDefinition()
>>> GBR.markov_trace_version()
Multivariate Polynomial Ring in u, v, w, s
over Integer Ring localized at (s, w, v, u)
```

mirror_involution()

Return the involution of `self` corresponding to the involution of the cubic Hecke algebra (with the same name).

This means that it maps the last generator of `self` to its inverse and both others to their product with the image of the former.

From the cubic equation for a braid generator β_i :

$$\beta_i^3 - u\beta_i^2 + v\beta_i - w = 0.$$

One deduces the following cubic equation for β_i^{-1} :

$$\beta_i^{-3} - \frac{v}{w}\beta_i^{-2} + \frac{u}{w}\beta_i^{-1} - \frac{1}{w} = 0.$$

Note

The mirror involution of the braid group does not factor through the cubic Hecke algebra over its base ring, but it does if it is considered as $\mathbf{Z}$ -algebra. The base ring elements are transformed by this automorphism.

OUTPUT: the involution as automorphism of `self`

EXAMPLES:

```
sage: from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
sage: BR = chbr.CubicHeckeRingOfDefinition()
sage: BR.mirror_involution()
Ring endomorphism of Multivariate Polynomial Ring in u, v, w
over Integer Ring localized at (w,)
Defn: u |--> v/w
      v |--> u/w
      w |--> 1/w
sage: _(BR.an_element())
(v^2 + u)/w

sage: MBR = chbr.CubicHeckeRingOfDefinition(markov_trace_version=True)
sage: MBR.mirror_involution()
Ring endomorphism of Multivariate Polynomial Ring in u, v, w, s
over Integer Ring localized at (s, w, v, u)
Defn: u |--> v/w
      v |--> u/w
      w |--> 1/w
      s |--> 1/s
sage: _(MBR.an_element())
(v^2 + u*s)/w
```

```

>>> from sage.all import *
>>> from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
>>> BR = chbr.CubicHeckeRingOfDefinition()
>>> BR.mirror_involution()
Ring endomorphism of Multivariate Polynomial Ring in u, v, w
over Integer Ring localized at (w,)
Defn: u |--> v/w
      v |--> u/w
      w |--> 1/w
>>> _(BR.an_element())
(v^2 + u)/w

>>> MBR = chbr.CubicHeckeRingOfDefinition(markov_trace_version=True)
>>> MBR.mirror_involution()
Ring endomorphism of Multivariate Polynomial Ring in u, v, w, s
over Integer Ring localized at (s, w, v, u)
Defn: u |--> v/w
      v |--> u/w
      w |--> 1/w
      s |--> 1/s
>>> _(MBR.an_element())
(v^2 + u*s)/w

```

specialize_homfly()

Return a map to the two variable Laurent polynomial ring that is the parent of the HOMFLY-PT polynomial.

EXAMPLES:

```

sage: from sage.knots.knotinfo import KnotInfo
sage: CHA2 = algebras.CubicHecke(2)
sage: K5_1 = KnotInfo.K5_1.link()
sage: br = CHA2(K5_1.braid())
sage: mt = br.formal_markov_trace()
sage: MT = mt.base_ring()
sage: f = MT.specialize_homfly(); f
Composite map:
  From: Multivariate Polynomial Ring in u, v, w, s over Integer Ring
        localized at (s, w, v, u)
  To:   Multivariate Laurent Polynomial Ring in L, M over Integer Ring
  Defn: Ring morphism:
        From: Multivariate Polynomial Ring in u, v, w, s
              over Integer Ring localized at (s, w, v, u)
        To:   Multivariate Polynomial Ring in L, M
              over Integer Ring localized at (M - 1, M, L)
        Defn: u |--> -M + 1
              v |--> -M + 1
              w |--> 1
              s |--> L
  then
    Conversion map:
    From: Multivariate Polynomial Ring in L, M
          over Integer Ring localized at (M - 1, M, L)
    To:   Multivariate Laurent Polynomial Ring in L, M

```

(continues on next page)

(continued from previous page)

```

        over Integer Ring
sage: sup = mt.support()
sage: h1 = sum(f(mt.coefficient(b)) * b.regular_homfly_polynomial() for b in_
↪sup)
sage: L, M = f.codomain().gens()
sage: h2 = K5_1.homfly_polynomial()
sage: h1*L**(-5) == h2 # since the writhe of K5_1 is 5
True

```

```

>>> from sage.all import *
>>> from sage.knots.knotinfo import KnotInfo
>>> CHA2 = algebras.CubicHecke(Integer(2))
>>> K5_1 = KnotInfo.K5_1.link()
>>> br = CHA2(K5_1.braid())
>>> mt = br.formal_markov_trace()
>>> MT = mt.base_ring()
>>> f = MT.specialize_homfly(); f
Composite map:
  From: Multivariate Polynomial Ring in u, v, w, s over Integer Ring
        localized at (s, w, v, u)
  To:   Multivariate Laurent Polynomial Ring in L, M over Integer Ring
  Defn: Ring morphism:
        From: Multivariate Polynomial Ring in u, v, w, s
              over Integer Ring localized at (s, w, v, u)
        To:   Multivariate Polynomial Ring in L, M
              over Integer Ring localized at (M - 1, M, L)
        Defn: u |--> -M + 1
              v |--> -M + 1
              w |--> 1
              s |--> L
  then
    Conversion map:
    From: Multivariate Polynomial Ring in L, M
          over Integer Ring localized at (M - 1, M, L)
    To:   Multivariate Laurent Polynomial Ring in L, M
          over Integer Ring
>>> sup = mt.support()
>>> h1 = sum(f(mt.coefficient(b)) * b.regular_homfly_polynomial() for b in_
↪sup)
>>> L, M = f.codomain().gens()
>>> h2 = K5_1.homfly_polynomial()
>>> h1*L**(-Integer(5)) == h2 # since the writhe of K5_1 is 5
True

```

specialize_kauffman()

Return a map to the two variable Laurent polynomial ring that is the parent of the Kauffman polynomial.

EXAMPLES:

```

sage: from sage.knots.knotinfo import KnotInfo
sage: CHA2 = algebras.CubicHecke(2)
sage: K5_1 = KnotInfo.K5_1.link()

```

(continues on next page)

(continued from previous page)

```

sage: br = CHA2(K5_1.braid())
sage: mt = br.formal_markov_trace()
sage: MT = mt.base_ring()
sage: f = MT.specialize_kauffman(); f
Composite map:
  From: Multivariate Polynomial Ring in u, v, w, s over Integer Ring
        localized at (s, w, v, u)
  To:   Multivariate Laurent Polynomial Ring in a, z over Integer Ring
  Defn: Ring morphism:
        From: Multivariate Polynomial Ring in u, v, w, s
              over Integer Ring localized at (s, w, v, u)
        To:   Multivariate Polynomial Ring in a, z
              over Integer Ring localized at (z, a, a + z, a*z + 1)
  Defn: u |--> (a*z + 1)/a
        v |--> (a + z)/a
        w |--> 1/a
        s |--> a
  then
    Conversion map:
    From: Multivariate Polynomial Ring in a, z over Integer Ring
          localized at (z, a, a + z, a*z + 1)
    To:   Multivariate Laurent Polynomial Ring in a, z
          over Integer Ring
sage: sup = mt.support()
sage: k1 = sum(f(mt.coefficient(b)) * b.regular_kauffman_polynomial() for b_
↪in sup)
sage: a, z = f.codomain().gens()
sage: k2 = KnotInfo.K5_1.kauffman_polynomial()
sage: k1*a**(-5) == k2 # since the writhe of K5_1 is 5
True

```

```

>>> from sage.all import *
>>> from sage.knots.knotinfo import KnotInfo
>>> CHA2 = algebras.CubicHecke(Integer(2))
>>> K5_1 = KnotInfo.K5_1.link()
>>> br = CHA2(K5_1.braid())
>>> mt = br.formal_markov_trace()
>>> MT = mt.base_ring()
>>> f = MT.specialize_kauffman(); f
Composite map:
  From: Multivariate Polynomial Ring in u, v, w, s over Integer Ring
        localized at (s, w, v, u)
  To:   Multivariate Laurent Polynomial Ring in a, z over Integer Ring
  Defn: Ring morphism:
        From: Multivariate Polynomial Ring in u, v, w, s
              over Integer Ring localized at (s, w, v, u)
        To:   Multivariate Polynomial Ring in a, z
              over Integer Ring localized at (z, a, a + z, a*z + 1)
  Defn: u |--> (a*z + 1)/a
        v |--> (a + z)/a
        w |--> 1/a
        s |--> a

```

(continues on next page)

(continued from previous page)

```

then
  Conversion map:
  From: Multivariate Polynomial Ring in a, z over Integer Ring
        localized at (z, a, a + z, a*z + 1)
  To:   Multivariate Laurent Polynomial Ring in a, z
        over Integer Ring
>>> sup = mt.support()
>>> k1 = sum(f(mt.coefficient(b)) * b.regular_kauffman_polynomial() for b in sup)
↪sup)
>>> a, z = f.codomain().gens()
>>> k2 = KnotInfo.K5_1.kauffman_polynomial()
>>> k1*a**(-Integer(5)) == k2 # since the writhe of K5_1 is 5
True

```

specialize_links_gould()

Return a map to the two variable Laurent polynomial ring that is the parent of the Links-Gould polynomial.

EXAMPLES:

```

sage: from sage.knots.knotinfo import KnotInfo
sage: CHA2 = algebras.CubicHecke(2)
sage: K5_1 = KnotInfo.K5_1.link()
sage: br = CHA2(K5_1.braid())
sage: mt = br.formal_markov_trace()
sage: MT = mt.base_ring()
sage: f = MT.specialize_links_gould(); f
Composite map:
  From: Multivariate Polynomial Ring in u, v, w, s over Integer Ring
        localized at (s, w, v, u)
  To:   Multivariate Laurent Polynomial Ring in t0, t1 over Integer Ring
  Defn: Ring morphism:
        From: Multivariate Polynomial Ring in u, v, w, s
              over Integer Ring localized at (s, w, v, u)
        To:   Multivariate Polynomial Ring in t0, t1 over Integer Ring
              localized at (t1, t0, t0 + t1 - 1, t0*t1 - t0 - t1)
  Defn: u |--> t0 + t1 - 1
        v |--> t0*t1 - t0 - t1
        w |--> -t0*t1
        s |--> 1

then
  Conversion map:
  From: Multivariate Polynomial Ring in t0, t1 over Integer Ring
        localized at (t1, t0, t0 + t1 - 1, t0*t1 - t0 - t1)
  To:   Multivariate Laurent Polynomial Ring in t0, t1 over Integer Ring
↪Ring
sage: sup = mt.support()
sage: sum(f(mt.coefficient(b)) * b.links_gould_polynomial() for b in sup)
-t0^4*t1 - t0^3*t1^2 - t0^2*t1^3 - t0*t1^4 + t0^4 + 2*t0^3*t1 + 2*t0^2*t1^2
+ 2*t0*t1^3 + t1^4 - t0^3 - 2*t0^2*t1 - 2*t0*t1^2 - t1^3 + t0^2 + 2*t0*t1
+ t1^2 - t0 - t1 + 1

```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```

>>> from sage.knots.knotinfo import KnotInfo
>>> CHA2 = algebras.CubicHecke(Integer(2))
>>> K5_1 = KnotInfo.K5_1.link()
>>> br = CHA2(K5_1.braid())
>>> mt = br.formal_markov_trace()
>>> MT = mt.base_ring()
>>> f = MT.specialize_links_gould(); f
Composite map:
  From: Multivariate Polynomial Ring in u, v, w, s over Integer Ring
        localized at (s, w, v, u)
  To:   Multivariate Laurent Polynomial Ring in t0, t1 over Integer Ring
  Defn: Ring morphism:
        From: Multivariate Polynomial Ring in u, v, w, s
              over Integer Ring localized at (s, w, v, u)
        To:   Multivariate Polynomial Ring in t0, t1 over Integer Ring
              localized at (t1, t0, t0 + t1 - 1, t0*t1 - t0 - t1)
  Defn: u |--> t0 + t1 - 1
        v |--> t0*t1 - t0 - t1
        w |--> -t0*t1
        s |--> 1
  then
    Conversion map:
    From: Multivariate Polynomial Ring in t0, t1 over Integer Ring
          localized at (t1, t0, t0 + t1 - 1, t0*t1 - t0 - t1)
    To:   Multivariate Laurent Polynomial Ring in t0, t1 over Integer
    ↪Ring
>>> sup = mt.support()
>>> sum(f(mt.coefficient(b)) * b.links_gould_polynomial() for b in sup)
-t0^4*t1 - t0^3*t1^2 - t0^2*t1^3 - t0*t1^4 + t0^4 + 2*t0^3*t1 + 2*t0^2*t1^2
+ 2*t0*t1^3 + t1^4 - t0^3 - 2*t0^2*t1 - 2*t0*t1^2 - t1^3 + t0^2 + 2*t0*t1
+ t1^2 - t0 - t1 + 1

```

class sage.algebras.hecke_algebras.cubic_hecke_base_ring.GaloisGroupAction

Bases: Action

Action on a multivariate polynomial ring by permuting the generators.

EXAMPLES:

```

sage: from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
sage: from operator import mul
sage: R.<x, y, z> = ZZ[]
sage: G = SymmetricGroup(3)
sage: p = 5*x*y + 3*z**2
sage: R._unset_coercions_used()
sage: R.register_action(chbr.GaloisGroupAction(G, R, op=mul))
sage: s = G([2,3,1])
sage: s*p
3*x^2 + 5*y*z

```

```

>>> from sage.all import *
>>> from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
>>> from operator import mul

```

(continues on next page)

(continued from previous page)

```

>>> R = ZZ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> G = SymmetricGroup(Integer(3))
>>> p = Integer(5)*x*y + Integer(3)*z**Integer(2)
>>> R._unset_coercions_used()
>>> R.register_action(chbr.GaloisGroupAction(G, R, op=mul))
>>> s = G([Integer(2), Integer(3), Integer(1)])
>>> s*p
3*x^2 + 5*y*z

```

sage.algebras.hecke_algebras.cubic_hecke_base_ring.**normalize_names_markov**(names,
markov_trace_version)

Return a tuple of strings of variable names of length 3 resp. 4 (if markov_trace_version is True) according to the given input names.

INPUT:

- names – passed to `normalize_names()`
- markov_trace_version – boolean; if set to True four names are expected the last of which corresponds to the writhe factor of the Markov trace

EXAMPLES:

```

sage: from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
sage: chbr.normalize_names_markov('a, b, c', False)
('a', 'b', 'c')
sage: chbr.normalize_names_markov(('u', 'v', 'w', 's'), False)
('u', 'v', 'w')

```

```

>>> from sage.all import *
>>> from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
>>> chbr.normalize_names_markov('a, b, c', False)
('a', 'b', 'c')
>>> chbr.normalize_names_markov(('u', 'v', 'w', 's'), False)
('u', 'v', 'w')

```

sage.algebras.hecke_algebras.cubic_hecke_base_ring.**register_ring_hom**(ring_hom)

Register the given ring homomorphism as conversion map.

EXAMPLES:

```

sage: from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
sage: BR = chbr.CubicHeckeRingOfDefinition()
sage: BR.create_specialization([E(5), E(7), E(3)]) # indirect doctest
Universal Cyclotomic Field
sage: _.convert_map_from(BR)
Ring morphism:
  From: Multivariate Polynomial Ring in u, v, w
        over Integer Ring localized at (w,)
  To:   Universal Cyclotomic Field
  Defn: u |--> E(5)
        v |--> E(7)
        w |--> E(3)

```



```

>>> from sage.all import *
>>> from sage.algebras.hecke_algebras import cubic_hecke_base_ring as chbr
>>> BR = chbr.CubicHeckeRingOfDefinition()
>>> BR.create_specialization([E(Integer(5)), E(Integer(7)), E(Integer(3))]) #_
↳ indirect doctest
Universal Cyclotomic Field
>>> _.convert_map_from(BR)
Ring morphism:
  From: Multivariate Polynomial Ring in u, v, w
        over Integer Ring localized at (w,)
  To:   Universal Cyclotomic Field
  Defn: u |--> E(5)
        v |--> E(7)
        w |--> E(3)

```

6.6.3 Cubic Hecke matrix representations

This module contains the class `CubicHeckeMatrixRep` which is used to treat the matrix representations of the elements of the cubic Hecke algebra (`CubicHeckeAlgebra`) together with its parent class `CubicHeckeMatrixSpace`. Furthermore, it contains enums for their types (`RepresentationType`) and names (`AbsIrreducibeRep`).

AUTHORS:

- Sebastian Oehms May 2020: initial version

```

class sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibeRep(value,
                                                                              names=<not
                                                                              given>, *values,
                                                                              module=None,
                                                                              qual-
                                                                              name=None,
                                                                              type=None,
                                                                              start=1, bound-
                                                                              ary=None)

```

Bases: `Enum`

Enum class to select an absolutely irreducible representation for the cubic Hecke algebra (`CHAn`) on n -strands.

The names are build as follows: Take the determinant of one of the generators of the `CHAn`. This is a monomial in the generic extension ring (`GER`) of `CHA`, say $a^i b^j c^k$ where a , b and c are the generators of `GER`. This does not depend on the choice of the generator of `CHA`, since these are conjugated to each other. This monomial might be looked as the weight of the representation. Therefore we use it as a name:

w_{n-ijk}

The only ambiguity among the available irreducible representations occurs for the two nine-dimensional modules, which are conjugated to each other and distinguished by these names:

w_{4-333} and $w_{4-333bar}$

Examples of names:

- w_{2-100} – one dimensional representation of the cubic Hecke algebra on 2 strands corresponding to the first root of the cubic equation
- w_{3-111} – three dimensional irreducible representation of the cubic Hecke algebra on 3 strands
- w_{4-242} – eight dimensional irreducible representation of the cubic Hecke algebra on 4 strands having the second root of the cubic equation as weight of dimension 4

Alternative names are taken from [MW2012] and can be shown by `alternative_name()`.

EXAMPLES:

```
sage: import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
sage: [irr.name for irr in chmr.AbsIrreducibeRep]
['W2_100', 'W2_001', 'W2_010', 'W3_100', 'W3_001', 'W3_010', 'W3_011', 'W3_110',
 'W3_101', 'W3_111', 'W4_100', 'W4_001', 'W4_010', 'W4_011', 'W4_110', 'W4_101',
 'W4_111', 'W4_120', 'W4_201', 'W4_012', 'W4_102', 'W4_210', 'W4_021', 'W4_213',
 'W4_132', 'W4_321', 'W4_231', 'W4_123', 'W4_312', 'W4_422', 'W4_224', 'W4_242',
 'W4_333', 'W4_333bar', 'W5_100', 'W5_001', 'W5_010', 'W5_013', 'W5_130', 'W5_301
→',
 'W5_031', 'W5_103', 'W5_310', 'W5_203', 'W5_032', 'W5_320', 'W5_230', 'W5_023',
 'W5_302', 'W5_033', 'W5_330', 'W5_303', 'W5_163', 'W5_631', 'W5_316', 'W5_136',
 'W5_613', 'W5_361', 'W5_366', 'W5_663', 'W5_636', 'W5_933', 'W5_339', 'W5_393']
```

```
>>> from sage.all import *
>>> import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
>>> [irr.name for irr in chmr.AbsIrreducibeRep]
['W2_100', 'W2_001', 'W2_010', 'W3_100', 'W3_001', 'W3_010', 'W3_011', 'W3_110',
 'W3_101', 'W3_111', 'W4_100', 'W4_001', 'W4_010', 'W4_011', 'W4_110', 'W4_101',
 'W4_111', 'W4_120', 'W4_201', 'W4_012', 'W4_102', 'W4_210', 'W4_021', 'W4_213',
 'W4_132', 'W4_321', 'W4_231', 'W4_123', 'W4_312', 'W4_422', 'W4_224', 'W4_242',
 'W4_333', 'W4_333bar', 'W5_100', 'W5_001', 'W5_010', 'W5_013', 'W5_130', 'W5_301
→',
 'W5_031', 'W5_103', 'W5_310', 'W5_203', 'W5_032', 'W5_320', 'W5_230', 'W5_023',
 'W5_302', 'W5_033', 'W5_330', 'W5_303', 'W5_163', 'W5_631', 'W5_316', 'W5_136',
 'W5_613', 'W5_361', 'W5_366', 'W5_663', 'W5_636', 'W5_933', 'W5_339', 'W5_393']
```

REFERENCES:

- [MW2012]

```
W2_001 = {'alt_name': 'Sc', 'dim': 1, 'gap_ind': 1, 'intern_ind': 1, 'len_orbit':
3, 'ngens': 1}
```

```
W2_010 = {'alt_name': 'Sb', 'dim': 1, 'gap_ind': 2, 'intern_ind': 2, 'len_orbit':
3, 'ngens': 1}
```

```
W2_100 = {'alt_name': 'Sa', 'dim': 1, 'gap_ind': 0, 'intern_ind': 0, 'len_orbit':
3, 'ngens': 1}
```

```
W3_001 = {'alt_name': 'Sc', 'dim': 1, 'gap_ind': 1, 'intern_ind': 1, 'len_orbit':
3, 'ngens': 2}
```

```
W3_010 = {'alt_name': 'Sb', 'dim': 1, 'gap_ind': 2, 'intern_ind': 2, 'len_orbit':
3, 'ngens': 2}
```

```
W3_011 = {'alt_name': 'Tbc', 'dim': 2, 'gap_ind': 3, 'intern_ind': 3,
'len_orbit': 3, 'ngens': 2}
```

```
W3_100 = {'alt_name': 'Sa', 'dim': 1, 'gap_ind': 0, 'intern_ind': 0, 'len_orbit':
3, 'ngens': 2}
```

```
W3_101 = {'alt_name': 'Tac', 'dim': 2, 'gap_ind': 5, 'intern_ind': 5,
'len_orbit': 3, 'ngens': 2}
```

```

W3_110 = {'alt_name': 'Tab', 'dim': 2, 'gap_ind': 4, 'intern_ind': 4,
'len_orbit': 3, 'ngens': 2}

W3_111 = {'alt_name': 'V', 'dim': 3, 'gap_ind': 6, 'intern_ind': 6, 'len_orbit':
1, 'ngens': 2}

W4_001 = {'alt_name': 'Sc', 'dim': 1, 'gap_ind': 1, 'intern_ind': 1, 'len_orbit':
3, 'ngens': 3}

W4_010 = {'alt_name': 'Sb', 'dim': 1, 'gap_ind': 2, 'intern_ind': 2, 'len_orbit':
3, 'ngens': 3}

W4_011 = {'alt_name': 'Tbc', 'dim': 2, 'gap_ind': 3, 'intern_ind': 3,
'len_orbit': 3, 'ngens': 3}

W4_012 = {'alt_name': 'Ucb', 'dim': 3, 'gap_ind': 9, 'intern_ind': 9,
'len_orbit': 6, 'ngens': 3}

W4_021 = {'alt_name': 'Ubc', 'dim': 3, 'gap_ind': 12, 'intern_ind': 12,
'len_orbit': 6, 'ngens': 3}

W4_100 = {'alt_name': 'Sa', 'dim': 1, 'gap_ind': 0, 'intern_ind': 0, 'len_orbit':
3, 'ngens': 3}

W4_101 = {'alt_name': 'Tac', 'dim': 2, 'gap_ind': 5, 'intern_ind': 5,
'len_orbit': 3, 'ngens': 3}

W4_102 = {'alt_name': 'Uca', 'dim': 3, 'gap_ind': 10, 'intern_ind': 10,
'len_orbit': 6, 'ngens': 3}

W4_110 = {'alt_name': 'Tab', 'dim': 2, 'gap_ind': 4, 'intern_ind': 4,
'len_orbit': 3, 'ngens': 3}

W4_111 = {'alt_name': 'V', 'dim': 3, 'gap_ind': 6, 'intern_ind': 6, 'len_orbit':
1, 'ngens': 3}

W4_120 = {'alt_name': 'Uba', 'dim': 3, 'gap_ind': 7, 'intern_ind': 7,
'len_orbit': 6, 'ngens': 3}

W4_123 = {'alt_name': 'Vcba', 'dim': 6, 'gap_ind': 17, 'intern_ind': 17,
'len_orbit': 6, 'ngens': 3}

W4_132 = {'alt_name': 'Vbca', 'dim': 6, 'gap_ind': 14, 'intern_ind': 14,
'len_orbit': 6, 'ngens': 3}

W4_201 = {'alt_name': 'Uac', 'dim': 3, 'gap_ind': 8, 'intern_ind': 8,
'len_orbit': 6, 'ngens': 3}

W4_210 = {'alt_name': 'Uab', 'dim': 3, 'gap_ind': 11, 'intern_ind': 11,
'len_orbit': 6, 'ngens': 3}

W4_213 = {'alt_name': 'Vcab', 'dim': 6, 'gap_ind': 13, 'intern_ind': 13,
'len_orbit': 6, 'ngens': 3}

W4_224 = {'alt_name': 'Wc', 'dim': 8, 'gap_ind': 20, 'intern_ind': 20,
'len_orbit': 3, 'ngens': 3}

```

```
W4_231 = {'alt_name': 'Vbac', 'dim': 6, 'gap_ind': 16, 'intern_ind': 16,
'len_orbit': 6, 'ngens': 3}

W4_242 = {'alt_name': 'Wb', 'dim': 8, 'gap_ind': 21, 'intern_ind': 21,
'len_orbit': 3, 'ngens': 3}

W4_312 = {'alt_name': 'Vacb', 'dim': 6, 'gap_ind': 18, 'intern_ind': 18,
'len_orbit': 6, 'ngens': 3}

W4_321 = {'alt_name': 'Vabc', 'dim': 6, 'gap_ind': 15, 'intern_ind': 15,
'len_orbit': 6, 'ngens': 3}

W4_333 = {'alt_name': 'X', 'dim': 9, 'gap_ind': 22, 'intern_ind': 22,
'len_orbit': 2, 'ngens': 3}

W4_333bar = {'alt_name': 'Xbar', 'dim': 9, 'gap_ind': 23, 'intern_ind': 23,
'len_orbit': 2, 'ngens': 3}

W4_422 = {'alt_name': 'Wa', 'dim': 8, 'gap_ind': 19, 'intern_ind': 19,
'len_orbit': 3, 'ngens': 3}

W5_001 = {'alt_name': None, 'dim': 1, 'gap_ind': 1, 'intern_ind': 1, 'len_orbit':
3, 'ngens': 4}

W5_010 = {'alt_name': None, 'dim': 1, 'gap_ind': 2, 'intern_ind': 2, 'len_orbit':
3, 'ngens': 4}

W5_013 = {'alt_name': None, 'dim': 4, 'gap_ind': 3, 'intern_ind': 3, 'len_orbit':
6, 'ngens': 4}

W5_023 = {'alt_name': None, 'dim': 5, 'gap_ind': 13, 'intern_ind': 13,
'len_orbit': 6, 'ngens': 4}

W5_031 = {'alt_name': None, 'dim': 4, 'gap_ind': 6, 'intern_ind': 6, 'len_orbit':
6, 'ngens': 4}

W5_032 = {'alt_name': None, 'dim': 5, 'gap_ind': 10, 'intern_ind': 10,
'len_orbit': 6, 'ngens': 4}

W5_033 = {'alt_name': None, 'dim': 6, 'gap_ind': 15, 'intern_ind': 15,
'len_orbit': 3, 'ngens': 4}

W5_100 = {'alt_name': None, 'dim': 1, 'gap_ind': 0, 'intern_ind': 0, 'len_orbit':
3, 'ngens': 4}

W5_103 = {'alt_name': None, 'dim': 4, 'gap_ind': 7, 'intern_ind': 7, 'len_orbit':
6, 'ngens': 4}

W5_130 = {'alt_name': None, 'dim': 4, 'gap_ind': 4, 'intern_ind': 4, 'len_orbit':
6, 'ngens': 4}

W5_136 = {'alt_name': None, 'dim': 10, 'gap_ind': 21, 'intern_ind': 21,
'len_orbit': 6, 'ngens': 4}

W5_163 = {'alt_name': None, 'dim': 10, 'gap_ind': 18, 'intern_ind': 18,
'len_orbit': 6, 'ngens': 4}
```

```

W5_203 = {'alt_name': None, 'dim': 5, 'gap_ind': 9, 'intern_ind': 9, 'len_orbit':
6, 'ngens': 4}

W5_230 = {'alt_name': None, 'dim': 5, 'gap_ind': 12, 'intern_ind': 12,
'len_orbit': 6, 'ngens': 4}

W5_301 = {'alt_name': None, 'dim': 4, 'gap_ind': 5, 'intern_ind': 5, 'len_orbit':
6, 'ngens': 4}

W5_302 = {'alt_name': None, 'dim': 5, 'gap_ind': 14, 'intern_ind': 14,
'len_orbit': 6, 'ngens': 4}

W5_303 = {'alt_name': None, 'dim': 6, 'gap_ind': 17, 'intern_ind': 17,
'len_orbit': 3, 'ngens': 4}

W5_310 = {'alt_name': None, 'dim': 4, 'gap_ind': 8, 'intern_ind': 8, 'len_orbit':
6, 'ngens': 4}

W5_316 = {'alt_name': None, 'dim': 10, 'gap_ind': 20, 'intern_ind': 20,
'len_orbit': 6, 'ngens': 4}

W5_320 = {'alt_name': None, 'dim': 5, 'gap_ind': 11, 'intern_ind': 11,
'len_orbit': 6, 'ngens': 4}

W5_330 = {'alt_name': None, 'dim': 6, 'gap_ind': 16, 'intern_ind': 16,
'len_orbit': 3, 'ngens': 4}

W5_339 = {'alt_name': None, 'dim': 15, 'gap_ind': 28, 'intern_ind': 28,
'len_orbit': 3, 'ngens': 4}

W5_361 = {'alt_name': None, 'dim': 10, 'gap_ind': 23, 'intern_ind': 23,
'len_orbit': 6, 'ngens': 4}

W5_366 = {'alt_name': None, 'dim': 15, 'gap_ind': 24, 'intern_ind': 24,
'len_orbit': 3, 'ngens': 4}

W5_393 = {'alt_name': None, 'dim': 15, 'gap_ind': 29, 'intern_ind': 29,
'len_orbit': 3, 'ngens': 4}

W5_613 = {'alt_name': None, 'dim': 10, 'gap_ind': 22, 'intern_ind': 22,
'len_orbit': 6, 'ngens': 4}

W5_631 = {'alt_name': None, 'dim': 10, 'gap_ind': 19, 'intern_ind': 19,
'len_orbit': 6, 'ngens': 4}

W5_636 = {'alt_name': None, 'dim': 15, 'gap_ind': 27, 'intern_ind': 26,
'len_orbit': 3, 'ngens': 4}

W5_663 = {'alt_name': None, 'dim': 15, 'gap_ind': 26, 'intern_ind': 25,
'len_orbit': 3, 'ngens': 4}

W5_933 = {'alt_name': None, 'dim': 15, 'gap_ind': 25, 'intern_ind': 27,
'len_orbit': 3, 'ngens': 4}

```

alternative_name()

Return the name of the split irreducible representation for cubic Hecke algebras for up to four strands as given in [MW2012].

EXAMPLES:

```
sage: import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
sage: chmr.AbsIrreducibeRep.W3_011.alternative_name()
'Tbc'
```

```
>>> from sage.all import *
>>> import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
>>> chmr.AbsIrreducibeRep.W3_011.alternative_name()
'Tbc'
```

dimension()

Return the dimension of the representation.

EXAMPLES:

```
sage: import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
sage: chmr.AbsIrreducibeRep.W3_111.dimension()
3
```

```
>>> from sage.all import *
>>> import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
>>> chmr.AbsIrreducibeRep.W3_111.dimension()
3
```

gap_index()

Return the array index of this representation for the access to the GAP3 package CHEVIE.

EXAMPLES:

```
sage: import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
sage: chmr.AbsIrreducibeRep.W3_111.gap_index()
6
```

```
>>> from sage.all import *
>>> import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
>>> chmr.AbsIrreducibeRep.W3_111.gap_index()
6
```

internal_index()

Return the array index of this representation for the internal access.

EXAMPLES:

```
sage: import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
sage: chmr.AbsIrreducibeRep.W3_111.internal_index()
6
```

```
>>> from sage.all import *
>>> import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
```

(continues on next page)

(continued from previous page)

```
>>> chmr.AbsIrreducibleRep.W3_111.internal_index()
6
```

length_orbit()

Return the length of the orbit of this representation under the action of the Galois group of the cubic equation.

EXAMPLES:

```
sage: import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
sage: chmr.AbsIrreducibleRep.W3_001.length_orbit()
3
sage: chmr.AbsIrreducibleRep.W3_111.length_orbit()
1
```

```
>>> from sage.all import *
>>> import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
>>> chmr.AbsIrreducibleRep.W3_001.length_orbit()
3
>>> chmr.AbsIrreducibleRep.W3_111.length_orbit()
1
```

number_gens()

Return the number of generators of the underlying cubic Hecke algebra.

EXAMPLES:

```
sage: import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
sage: chmr.AbsIrreducibleRep.W3_001.number_gens()
2
sage: chmr.AbsIrreducibleRep.W4_001.number_gens()
3
```

```
>>> from sage.all import *
>>> import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
>>> chmr.AbsIrreducibleRep.W3_001.number_gens()
2
>>> chmr.AbsIrreducibleRep.W4_001.number_gens()
3
```

class sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.CubicHeckeMatrixRep

Bases: Matrix_generic_dense

Class to supervise the diagonal block matrix structure arising from cubic Hecke algebra-representations.

EXAMPLES:

```
sage: import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
sage: CHA2.<c1> = algebras.CubicHecke(2)
sage: MS = chmr.CubicHeckeMatrixSpace(CHA2)
sage: m1 = MS(c1); m1
[      a      0      0]
[      0      b      0]
[      0      0 -b - a + u]
```

(continues on next page)

(continued from previous page)

```

sage: type(m1)
<class 'sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.CubicHeckeMatrixSpace_
↳with_category.element_class'>
sage: m1.block_diagonal_list()
[[a], [b], [-b - a + u]]

sage: MSo = chmr.CubicHeckeMatrixSpace(CHA2, original=True)
sage: MSo(c1)
[a 0 0]
[0 b 0]
[0 0 c]

sage: reg_left = chmr.RepresentationType.RegularLeft
sage: MSreg = chmr.CubicHeckeMatrixSpace(CHA2, representation_type=reg_left)
sage: MSreg(c1)
[ 0 -v  1]
[ 1  u  0]
[ 0  w  0]
sage: len(_.block_diagonal_list())
1

```

```

>>> from sage.all import *
>>> import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
>>> CHA2 = algebras.CubicHecke(Integer(2), names=('c1',)); (c1,) = CHA2._first_
↳ngens(1)
>>> MS = chmr.CubicHeckeMatrixSpace(CHA2)
>>> m1 = MS(c1); m1
[      a      0      0]
[      0      b      0]
[      0      0 -b - a + u]
>>> type(m1)
<class 'sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.CubicHeckeMatrixSpace_
↳with_category.element_class'>
>>> m1.block_diagonal_list()
[[a], [b], [-b - a + u]]

>>> MSo = chmr.CubicHeckeMatrixSpace(CHA2, original=True)
>>> MSo(c1)
[a 0 0]
[0 b 0]
[0 0 c]

>>> reg_left = chmr.RepresentationType.RegularLeft
>>> MSreg = chmr.CubicHeckeMatrixSpace(CHA2, representation_type=reg_left)
>>> MSreg(c1)
[ 0 -v  1]
[ 1  u  0]
[ 0  w  0]
>>> len(_.block_diagonal_list())
1

```

block_diagonal_list()

Return the list of sub-matrix blocks of `self` considered as block diagonal matrix.

OUTPUT:

A list of instances of `Matrix_generic_dense` each of which represents a diagonal block of `self`.

EXAMPLES:

```
sage: CHA2.<c1> = algebras.CubicHecke(2)
sage: c1.matrix().block_diagonal_list()
[[a], [b], [-b - a + u]]
```

```
>>> from sage.all import *
>>> CHA2 = algebras.CubicHecke(Integer(2), names=('c1',)); (c1,) = CHA2._
↳first_ngens(1)
>>> c1.matrix().block_diagonal_list()
[[a], [b], [-b - a + u]]
```

`reduce_to_irr_block(irr)`

Return a copy of `self` with zeroes outside the block corresponding to `irr` but the block according to the input identical to that of `self`.

INPUT:

- `irr` – an *AbsIrreducibleRep* specifying an absolute irreducible representation of the cubic Hecke algebra; alternatively, it can be specified by list index (see `internal_index()` respectively `gap_index()`)

OUTPUT:

An instance of `Matrix_generic_dense` with exactly one nonzero block according to `irr`.

EXAMPLES:

```
sage: CHA2.<c1> = algebras.CubicHecke(2)
sage: m1 = c1.matrix()
sage: m1.reduce_to_irr_block(0)
[a 0 0]
[0 0 0]
[0 0 0]
sage: m1.reduce_to_irr_block(CHA2.irred_repr.W2_001)
[0 0 0]
[0 b 0]
[0 0 0]
```

```
>>> from sage.all import *
>>> CHA2 = algebras.CubicHecke(Integer(2), names=('c1',)); (c1,) = CHA2._
↳first_ngens(1)
>>> m1 = c1.matrix()
>>> m1.reduce_to_irr_block(Integer(0))
[a 0 0]
[0 0 0]
[0 0 0]
>>> m1.reduce_to_irr_block(CHA2.irred_repr.W2_001)
[0 0 0]
[0 b 0]
[0 0 0]
```

```
class sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.CubicHeckeMatrixSpace (base_ring,
                                                                                     dimen-
                                                                                     sion,
                                                                                     cu-
                                                                                     bic_hecke_al-
                                                                                     gebra,
                                                                                     represen-
                                                                                     ta-
                                                                                     tion_type,
                                                                                     subdi-
                                                                                     vide)
```

Bases: `MatrixSpace`

The matrix space of cubic Hecke algebra representations.

INPUT:

- `cubic_hecke_algebra` – (optional) `CubicHeckeAlgebra` must be given if `element` fails to be an instance of its element class
- `representation_type` – (default: `RepresentationType.SplitIrredChevie`) `RepresentationType` specifying the type of the representation
- `subdivide` – boolean (default: `False`); whether or not to subdivide the resulting matrices
- `original` – boolean (default: `False`); if `True`, the matrix will have coefficients in the generic base / extension ring

EXAMPLES:

```
sage: CHA2.<c1> = algebras.CubicHecke(2)
sage: c1.matrix()          # indirect doctest
[      a      0      0]
[      0      b      0]
[      0      0 -b - a + u]
sage: c1.matrix(original=True)
[a 0 0]
[0 b 0]
[0 0 c]
sage: c1.matrix(representation_type = CHA2.repr_type.RegularLeft) # indirect_
↪doctest
[ 0 -v  1]
[ 1  u  0]
[ 0  w  0]
```

```
>>> from sage.all import *
>>> CHA2 = algebras.CubicHecke(Integer(2), names=('c1',)); (c1,) = CHA2._first_
↪ngens(1)
>>> c1.matrix()          # indirect doctest
[      a      0      0]
[      0      b      0]
[      0      0 -b - a + u]
>>> c1.matrix(original=True)
[a 0 0]
[0 b 0]
[0 0 c]
```

(continues on next page)

(continued from previous page)

```
>>> c1.matrix(representation_type = CHA2.repr_type.RegularLeft) # indirect_
↪doctest
[ 0 -v  1]
[ 1  u  0]
[ 0  w  0]
```

construction()

Return None since this construction is not functorial.

EXAMPLES:

```
sage: CHA2.<c1> = algebras.CubicHecke(2)
sage: MS = c1.matrix().parent()
sage: MS._test_category() # indirect doctest
```

```
>>> from sage.all import *
>>> CHA2 = algebras.CubicHecke(Integer(2), names=('c1',)); (c1,) = CHA2._
↪first_ngens(1)
>>> MS = c1.matrix().parent()
>>> MS._test_category() # indirect doctest
```

one()

Return the one element of self.

EXAMPLES:

```
sage: CHA2.<c1> = algebras.CubicHecke(2)
sage: m1 = c1.matrix()
sage: m1rl = c1.matrix(representation_type = CHA2.repr_type.RegularLeft)
sage: o = m1.parent().one()
sage: orl = m1rl.parent().one()
sage: matrix(o) == matrix(orl), o.is_one(), orl.is_one()
(True, True, True)
sage: o.block_diagonal_list()
[[1], [1], [1]]
sage: orl.block_diagonal_list()
[
[1 0 0]
[0 1 0]
[0 0 1]
]
```

```
>>> from sage.all import *
>>> CHA2 = algebras.CubicHecke(Integer(2), names=('c1',)); (c1,) = CHA2._
↪first_ngens(1)
>>> m1 = c1.matrix()
>>> m1rl = c1.matrix(representation_type = CHA2.repr_type.RegularLeft)
>>> o = m1.parent().one()
>>> orl = m1rl.parent().one()
>>> matrix(o) == matrix(orl), o.is_one(), orl.is_one()
(True, True, True)
>>> o.block_diagonal_list()
```

(continues on next page)

(continued from previous page)

```

[[1], [1], [1]]
>>> orl.block_diagonal_list()
[
[1 0 0]
[0 1 0]
[0 0 1]
]

```

some_elements()

Return a generator of elements of `self`.

EXAMPLES:

```

sage: CHA2.<c1> = algebras.CubicHecke(2, cubic_equation_roots=(2, 3, 5))
sage: M = c1.matrix(); M
[2 0 0]
[0 3 0]
[0 0 5]
sage: MS = M.parent()
sage: MS.some_elements()
(
[ 94/3      0      0]
[    0 187/3      0]
[    0      0 373/3]
)
sage: MS.some_elements() == tuple(MS(x) for x in CHA2.some_elements())
True

```

```

>>> from sage.all import *
>>> CHA2 = algebras.CubicHecke(Integer(2), cubic_equation_roots=(Integer(2), Integer(3), Integer(5)), names=('c1',)); (c1,) = CHA2._first_ngens(1)
>>> M = c1.matrix(); M
[2 0 0]
[0 3 0]
[0 0 5]
>>> MS = M.parent()
>>> MS.some_elements()
(
[ 94/3      0      0]
[    0 187/3      0]
[    0      0 373/3]
)
>>> MS.some_elements() == tuple(MS(x) for x in CHA2.some_elements())
True

```

zero()

Return the zero element of `self`.

EXAMPLES:

```

sage: CHA2.<c1> = algebras.CubicHecke(2)
sage: m1 = c1.matrix()
sage: m1rl = c1.matrix(representation_type = CHA2.repr_type.RegularLeft)

```

(continues on next page)

(continued from previous page)

```

sage: z = m1.parent().zero()
sage: zrl = m1rl.parent().zero()
sage: matrix(z) == matrix(zrl), z.is_zero(), zrl.is_zero()
(True, True, True)
sage: z.block_diagonal_list()
[[0], [0], [0]]
sage: zrl.block_diagonal_list()
[
[0 0 0]
[0 0 0]
[0 0 0]
]

```

```

>>> from sage.all import *
>>> CHA2 = algebras.CubicHecke(Integer(2), names=('c1',)); (c1,) = CHA2._
↳first_ngens(1)
>>> m1 = c1.matrix()
>>> m1rl = c1.matrix(representation_type = CHA2.repr_type.RegularLeft)
>>> z = m1.parent().zero()
>>> zrl = m1rl.parent().zero()
>>> matrix(z) == matrix(zrl), z.is_zero(), zrl.is_zero()
(True, True, True)
>>> z.block_diagonal_list()
[[0], [0], [0]]
>>> zrl.block_diagonal_list()
[
[0 0 0]
[0 0 0]
[0 0 0]
]

```

```

class sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.GenSign(value, names=<not given>,
                                                                    *values, module=None,
                                                                    qualname=None,
                                                                    type=None, start=1,
                                                                    boundary=None)

```

Bases: Enum

Enum class to select the braid generators sign.

EXAMPLES:

```

sage: import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
sage: chmr.GenSign.pos
<GenSign.pos: 1>
sage: chmr.GenSign.neg
<GenSign.neg: -1>

```

```

>>> from sage.all import *
>>> import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
>>> chmr.GenSign.pos
<GenSign.pos: 1>

```

(continues on next page)

(continued from previous page)

```
>>> chmr.GenSign.neg
<GenSign.neg: -1>
```

```
neg = -1
```

```
pos = 1
```

```
class sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.RepresentationType(value,
                                                                              names=<not
                                                                              given>,
                                                                              *values,
                                                                              mod-
                                                                              ule=None,
                                                                              qual-
                                                                              name=None,
                                                                              type=None,
                                                                              start=1,
                                                                              bound-
                                                                              ary=None)
```

Bases: `Enum`

Enum class to select a representation type for the cubic Hecke algebra.

- `RegularLeft` – left regular representations
- `RegularRight` – right regular representations
- `SplitIrredMarin` – split irreducible representations obtained from Ivan Marin’s data
- `SplitIrredChevie` – the split irreducible representations obtained from CHEVIE via the GAP3 interface

EXAMPLES:

```
sage: import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
sage: chmr.RepresentationType.RegularLeft.is_regular()
True
```

```
>>> from sage.all import *
>>> import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
>>> chmr.RepresentationType.RegularLeft.is_regular()
True
```

```
RegularLeft = {'data': CubicHeckeDataSection.regular_left, 'num_rep': [1, 1, 1,
1], 'regular': True, 'split': False}
```

```
RegularRight = {'data': CubicHeckeDataSection.regular_right, 'num_rep': [1, 1, 1,
1], 'regular': True, 'split': False}
```

```
SplitIrredChevie = {'data': None, 'num_rep': [1, 3, 7, 24, 30], 'regular': False,
'split': True}
```

```
SplitIrredMarin = {'data': CubicHeckeDataSection.split_irred, 'num_rep': [1, 3, 7,
24], 'regular': False, 'split': True}
```

data_section()

Return the name of the data file. For more information see [CubicHeckeDataBase](#).

EXAMPLES:

```
sage: import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
sage: reg_left = chmr.RepresentationType.RegularLeft
sage: reg_left.data_section()
<CubicHeckeDataSection.regular_left: 'regular_left'>
```

```
>>> from sage.all import *
>>> import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
>>> reg_left = chmr.RepresentationType.RegularLeft
>>> reg_left.data_section()
<CubicHeckeDataSection.regular_left: 'regular_left'>
```

is_regular()

Return True if this representation type is regular, False else-wise.

EXAMPLES:

```
sage: import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
sage: reg_left = chmr.RepresentationType.RegularLeft
sage: reg_left.is_regular()
True
```

```
>>> from sage.all import *
>>> import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
>>> reg_left = chmr.RepresentationType.RegularLeft
>>> reg_left.is_regular()
True
```

is_split()

Return True if this representation type is absolutely split, False else-wise.

EXAMPLES:

```
sage: import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
sage: chevie = chmr.RepresentationType.SplitIrredChevie
sage: chevie.is_split()
True
```

```
>>> from sage.all import *
>>> import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
>>> chevie = chmr.RepresentationType.SplitIrredChevie
>>> chevie.is_split()
True
```

number_of_representations(nstrands)

Return the number of representations existing to that type.

EXAMPLES:

```
sage: import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
sage: chmr.RepresentationType.SplitIrredChevie.number_of_representations(4)
24
sage: chmr.RepresentationType.SplitIrredMarin.number_of_representations(4)
24
```

```
>>> from sage.all import *
>>> import sage.algebras.hecke_algebras.cubic_hecke_matrix_rep as chmr
>>> chmr.RepresentationType.SplitIrredChevie.number_of_
↳representations(Integer(4))
24
>>> chmr.RepresentationType.SplitIrredMarin.number_of_
↳representations(Integer(4))
24
```


GRADED ALGEBRAS

7.1 Finite dimensional graded commutative algebras

AUTHORS:

- Michael Jung (2021): initial version

```
class sage.algebras.finite_gca.FiniteGCAAlgebra (base, names, degrees, max_degree, category=None,
**kwargs)
```

Bases: `CombinatorialFreeModule`

Finite dimensional graded commutative algebras.

A finite dimensional graded commutative algebra A is an integer-graded algebra satisfying the super-algebra relation w.r.t. the degree modulo 2. More precisely, A has a graded ring structure

$$A = \bigoplus_{i=0}^n A_i,$$

where $n \in \mathbf{N}$ is the finite maximal degree, and the multiplication satisfies

$$A_i \cdot A_j \subset \begin{cases} A_{i+j} & \text{if } i+j \leq n, \\ 0 & \text{if } i+j > n, \end{cases}$$

as well as the super-algebra relation

$$xy = (-1)^{ij}yx$$

for all homogeneous elements $x \in A_i$ and $y \in A_j$.

Such an algebra is multiplicatively generated by a set of single monomials $\{x_1, \dots, x_k\}$, where each x_i is given a certain degree $\deg(x_i)$. To that end, this algebra can be given a vector space basis, and the basis vectors are of the form $x_1^{w_1} \cdots x_n^{w_n}$, where $\sum_{i=1}^k \deg(x_i) w_i \leq n$ and

$$w_i \in \begin{cases} \mathbf{Z}_2 & \text{if } \deg(x_i) \text{ is odd,} \\ \mathbf{N} & \text{if } \deg(x_i) \text{ is even.} \end{cases}$$

Typical examples of finite dimensional graded commutative algebras are cohomology rings over finite dimensional CW-complexes.

INPUT:

- `base` – the base field
- `names` – (optional) names of the generators: a list of strings or a single string with the names separated by commas. If not specified, the generators are named “x0”, “x1”,...

- `degrees` – (optional) a tuple or list specifying the degrees of the generators; if omitted, each generator is given degree 1, and if both `names` and `degrees` are omitted, an error is raised
- `max_degree` – the maximal degree of the graded algebra
- `mul_symbol` – (optional) symbol used for multiplication; if omitted, the string “*” is used
- `mul_latex_symbol` – (optional) latex symbol used for multiplication; if omitted, the empty string is used

EXAMPLES:

```
sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(1,2,2,3), max_degree=6)
sage: A
Graded commutative algebra with generators ('x', 'y', 'z', 't')
in degrees (1, 2, 2, 3) with maximal degree 6
sage: t*x + x*t
0
sage: x^2
0
sage: x*t^2
0
sage: x*y^2 + z*t
x*y^2 + z*t
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1),Integer(2),Integer(2),
↳Integer(3)), max_degree=Integer(6), names=('x', 'y', 'z', 't')); (x, y, z, t)↳
↳= A._first_ngens(4)
>>> A
Graded commutative algebra with generators ('x', 'y', 'z', 't')
in degrees (1, 2, 2, 3) with maximal degree 6
>>> t*x + x*t
0
>>> x**Integer(2)
0
>>> x*t**Integer(2)
0
>>> x*y**Integer(2) + z*t
x*y^2 + z*t
```

The generators can be returned with `algebra_generators()`:

```
sage: F = A.algebra_generators(); F
Family (x, y, z, t)
sage: [g.degree() for g in F]
[1, 2, 2, 3]
```

```
>>> from sage.all import *
>>> F = A.algebra_generators(); F
Family (x, y, z, t)
>>> [g.degree() for g in F]
[1, 2, 2, 3]
```

We can also return the basis:

```
sage: list(A.basis())
[1, x, z, y, t, x*z, x*y, x*t, z^2, y*z, y^2, z*t, y*t, x*z^2, x*y*z, x*y^2]
```

```
>>> from sage.all import *
>>> list(A.basis())
[1, x, z, y, t, x*z, x*y, x*t, z^2, y*z, y^2, z*t, y*t, x*z^2, x*y*z, x*y^2]
```

Depending on the context, the multiplication can be given a different symbol:

```
sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(1,2,6,6), max_degree=10,
.....:                                     mul_symbol='~',
.....:                                     mul_latex_symbol=r'\smile')
sage: x*y^2 + x*t
x~y^2 + x~t
sage: latex(x*y^2 - z*x)
x\smile y^{2} - x\smile z
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1),Integer(2),Integer(6),
↪Integer(6)), max_degree=Integer(10),
...                                     mul_symbol='~',
...                                     mul_latex_symbol=r'\smile', names=('x',
↪'y', 'z', 't',)); (x, y, z, t,) = A._first_ngens(4)
>>> x*y**Integer(2) + x*t
x~y^2 + x~t
>>> latex(x*y**Integer(2) - z*x)
x\smile y^{2} - x\smile z
```

Note

Notice, when the argument `max_degree` in the global namespace is omitted, an instance of the class `sage.algebras.commutative_dga.GCAlgebra` is created instead:

```
sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(1,2,6,6))
sage: type(A)
<class 'sage.algebras.commutative_dga.GCAlgebra_with_category'>
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1),Integer(2),Integer(6),
↪Integer(6)), names=('x', 'y', 'z', 't',)); (x, y, z, t,) = A._first_ngens(4)
>>> type(A)
<class 'sage.algebras.commutative_dga.GCAlgebra_with_category'>
```

`algebra_generators()`

Return the generators of `self` as a `sage.sets.family.TrivialFamily`.

EXAMPLES:

```
sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(4,8,2), max_degree=10)
sage: A.algebra_generators()
Family (x, y, z)
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(4), Integer(8),
↳ Integer(2)), max_degree=Integer(10), names=('x', 'y', 'z')); (x, y, z) =
↳ A._first_ngens(3)
>>> A.algebra_generators()
Family (x, y, z)
```

degree_on_basis(*i*)

Return the degree of a homogeneous element with index *i*.

EXAMPLES:

```
sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=(2,4,6), max_degree=7)
sage: a.degree()
2
sage: (2*a*b).degree()
6
sage: (a+b).degree()
Traceback (most recent call last):
...
ValueError: element is not homogeneous
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(2), Integer(4),
↳ Integer(6)), max_degree=Integer(7), names=('a', 'b', 'c')); (a, b, c) = A.
↳ _first_ngens(3)
>>> a.degree()
2
>>> (Integer(2)*a*b).degree()
6
>>> (a+b).degree()
Traceback (most recent call last):
...
ValueError: element is not homogeneous
```

gen(*i*)

Return the *i*-th generator of self.

EXAMPLES:

```
sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(4,8,2), max_degree=10)
sage: A.gen(0)
x
sage: A.gen(1)
y
sage: A.gen(2)
z
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(4), Integer(8),
↳ Integer(2)), max_degree=Integer(10), names=('x', 'y', 'z')); (x, y, z) =
↳ A._first_ngens(3)
>>> A.gen(Integer(0))
```

(continues on next page)

(continued from previous page)

```

x
>>> A.gen(Integer(1))
y
>>> A.gen(Integer(2))
z

```

gens()

Return the generators of `self` as a tuple.

EXAMPLES:

```

sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(4,8,2), max_degree=10)
sage: A.gens()
(x, y, z)

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(4),Integer(8),
↳Integer(2)), max_degree=Integer(10), names=('x', 'y', 'z')); (x, y, z,) = A.
↳A._first_ngens(3)
>>> A.gens()
(x, y, z)

```

max_degree()

Return the maximal degree of `self`.

EXAMPLES:

```

sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(1,2,3), max_degree=8)
sage: A.maximal_degree()
8

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1),Integer(2),
↳Integer(3)), max_degree=Integer(8), names=('x', 'y', 'z')); (x, y, z,) = A.
↳_first_ngens(3)
>>> A.maximal_degree()
8

```

maximal_degree()

Return the maximal degree of `self`.

EXAMPLES:

```

sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(1,2,3), max_degree=8)
sage: A.maximal_degree()
8

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1),Integer(2),
↳Integer(3)), max_degree=Integer(8), names=('x', 'y', 'z')); (x, y, z,) = A.
↳_first_ngens(3)
>>> A.maximal_degree()
8

```

ngens()

Return the number of generators of self.

EXAMPLES:

```
sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(4,8,2), max_degree=10)
sage: A.ngens()
3
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(4),Integer(8),
↳Integer(2)), max_degree=Integer(10), names=('x', 'y', 'z')); (x, y, z,) =_
↳A._first_ngens(3)
>>> A.ngens()
3
```

one_basis()

Return the index of the one element of self.

EXAMPLES:

```
sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(4,8,2), max_degree=10)
sage: ind = A.one_basis(); ind
[0, 0, 0]
sage: A.monomial(ind)
1
sage: A.one() # indirect doctest
1
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(4),Integer(8),
↳Integer(2)), max_degree=Integer(10), names=('x', 'y', 'z')); (x, y, z,) =_
↳A._first_ngens(3)
>>> ind = A.one_basis(); ind
[0, 0, 0]
>>> A.monomial(ind)
1
>>> A.one() # indirect doctest
1
```

product_on_basis(w1, w2)

Return the product of two indices within the algebra.

EXAMPLES:

```
sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(4,8,2), max_degree=10)
sage: z*x
x*z
sage: x^3
0
sage: 5*z + 4*z*x
5*z + 4*x*z
```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(4), Integer(8),
↳ Integer(2)), max_degree=Integer(10), names=('x', 'y', 'z')); (x, y, z) = A.
↳ A._first_ngens(3)
>>> z*x
x*z
>>> x**Integer(3)
0
>>> Integer(5)*z + Integer(4)*z*x
5*z + 4*x*z

```

```

sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(1,2,3), max_degree=5)
sage: 2*x*y
2*x*y
sage: x^2
0
sage: x*z
x*z
sage: z*x
-x*z
sage: x*y*z
0

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1), Integer(2),
↳ Integer(3)), max_degree=Integer(5), names=('x', 'y', 'z')); (x, y, z) = A.
↳ A._first_ngens(3)
>>> Integer(2)*x*y
2*x*y
>>> x**Integer(2)
0
>>> x*z
x*z
>>> z*x
-x*z
>>> x*y*z
0

```

7.2 Commutative Differential Graded Algebras

An algebra is said to be *graded commutative* if it is endowed with a grading and its multiplication satisfies the Koszul sign convention: $yx = (-1)^{ij}xy$ if x and y are homogeneous of degrees i and j , respectively. Thus the multiplication is anticommutative for odd degree elements, commutative otherwise. *Commutative differential graded algebras* are graded commutative algebras endowed with a graded differential of degree 1. These algebras can be graded over the integers or they can be multi-graded (i.e., graded over a finite rank free abelian group $\mathbb{Z}^n$); if multi-graded, the total degree is used in the Koszul sign convention, and the differential must have total degree 1.

EXAMPLES:

All of these algebras may be constructed with the function `GradedCommutativeAlgebra()`. For most users, that will be the main function of interest. See its documentation for many more examples.

We start by constructing some graded commutative algebras. Generators have degree 1 by default:

```

sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ)
sage: x.degree()
1
sage: x^2
0
sage: y*x
-x*y
sage: B.<a,b> = GradedCommutativeAlgebra(QQ, degrees = (2,3))
sage: a.degree()
2
sage: b.degree()
3

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z) = A._first_
↳ngens(3)
>>> x.degree()
1
>>> x**Integer(2)
0
>>> y*x
-x*y
>>> B = GradedCommutativeAlgebra(QQ, degrees = (Integer(2),Integer(3)), names=('a', 'b
↳',)); (a, b,) = B._first_ngens(2)
>>> a.degree()
2
>>> b.degree()
3

```

Once we have defined a graded commutative algebra, it is easy to define a differential on it using the `GCAAlgebra.cdg_algebra()` method:

```

sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(1,1,2))
sage: B = A.cdg_algebra({x: x*y, y: -x*y})
sage: B
Commutative Differential Graded Algebra with generators ('x', 'y', 'z')
in degrees (1, 1, 2) over Rational Field with differential:
  x --> x*y
  y --> -x*y
  z --> 0
sage: B.cohomology(3)
Free module generated by {[x*z + y*z]} over Rational Field
sage: B.cohomology(4)
Free module generated by {[z^2]} over Rational Field

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1),Integer(1),Integer(2)),
↳names=('x', 'y', 'z')); (x, y, z) = A._first_ngens(3)
>>> B = A.cdg_algebra({x: x*y, y: -x*y})
>>> B
Commutative Differential Graded Algebra with generators ('x', 'y', 'z')
in degrees (1, 1, 2) over Rational Field with differential:

```

(continues on next page)

(continued from previous page)

```

x --> x*y
y --> -x*y
z --> 0
>>> B.cohomology(Integer(3))
Free module generated by {[x*z + y*z]} over Rational Field
>>> B.cohomology(Integer(4))
Free module generated by {[z^2]} over Rational Field

```

We can also compute algebra generators for the cohomology in a range of degrees, and in this case we compute up to degree 10:

```

sage: B.cohomology_generators(10)
{1: [x + y], 2: [z]}

```

```

>>> from sage.all import *
>>> B.cohomology_generators(Integer(10))
{1: [x + y], 2: [z]}

```

AUTHORS:

- Miguel Marco, John Palmieri (2014-07): initial version

class sage.algebras.commutative_dga.CohomologyClass(x, cdga=None)

Bases: SageObject, CachedRepresentation

A class for representing cohomology classes.

This just has `_repr_` and `_latex_` methods which put brackets around the object's name.

EXAMPLES:

```

sage: from sage.algebras.commutative_dga import CohomologyClass
sage: CohomologyClass(3)
[3]
sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(2,2,3,3))
sage: CohomologyClass(x^2 + 2*y*z, A)
[2*y*z + x^2]

```

```

>>> from sage.all import *
>>> from sage.algebras.commutative_dga import CohomologyClass
>>> CohomologyClass(Integer(3))
[3]
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(2), Integer(2), Integer(3),
↳ Integer(3)), names=('x', 'y', 'z', 't',)); (x, y, z, t) = A._first_ngens(4)
>>> CohomologyClass(x**Integer(2) + Integer(2)*y*z, A)
[2*y*z + x^2]

```

representative()

Return the representative of self.

EXAMPLES:

```

sage: from sage.algebras.commutative_dga import CohomologyClass
sage: x = CohomologyClass(sin)

```

#

(continues on next page)

(continued from previous page)

```

↪needs sage.symbolic
sage: x.representative() == sin
↪needs sage.symbolic
True

```

```

>>> from sage.all import *
>>> from sage.algebras.commutative_dga import CohomologyClass
>>> x = CohomologyClass(sin)
↪needs sage.symbolic
>>> x.representative() == sin
↪needs sage.symbolic
True

```

class sage.algebras.commutative_dga.**Differential** (*A*, *im_gens*)

Bases: UniqueRepresentation, Morphism

Differential of a commutative graded algebra.

INPUT:

- *A* – algebra where the differential is defined
- *im_gens* – tuple containing the image of each generator

EXAMPLES:

```

sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(1, 1, 2, 3))
sage: B = A.cdg_algebra({x: x*y, y: -x*y, z: t})
sage: B
Commutative Differential Graded Algebra with generators ('x', 'y', 'z', 't')
in degrees (1, 1, 2, 3) over Rational Field with differential:
  x --> x*y
  y --> -x*y
  z --> t
  t --> 0
sage: B.differential()(x)
x*y

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1), Integer(1), Integer(2),
↪Integer(3)), names=('x', 'y', 'z', 't')); (x, y, z, t) = A._first_ngens(4)
>>> B = A.cdg_algebra({x: x*y, y: -x*y, z: t})
>>> B
Commutative Differential Graded Algebra with generators ('x', 'y', 'z', 't')
in degrees (1, 1, 2, 3) over Rational Field with differential:
  x --> x*y
  y --> -x*y
  z --> t
  t --> 0
>>> B.differential()(x)
x*y

```

coboundaries (*n*)

The *n*-th coboundary group of the algebra.

This is a vector space over the base field F , and it is returned as a subspace of the vector space F^d , where the n -th homogeneous component has dimension d .

INPUT:

- n – degree

EXAMPLES:

```
sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(1, 1, 2))
sage: d = A.differential({z: x*z})
sage: d.coboundaries(2)
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
sage: d.coboundaries(3)
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 0]
sage: d.coboundaries(1)
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1), Integer(1),
↳ Integer(2)), names=('x', 'y', 'z',)); (x, y, z,) = A._first_ngens(3)
>>> d = A.differential({z: x*z})
>>> d.coboundaries(Integer(2))
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
>>> d.coboundaries(Integer(3))
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 0]
>>> d.coboundaries(Integer(1))
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
```

cocycles(n)

The n -th cocycle group of the algebra.

This is a vector space over the base field F , and it is returned as a subspace of the vector space F^d , where the n -th homogeneous component has dimension d .

INPUT:

- n – degree

EXAMPLES:

```
sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(1, 1, 2))
sage: d = A.differential({z: x*z})
sage: d.cocycles(2)
Vector space of degree 2 and dimension 1 over Rational Field
```

(continues on next page)

(continued from previous page)

```
Basis matrix:
[1 0]
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1), Integer(1),
↳Integer(2)), names=('x', 'y', 'z',)); (x, y, z,) = A._first_ngens(3)
>>> d = A.differential({z: x*z})
>>> d.cocycles(Integer(2))
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 0]
```

cohomology(*n*)

The *n*-th cohomology group of *self*.

This is a vector space over the base ring, defined as the quotient cocycles/coboundaries. The elements of the quotient are lifted to the vector space of cocycles, and this is described in terms of those lifts.

INPUT:

- *n* – degree

See also

[*cohomology_raw\(\)*](#)

EXAMPLES:

```
sage: A.<a,b,c,d,e> = GradedCommutativeAlgebra(QQ, degrees=(1, 1, 1, 1, 1))
sage: d = A.differential({d: a*b, e: b*c})
sage: d.cohomology(2)
Free module generated by {[a*c], [a*d], [b*d], [c*d - a*e], [b*e], [c*e]}
↳over Rational Field
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1), Integer(1),
↳Integer(1), Integer(1), Integer(1)), names=('a', 'b', 'c', 'd', 'e',)); (a,
↳b, c, d, e,) = A._first_ngens(5)
>>> d = A.differential({d: a*b, e: b*c})
>>> d.cohomology(Integer(2))
Free module generated by {[a*c], [a*d], [b*d], [c*d - a*e], [b*e], [c*e]}
↳over Rational Field
```

Compare to [*cohomology_raw\(\)*](#):

```
sage: d.cohomology_raw(2)
Vector space quotient V/W of dimension 6 over Rational Field where
V: Vector space of degree 10 and dimension 8 over Rational Field
Basis matrix:
[ 1  0  0  0  0  0  0  0  0  0]
[ 0  1  0  0  0  0  0  0  0  0]
[ 0  0  1  0  0  0  0  0  0  0]
```

(continues on next page)

(continued from previous page)

```
[ 0 0 0 1 0 0 0 0 0 0]
[ 0 0 0 0 1 0 0 0 0 0]
[ 0 0 0 0 0 1 -1 0 0 0]
[ 0 0 0 0 0 0 0 1 0 0]
[ 0 0 0 0 0 0 0 0 1 0]
```

W: Vector space of degree 10 and dimension 2 over Rational Field

Basis matrix:

```
[1 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0]
```

```
>>> from sage.all import *
```

```
>>> d.cohomology_raw(Integer(2))
```

Vector space quotient V/W of dimension 6 over Rational Field where

V: Vector space of degree 10 and dimension 8 over Rational Field

Basis matrix:

```
[ 1 0 0 0 0 0 0 0 0 0]
[ 0 1 0 0 0 0 0 0 0 0]
[ 0 0 1 0 0 0 0 0 0 0]
[ 0 0 0 1 0 0 0 0 0 0]
[ 0 0 0 0 1 0 0 0 0 0]
[ 0 0 0 0 0 1 -1 0 0 0]
[ 0 0 0 0 0 0 0 1 0 0]
[ 0 0 0 0 0 0 0 0 1 0]
```

W: Vector space of degree 10 and dimension 2 over Rational Field

Basis matrix:

```
[1 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0]
```

`cohomology_raw(n)`

The n -th cohomology group of `self`.

This is a vector space over the base ring, and it is returned as the quotient cocycles/coboundaries.

INPUT:

- n – degree

See also

[`cohomology\(\)`](#)

EXAMPLES:

```
sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(2, 2, 3, 4))
```

```
sage: d = A.differential({t: x*z, x: z, y: z})
```

```
sage: d.cohomology_raw(4)
```

Vector space quotient V/W of dimension 2 over Rational Field where

V: Vector space of degree 4 and dimension 2 over Rational Field

Basis matrix:

```
[ 1 0 0 -2]
[ 0 1 -1/2 -1]
```

W: Vector space of degree 4 and dimension 0 over Rational Field

(continues on next page)

(continued from previous page)

```
Basis matrix:
[]
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(2), Integer(2),
↳ Integer(3), Integer(4)), names=('x', 'y', 'z', 't')); (x, y, z, t) = A._
↳ first_ngens(4)
>>> d = A.differential({t: x*z, x: z, y: z})
>>> d.cohomology_raw(Integer(4))
Vector space quotient V/W of dimension 2 over Rational Field where
V: Vector space of degree 4 and dimension 2 over Rational Field
Basis matrix:
[ 1  0  0 -2]
[ 0  1 -1/2 -1]
W: Vector space of degree 4 and dimension 0 over Rational Field
Basis matrix:
[]
```

Compare to `cohomology()`:

```
sage: d.cohomology(4)
Free module generated by {[x^2 - 2*t], [x*y - 1/2*y^2 - t]} over Rational
↳ Field
```

```
>>> from sage.all import *
>>> d.cohomology(Integer(4))
Free module generated by {[x^2 - 2*t], [x*y - 1/2*y^2 - t]} over Rational
↳ Field
```

`differential_matrix(n)`

The matrix that gives the differential in degree n .

INPUT:

- n – degree

EXAMPLES:

```
sage: A.<x,y,z,t> = GradedCommutativeAlgebra(GF(5), degrees=(2, 2, 3, 4))
sage: d = A.differential({t: x*z, x: z, y: z})
sage: d.differential_matrix(4)
[2 0]
[1 1]
[0 2]
[1 0]
sage: A.inject_variables()
Defining x, y, z, t
sage: d(t)
x*z
sage: d(y^2)
2*y*z
sage: d(x*y)
x*z + y*z
```

(continues on next page)

(continued from previous page)

```
sage: d(x^2)
2*x*z
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(GF(Integer(5)), degrees=(Integer(2), Integer(2), Integer(3), Integer(4)), names=('x', 'y', 'z', 't')); (x, y, z, t) = A._first_ngens(4)
>>> d = A.differential({t: x*z, x: z, y: z})
>>> d.differential_matrix(Integer(4))
[2 0]
[1 1]
[0 2]
[1 0]
>>> A.inject_variables()
Defining x, y, z, t
>>> d(t)
x*z
>>> d(y**Integer(2))
2*y*z
>>> d(x*y)
x*z + y*z
>>> d(x**Integer(2))
2*x*z
```

homology (*n*)

The *n*-th cohomology group of `self`.

This is a vector space over the base ring, defined as the quotient cocycles/coboundaries. The elements of the quotient are lifted to the vector space of cocycles, and this is described in terms of those lifts.

INPUT:

- *n* – degree

See also

`cohomology_raw()`

EXAMPLES:

```
sage: A.<a,b,c,d,e> = GradedCommutativeAlgebra(QQ, degrees=(1, 1, 1, 1, 1))
sage: d = A.differential({d: a*b, e: b*c})
sage: d.cohomology(2)
Free module generated by {[a*c], [a*d], [b*d], [c*d - a*e], [b*e], [c*e]}
over Rational Field
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1), Integer(1), Integer(1), Integer(1), Integer(1)), names=('a', 'b', 'c', 'd', 'e')); (a, b, c, d, e) = A._first_ngens(5)
>>> d = A.differential({d: a*b, e: b*c})
>>> d.cohomology(Integer(2))
```

(continues on next page)

(continued from previous page)

```
Free module generated by {[a*c], [a*d], [b*d], [c*d - a*e], [b*e], [c*e]}_
↳ Over Rational Field
```

Compare to `cohomology_raw()`:

```
sage: d.cohomology_raw(2)
Vector space quotient V/W of dimension 6 over Rational Field where
V: Vector space of degree 10 and dimension 8 over Rational Field
Basis matrix:
[ 1  0  0  0  0  0  0  0  0  0  0]
[ 0  1  0  0  0  0  0  0  0  0  0]
[ 0  0  1  0  0  0  0  0  0  0  0]
[ 0  0  0  1  0  0  0  0  0  0  0]
[ 0  0  0  0  1  0  0  0  0  0  0]
[ 0  0  0  0  0  1 -1  0  0  0  0]
[ 0  0  0  0  0  0  0  1  0  0  0]
[ 0  0  0  0  0  0  0  0  1  0  0]
W: Vector space of degree 10 and dimension 2 over Rational Field
Basis matrix:
[1 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0]
```

```
>>> from sage.all import *
>>> d.cohomology_raw(Integer(2))
Vector space quotient V/W of dimension 6 over Rational Field where
V: Vector space of degree 10 and dimension 8 over Rational Field
Basis matrix:
[ 1  0  0  0  0  0  0  0  0  0  0]
[ 0  1  0  0  0  0  0  0  0  0  0]
[ 0  0  1  0  0  0  0  0  0  0  0]
[ 0  0  0  1  0  0  0  0  0  0  0]
[ 0  0  0  0  1  0  0  0  0  0  0]
[ 0  0  0  0  0  1 -1  0  0  0  0]
[ 0  0  0  0  0  0  0  1  0  0  0]
[ 0  0  0  0  0  0  0  0  1  0  0]
W: Vector space of degree 10 and dimension 2 over Rational Field
Basis matrix:
[1 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0]
```

class `sage.algebras.commutative_dga.DifferentialGCAgebra` (*A*, *differential*)

Bases: `GCAgebra`

A commutative differential graded algebra.

INPUT:

- *A* – a graded commutative algebra; that is, an instance of `GCAgebra`
- *differential* – a differential

As described in the module-level documentation, these are graded algebras for which oddly graded elements anti-commute and evenly graded elements commute, and on which there is a graded differential of degree 1.

These algebras should be graded over the integers; multi-graded algebras should be constructed using `DifferentialGCAgebra_multigraded` instead.

Note that a natural way to construct these is to use the `GradedCommutativeAlgebra()` function and the `GCAgebra.cdg_algebra()` method.

EXAMPLES:

```
sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(2, 2, 3, 3))
sage: A.cdg_algebra({z: x*y})
Commutative Differential Graded Algebra with generators ('x', 'y', 'z', 't')
in degrees (2, 2, 3, 3) over Rational Field with differential:
  x --> 0
  y --> 0
  z --> x*y
  t --> 0
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(2), Integer(2), Integer(3),
↳ Integer(3)), names=('x', 'y', 'z', 't',)); (x, y, z, t,) = A._first_ngens(4)
>>> A.cdg_algebra({z: x*y})
Commutative Differential Graded Algebra with generators ('x', 'y', 'z', 't')
in degrees (2, 2, 3, 3) over Rational Field with differential:
  x --> 0
  y --> 0
  z --> x*y
  t --> 0
```

Alternatively, starting with `GradedCommutativeAlgebra()`:

```
sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(2, 2, 3, 3))
sage: A.cdg_algebra(differential={z: x*y})
Commutative Differential Graded Algebra with generators ('x', 'y', 'z', 't')
in degrees (2, 2, 3, 3) over Rational Field with differential:
  x --> 0
  y --> 0
  z --> x*y
  t --> 0
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(2), Integer(2), Integer(3),
↳ Integer(3)), names=('x', 'y', 'z', 't',)); (x, y, z, t,) = A._first_ngens(4)
>>> A.cdg_algebra(differential={z: x*y})
Commutative Differential Graded Algebra with generators ('x', 'y', 'z', 't')
in degrees (2, 2, 3, 3) over Rational Field with differential:
  x --> 0
  y --> 0
  z --> x*y
  t --> 0
```

See the function `GradedCommutativeAlgebra()` for more examples.

class `Element` (`A`, `rep`)

Bases: `Element`

cohomology_class ()

Return the cohomology class of a homogeneous cycle, as an element of the corresponding cohomology group.

EXAMPLES:

```

sage: A.<e1,e2,e3,e4,e5> = GradedCommutativeAlgebra(QQ)
sage: B = A.cdg_algebra({e5: e1*e2+e3*e4})
sage: B.inject_variables()
Defining e1, e2, e3, e4, e5
sage: a = e1*e3*e5 - 3*e2*e3*e5
sage: a.cohomology_class()
B[[e1*e3*e5]] - 3*B[[e2*e3*e5]]

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, names=('e1', 'e2', 'e3', 'e4', 'e5',
↪)); (e1, e2, e3, e4, e5,) = A._first_ngens(5)
>>> B = A.cdg_algebra({e5: e1*e2+e3*e4})
>>> B.inject_variables()
Defining e1, e2, e3, e4, e5
>>> a = e1*e3*e5 - Integer(3)*e2*e3*e5
>>> a.cohomology_class()
B[[e1*e3*e5]] - 3*B[[e2*e3*e5]]

```

differential()

The differential on this element.

EXAMPLES:

```

sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(2, 2, 3, 4))
sage: B = A.cdg_algebra({t: x*z, x: z, y: z})
sage: B.inject_variables()
Defining x, y, z, t
sage: x.differential()
z
sage: (-1/2 * x^2 + t).differential()
0

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(2), Integer(2), ↪
↪Integer(3), Integer(4)), names=('x', 'y', 'z', 't')); (x, y, z, t,) = ↪
↪A._first_ngens(4)
>>> B = A.cdg_algebra({t: x*z, x: z, y: z})
>>> B.inject_variables()
Defining x, y, z, t
>>> x.differential()
z
>>> (-Integer(1)/Integer(2) * x**Integer(2) + t).differential()
0

```

is_coboundary()

Return True if self is a coboundary and False otherwise.

This raises an error if the element is not homogeneous.

EXAMPLES:

```

sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=(1,2,2))
sage: B = A.cdg_algebra(differential={b: a*c})
sage: x,y,z = B.gens()
sage: x.is_coboundary()
False
sage: (x*z).is_coboundary()
True
sage: (x*z + x*y).is_coboundary()
False
sage: (x*z + y**2).is_coboundary()
Traceback (most recent call last):
...
ValueError: this element is not homogeneous

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1),Integer(2),
↳Integer(2)), names=('a', 'b', 'c',)); (a, b, c,) = A._first_ngens(3)
>>> B = A.cdg_algebra(differential={b: a*c})
>>> x,y,z = B.gens()
>>> x.is_coboundary()
False
>>> (x*z).is_coboundary()
True
>>> (x*z + x*y).is_coboundary()
False
>>> (x*z + y**Integer(2)).is_coboundary()
Traceback (most recent call last):
...
ValueError: this element is not homogeneous

```

`is_cohomologous_to(other)`

Return True if self is cohomologous to other and False otherwise.

INPUT:

- other – another element of this algebra

EXAMPLES:

```

sage: A.<a,b,c,d> = GradedCommutativeAlgebra(QQ, degrees=(1,1,1,1))
sage: B = A.cdg_algebra(differential={a: b*c-c*d})
sage: w, x, y, z = B.gens()
sage: (x*y).is_cohomologous_to(y*z)
True
sage: (x*y).is_cohomologous_to(x*z)
False
sage: (x*y).is_cohomologous_to(x*y)
True

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1),Integer(1),
↳Integer(1),Integer(1)), names=('a', 'b', 'c', 'd',)); (a, b, c, d,) = A.
↳_first_ngens(4)
>>> B = A.cdg_algebra(differential={a: b*c-c*d})
>>> w, x, y, z = B.gens()

```

(continues on next page)

(continued from previous page)

```
>>> (x*y).is_cohomologous_to(y*z)
True
>>> (x*y).is_cohomologous_to(x*z)
False
>>> (x*y).is_cohomologous_to(x*y)
True
```

Two elements whose difference is not homogeneous are cohomologous if and only if they are both coboundaries:

```
sage: w.is_cohomologous_to(y*z)
False
sage: (x*y-y*z).is_cohomologous_to(x*y*z)
True
sage: (x*y*z).is_cohomologous_to(0) # make sure 0 works
True
```

```
>>> from sage.all import *
>>> w.is_cohomologous_to(y*z)
False
>>> (x*y-y*z).is_cohomologous_to(x*y*z)
True
>>> (x*y*z).is_cohomologous_to(Integer(0)) # make sure 0 works
True
```

cdg_algebra (*differential*)

Construct a differential graded commutative algebra from the underlying graded commutative algebra by specifying a differential. This may be used to get a new differential over the same algebra structure.

INPUT:

- `differential` – dictionary defining a differential or a map defining a valid differential

The keys of the dictionary are generators of the algebra, and the associated values are their targets under the differential. Any generators which are not specified are assumed to have zero differential. Alternatively, the differential can be defined using the `differential()` method; see below for an example.

See also

`differential()`

EXAMPLES:

```
sage: A.<x,y,z,t> = GradedCommutativeAlgebra(GF(5), degrees=(2, 3, 2, 4))
sage: B = A.quotient(A.ideal(x^3-z*t))
sage: C = B.cdg_algebra({y:t})
sage: C
Commutative Differential Graded Algebra with generators ('x', 'y', 'z', 't')
in degrees (2, 3, 2, 4) with relations [x^3 - z*t]
over Finite Field of size 5 with differential:
x --> 0
y --> t
z --> 0
```

(continues on next page)

(continued from previous page)

```

t --> 0
sage: C.cdg_algebra({})
Commutative Differential Graded Algebra with generators ('x', 'y', 'z', 't')
in degrees (2, 3, 2, 4) with relations [x^3 - z*t]
over Finite Field of size 5 with differential:
x --> 0
y --> 0
z --> 0
t --> 0

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(GF(Integer(5)), degrees=(Integer(2), Integer(3), Integer(2), Integer(4)), names=('x', 'y', 'z', 't')); (x, y, z, t) = A._first_ngens(4)
>>> B = A.quotient(A.ideal(x**Integer(3)-z*t))
>>> C = B.cdg_algebra({y:t})
>>> C
Commutative Differential Graded Algebra with generators ('x', 'y', 'z', 't')
in degrees (2, 3, 2, 4) with relations [x^3 - z*t]
over Finite Field of size 5 with differential:
x --> 0
y --> t
z --> 0
t --> 0
>>> C.cdg_algebra({})
Commutative Differential Graded Algebra with generators ('x', 'y', 'z', 't')
in degrees (2, 3, 2, 4) with relations [x^3 - z*t]
over Finite Field of size 5 with differential:
x --> 0
y --> 0
z --> 0
t --> 0

```

coboundaries (*n*)

The n -th coboundary group of the algebra.

This is a vector space over the base field F , and it is returned as a subspace of the vector space F^d , where the n -th homogeneous component has dimension d .

INPUT:

- n – degree

EXAMPLES:

```

sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(1,1,2))
sage: B = A.cdg_algebra(differential={z: x*z})
sage: B.coboundaries(2)
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
sage: B.coboundaries(3)
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:

```

(continues on next page)

(continued from previous page)

```
[1 0]
sage: B.basis(3)
[x*z, y*z]
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1), Integer(1),
↳ Integer(2)), names=('x', 'y', 'z',)); (x, y, z,) = A._first_ngens(3)
>>> B = A.cdg_algebra(differential={z: x*z})
>>> B.coboundaries(Integer(2))
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
>>> B.coboundaries(Integer(3))
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 0]
>>> B.basis(Integer(3))
[x*z, y*z]
```

cocycles (*n*)

The n -th cocycle group of the algebra.

This is a vector space over the base field F , and it is returned as a subspace of the vector space F^d , where the n -th homogeneous component has dimension d .

INPUT:

- n – degree

EXAMPLES:

```
sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(1,1,2))
sage: B = A.cdg_algebra(differential={z: x*z})
sage: B.cocycles(2)
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 0]
sage: B.basis(2)
[x*y, z]
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1), Integer(1),
↳ Integer(2)), names=('x', 'y', 'z',)); (x, y, z,) = A._first_ngens(3)
>>> B = A.cdg_algebra(differential={z: x*z})
>>> B.cocycles(Integer(2))
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 0]
>>> B.basis(Integer(2))
[x*y, z]
```

cohomology (*n*)

The n -th cohomology group of `self`.

This is a vector space over the base ring, defined as the quotient cocycles/coboundaries. The elements of the quotient are lifted to the vector space of cocycles, and this is described in terms of those lifts.

INPUT:

- n – degree

EXAMPLES:

```
sage: A.<a,b,c,d,e> = GradedCommutativeAlgebra(QQ, degrees=(1,1,1,1,1))
sage: B = A.cdg_algebra({d: a*b, e: b*c})
sage: B.cohomology(2)
Free module generated by {[a*c], [a*d], [b*d], [c*d - a*e], [b*e], [c*e]}
over Rational Field
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1),Integer(1),
↳Integer(1),Integer(1),Integer(1)), names=('a', 'b', 'c', 'd', 'e',)); (a, b,
↳c, d, e,) = A._first_ngens(5)
>>> B = A.cdg_algebra({d: a*b, e: b*c})
>>> B.cohomology(Integer(2))
Free module generated by {[a*c], [a*d], [b*d], [c*d - a*e], [b*e], [c*e]}
over Rational Field
```

Compare to `cohomology_raw()`:

```
sage: B.cohomology_raw(2)
Vector space quotient V/W of dimension 6 over Rational Field where
V: Vector space of degree 10 and dimension 8 over Rational Field
Basis matrix:
[ 1  0  0  0  0  0  0  0  0  0]
[ 0  1  0  0  0  0  0  0  0  0]
[ 0  0  1  0  0  0  0  0  0  0]
[ 0  0  0  1  0  0  0  0  0  0]
[ 0  0  0  0  1  0  0  0  0  0]
[ 0  0  0  0  0  1 -1  0  0  0]
[ 0  0  0  0  0  0  0  1  0  0]
[ 0  0  0  0  0  0  0  0  1  0]
W: Vector space of degree 10 and dimension 2 over Rational Field
Basis matrix:
[1 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0]
```

```
>>> from sage.all import *
>>> B.cohomology_raw(Integer(2))
Vector space quotient V/W of dimension 6 over Rational Field where
V: Vector space of degree 10 and dimension 8 over Rational Field
Basis matrix:
[ 1  0  0  0  0  0  0  0  0  0]
[ 0  1  0  0  0  0  0  0  0  0]
[ 0  0  1  0  0  0  0  0  0  0]
[ 0  0  0  1  0  0  0  0  0  0]
[ 0  0  0  0  1  0  0  0  0  0]
[ 0  0  0  0  0  1 -1  0  0  0]
[ 0  0  0  0  0  0  0  1  0  0]
```

(continues on next page)

(continued from previous page)

```
[ 0 0 0 0 0 0 0 0 0 1 0]
W: Vector space of degree 10 and dimension 2 over Rational Field
Basis matrix:
[1 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0]
```

cohomology_algebra (*max_degree=3*)Compute a CDGA with trivial differential, that is isomorphic to the cohomology of `self` up to `max_degree`

INPUT:

- `max_degree` – integer (default: 3); degree to which the result is required to be isomorphic to `self`'s cohomology

EXAMPLES:

```
sage: A.<e1, e2, e3, e4, e5, e6, e7> = GradedCommutativeAlgebra(QQ)
sage: d = A.differential({e1:-e1*e6, e2:-e2*e6, e3:-e3*e6, e4:-e5*e6,
↪ e5:e4*e6})
sage: B = A.cdg_algebra(d)
sage: M = B.cohomology_algebra()
sage: M
Commutative Differential Graded Algebra with generators ('x0', 'x1', 'x2')
in degrees (1, 1, 2) over Rational Field with differential:
x0 --> 0
x1 --> 0
x2 --> 0
sage: M.cohomology(1)
Free module generated by {[x0], [x1]} over Rational Field
sage: B.cohomology(1)
Free module generated by {[e6], [e7]} over Rational Field
sage: M.cohomology(2)
Free module generated by {[x0*x1], [x2]} over Rational Field
sage: B.cohomology(2)
Free module generated by {[e4*e5], [e6*e7]} over Rational Field
sage: M.cohomology(3)
Free module generated by {[x0*x2], [x1*x2]} over Rational Field
sage: B.cohomology(3)
Free module generated by {[e4*e5*e6], [e4*e5*e7]} over Rational Field
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, names=('e1', 'e2', 'e3', 'e4', 'e5', 'e6',
↪ 'e7',)); (e1, e2, e3, e4, e5, e6, e7,) = A._first_ngens(7)
>>> d = A.differential({e1:-e1*e6, e2:-e2*e6, e3:-e3*e6, e4:-e5*e6, e5:e4*e6})
>>> B = A.cdg_algebra(d)
>>> M = B.cohomology_algebra()
>>> M
Commutative Differential Graded Algebra with generators ('x0', 'x1', 'x2')
in degrees (1, 1, 2) over Rational Field with differential:
x0 --> 0
x1 --> 0
x2 --> 0
>>> M.cohomology(Integer(1))
```

(continues on next page)

(continued from previous page)

```

Free module generated by {[x0], [x1]} over Rational Field
>>> B.cohomology(Integer(1))
Free module generated by {[e6], [e7]} over Rational Field
>>> M.cohomology(Integer(2))
Free module generated by {[x0*x1], [x2]} over Rational Field
>>> B.cohomology(Integer(2))
Free module generated by {[e4*e5], [e6*e7]} over Rational Field
>>> M.cohomology(Integer(3))
Free module generated by {[x0*x2], [x1*x2]} over Rational Field
>>> B.cohomology(Integer(3))
Free module generated by {[e4*e5*e6], [e4*e5*e7]} over Rational Field

```

cohomology_generators (*max_degree*)

Return lifts of algebra generators for cohomology in degrees at most *max_degree*.

INPUT:

- *max_degree* – integer

OUTPUT:

A dictionary keyed by degree, where the corresponding value is a list of cohomology generators in that degree. Actually, the elements are lifts of cohomology generators, which means that they lie in this differential graded algebra. It also means that they are only well-defined up to cohomology, not on the nose.

ALGORITHM:

Reduce a basis of the n -th cohomology modulo all the degree n products of the lower degree cohomologies.

EXAMPLES:

```

sage: A.<a,x,y> = GradedCommutativeAlgebra(QQ, degrees=(1,2,2))
sage: B = A.cdg_algebra(differential={y: a*x})
sage: B.cohomology_generators(3)
{1: [a], 2: [x], 3: [a*y]}

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1),Integer(2),
↳Integer(2)), names=('a', 'x', 'y',)); (a, x, y,) = A._first_ngens(3)
>>> B = A.cdg_algebra(differential={y: a*x})
>>> B.cohomology_generators(Integer(3))
{1: [a], 2: [x], 3: [a*y]}

```

The previous example has infinitely generated cohomology: ay^n is a cohomology generator for each n :

```

sage: B.cohomology_generators(10)
{1: [a], 2: [x], 3: [a*y], 5: [a*y^2], 7: [a*y^3], 9: [a*y^4]}

```

```

>>> from sage.all import *
>>> B.cohomology_generators(Integer(10))
{1: [a], 2: [x], 3: [a*y], 5: [a*y^2], 7: [a*y^3], 9: [a*y^4]}

```

In contrast, the corresponding algebra in characteristic p has finitely generated cohomology:

```

sage: A3.<a,x,y> = GradedCommutativeAlgebra(GF(3), degrees=(1,2,2))
sage: B3 = A3.cdg_algebra(differential={y: a*x})
sage: B3.cohomology_generators(16)
{1: [a], 2: [x], 3: [a*y], 5: [a*y^2], 6: [y^3]}

```

```

>>> from sage.all import *
>>> A3 = GradedCommutativeAlgebra(GF(Integer(3)), degrees=(Integer(1),
↳Integer(2),Integer(2)), names=('a', 'x', 'y')); (a, x, y) = A3._first_
↳ngens(3)
>>> B3 = A3.cdg_algebra(differential={y: a*x})
>>> B3.cohomology_generators(Integer(16))
{1: [a], 2: [x], 3: [a*y], 5: [a*y^2], 6: [y^3]}

```

This method works with both singly graded and multi-graded algebras:

```

sage: Cs.<a,b,c,d> = GradedCommutativeAlgebra(GF(2), degrees=(1,2,2,3))
sage: Ds = Cs.cdg_algebra({a:c, b:d})
sage: Ds.cohomology_generators(10)
{2: [a^2], 4: [b^2]}

```

```

sage: Cm.<a,b,c,d> = GradedCommutativeAlgebra(GF(2),
.....:                                     degrees=((1,0), (1,1),
.....:                                     (0,2), (0,3)))
sage: Dm = Cm.cdg_algebra({a:c, b:d})
sage: Dm.cohomology_generators(10)
{2: [a^2], 4: [b^2]}

```

```

>>> from sage.all import *
>>> Cs = GradedCommutativeAlgebra(GF(Integer(2)), degrees=(Integer(1),
↳Integer(2),Integer(2),Integer(3)), names=('a', 'b', 'c', 'd')); (a, b, c,
↳d) = Cs._first_ngens(4)
>>> Ds = Cs.cdg_algebra({a:c, b:d})
>>> Ds.cohomology_generators(Integer(10))
{2: [a^2], 4: [b^2]}

>>> Cm = GradedCommutativeAlgebra(GF(Integer(2)),
...                               degrees=((Integer(1),Integer(0)),
↳Integer(1),Integer(1)),
...                               (Integer(0),Integer(2)),
↳Integer(0),Integer(3))), names=('a', 'b', 'c', 'd')); (a, b, c, d) = Cm._
↳first_ngens(4)
>>> Dm = Cm.cdg_algebra({a:c, b:d})
>>> Dm.cohomology_generators(Integer(10))
{2: [a^2], 4: [b^2]}

```

cohomology_raw(*n*)

The *n*-th cohomology group of *self*.

This is a vector space over the base ring, and it is returned as the quotient cocycles/coboundaries.

INPUT:

- *n* – degree

EXAMPLES:

```

sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees = (2,2,3,4))
sage: B = A.cdg_algebra({t: x*z, x: z, y: z})
sage: B.cohomology_raw(4)
Vector space quotient V/W of dimension 2 over Rational Field where
V: Vector space of degree 4 and dimension 2 over Rational Field
Basis matrix:
[ 1  0  0 -2]
[ 0  1 -1/2 -1]
W: Vector space of degree 4 and dimension 0 over Rational Field
Basis matrix:
[]

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees = (Integer(2),Integer(2),
↳Integer(3),Integer(4)), names=('x', 'y', 'z', 't')); (x, y, z, t) = A._
↳first_ngens(4)
>>> B = A.cdg_algebra({t: x*z, x: z, y: z})
>>> B.cohomology_raw(Integer(4))
Vector space quotient V/W of dimension 2 over Rational Field where
V: Vector space of degree 4 and dimension 2 over Rational Field
Basis matrix:
[ 1  0  0 -2]
[ 0  1 -1/2 -1]
W: Vector space of degree 4 and dimension 0 over Rational Field
Basis matrix:
[]

```

Compare to `cohomology()`:

```

sage: B.cohomology(4)
Free module generated by {[x^2 - 2*t], [x*y - 1/2*y^2 - t]} over Rational_
↳Field

```

```

>>> from sage.all import *
>>> B.cohomology(Integer(4))
Free module generated by {[x^2 - 2*t], [x*y - 1/2*y^2 - t]} over Rational_
↳Field

```

differential (*x=None*)

The differential of self.

This returns a map, and so it may be evaluated on elements of this algebra.

EXAMPLES:

```

sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(1,1,2))
sage: B = A.cdg_algebra({y:x*y, x: y*x})
sage: d = B.differential(); d
Differential of Commutative Differential Graded Algebra
with generators ('x', 'y', 'z') in degrees (1, 1, 2) over Rational Field
Defn: x --> -x*y
      y --> x*y
      z --> 0

```

(continues on next page)

(continued from previous page)

```
sage: d(y)
x*y
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1), Integer(1),
↳ Integer(2)), names=('x', 'y', 'z',)); (x, y, z,) = A._first_ngens(3)
>>> B = A.cdg_algebra({y:x*y, x: y*x})
>>> d = B.differential(); d
Differential of Commutative Differential Graded Algebra
with generators ('x', 'y', 'z') in degrees (1, 1, 2) over Rational Field
Defn: x --> -x*y
      y --> x*y
      z --> 0
>>> d(y)
x*y
```

graded_commutative_algebra()

Return the base graded commutative algebra of self.

EXAMPLES:

```
sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(2, 2, 3, 3))
sage: D = A.cdg_algebra({z: x*y})
sage: D.graded_commutative_algebra() == A
True
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(2), Integer(2),
↳ Integer(3), Integer(3)), names=('x', 'y', 'z', 't',)); (x, y, z, t,) = A._
↳ first_ngens(4)
>>> D = A.cdg_algebra({z: x*y})
>>> D.graded_commutative_algebra() == A
True
```

homology(n)

The n -th cohomology group of self.

This is a vector space over the base ring, defined as the quotient cocycles/coboundaries. The elements of the quotient are lifted to the vector space of cocycles, and this is described in terms of those lifts.

INPUT:

- n – degree

EXAMPLES:

```
sage: A.<a,b,c,d,e> = GradedCommutativeAlgebra(QQ, degrees=(1,1,1,1,1))
sage: B = A.cdg_algebra({d: a*b, e: b*c})
sage: B.cohomology(2)
Free module generated by {[a*c], [a*d], [b*d], [c*d - a*e], [b*e], [c*e]}
over Rational Field
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1), Integer(1),
```

(continues on next page)

(continued from previous page)

```

↪ Integer(1), Integer(1), Integer(1)), names=('a', 'b', 'c', 'd', 'e',)); (a, b,
↪ c, d, e,) = A._first_ngens(5)
>>> B = A.cdg_algebra({d: a*b, e: b*c})
>>> B.cohomology(Integer(2))
Free module generated by {[a*c], [a*d], [b*d], [c*d - a*e], [b*e], [c*e]}
over Rational Field

```

Compare to `cohomology_raw()`:

```

sage: B.cohomology_raw(2)
Vector space quotient V/W of dimension 6 over Rational Field where
V: Vector space of degree 10 and dimension 8 over Rational Field
Basis matrix:
[ 1  0  0  0  0  0  0  0  0  0  0]
[ 0  1  0  0  0  0  0  0  0  0  0]
[ 0  0  1  0  0  0  0  0  0  0  0]
[ 0  0  0  1  0  0  0  0  0  0  0]
[ 0  0  0  0  1  0  0  0  0  0  0]
[ 0  0  0  0  0  1  -1  0  0  0  0]
[ 0  0  0  0  0  0  0  1  0  0  0]
[ 0  0  0  0  0  0  0  0  1  0  0]
W: Vector space of degree 10 and dimension 2 over Rational Field
Basis matrix:
[1 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0]

```

```

>>> from sage.all import *
>>> B.cohomology_raw(Integer(2))
Vector space quotient V/W of dimension 6 over Rational Field where
V: Vector space of degree 10 and dimension 8 over Rational Field
Basis matrix:
[ 1  0  0  0  0  0  0  0  0  0  0]
[ 0  1  0  0  0  0  0  0  0  0  0]
[ 0  0  1  0  0  0  0  0  0  0  0]
[ 0  0  0  1  0  0  0  0  0  0  0]
[ 0  0  0  0  1  0  0  0  0  0  0]
[ 0  0  0  0  0  1  -1  0  0  0  0]
[ 0  0  0  0  0  0  0  1  0  0  0]
[ 0  0  0  0  0  0  0  0  1  0  0]
W: Vector space of degree 10 and dimension 2 over Rational Field
Basis matrix:
[1 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0]

```

is_formal(*i*, *max_iterations*=3)

Check if the algebra is *i*-formal. That is, if it is *i*-quasi-isomorphic to its cohomology algebra.

INPUT:

- *i* – integer; the degree up to which the formality is checked
- *max_iterations* – integer (default: 3); the maximum number of iterations used in the computation of the minimal model

Warning

The method is not granted to finish (it can't, since the minimal model could be infinitely generated in some degrees). The parameter `max_iterations` controls how many iterations of the method are attempted at each degree. In case they are not enough, an exception is raised. If you think that the result will be finitely generated, you can try to run it again with a higher value for `max_iterations`.

Moreover, the method uses criteria that are often enough to conclude that the algebra is either formal or non-formal. However, it could happen that the used criteria can not determine the formality. In that case, an error is raised.

EXAMPLES:

```
sage: A.<e1, e2, e3, e4, e5> = GradedCommutativeAlgebra(QQ)
sage: B = A.cdg_algebra({e5: e1*e2 + e3*e4})
sage: B.is_formal(1)
True
sage: B.is_formal(2)
False
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, names=('e1', 'e2', 'e3', 'e4', 'e5'));
↳ (e1, e2, e3, e4, e5,) = A._first_ngens(5)
>>> B = A.cdg_algebra({e5: e1*e2 + e3*e4})
>>> B.is_formal(Integer(1))
True
>>> B.is_formal(Integer(2))
False
```

ALGORITHM:

Apply the criteria in [Man2019]. Both the i -minimal model of the algebra and its cohomology algebra are computed. If the numerical invariants are different, the algebra is not i -formal.

If the numerical invariants match, the ψ condition is checked.

minimal_model ($i=3$, $max\_iterations=3$, $partial\_result=False$)

Try to compute a map from a i -minimal gcda that is a i -quasi-isomorphism to self.

INPUT:

- i – integer (default: 3); degree to which the result is required to induce an isomorphism in cohomology, and the domain is required to be minimal
- `max_iterations` – integer (default: 3); the number of iterations of the method at each degree. If the algorithm does not finish in this many iterations at each degree, an error is raised, or the partial result computed up to that point is returned, depending on the `partial_result` flag.
- `partial_result` – boolean (default: False); whether to return the partial result if the `max_iterations` limit is reached

OUTPUT:

A morphism from a minimal Sullivan (up to degree i) CDGA's to self, that induces an isomorphism in cohomology up to degree i , and a monomorphism in degree $i+1$.

EXAMPLES:

```

sage: S.<x, y, z> = GradedCommutativeAlgebra(QQ, degrees = (1, 1, 2))
sage: d = S.differential({x:x*y, y:x*y})
sage: R = S.cdg_algebra(d)
sage: p = R.minimal_model()
sage: T = p.domain()
sage: p
Commutative Differential Graded Algebra morphism:
  From: Commutative Differential Graded Algebra
        with generators ('x1_0', 'x2_0') in degrees (1, 2)
        over Rational Field with differential:
            x1_0 --> 0
            x2_0 --> 0
  To:    Commutative Differential Graded Algebra
        with generators ('x', 'y', 'z') in degrees (1, 1, 2)
        over Rational Field with differential:
            x --> x*y
            y --> x*y
            z --> 0
  Defn: (x1_0, x2_0) --> (x - y, z)
sage: R.cohomology(1)
Free module generated by {[x - y]} over Rational Field
sage: T.cohomology(1)
Free module generated by {[x1_0]} over Rational Field
sage: [p(g.representative()) for g in T.cohomology(1).basis().keys()]
[x - y]
sage: R.cohomology(2)
Free module generated by {[z]} over Rational Field
sage: T.cohomology(2)
Free module generated by {[x2_0]} over Rational Field
sage: [p(g.representative()) for g in T.cohomology(2).basis().keys()]
[z]

sage: A.<e1, e2, e3, e4, e5, e6, e7> = GradedCommutativeAlgebra(QQ)
sage: d = A.differential({e1:e1*e7, e2:e2*e7, e3:-e3*e7, e4:-e4*e7})
sage: B = A.cdg_algebra(d)
sage: phi = B.minimal_model(i=3)
sage: M = phi.domain()
sage: M
Commutative Differential Graded Algebra with generators
('x1_0', 'x1_1', 'x1_2', 'x2_0', 'x2_1', 'x2_2', 'x2_3',
 'y3_0', 'y3_1', 'y3_2', 'y3_3', 'y3_4', 'y3_5', 'y3_6', 'y3_7', 'y3_8')
in degrees (1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3)
over Rational Field with differential:
  x1_0 --> 0
  x1_1 --> 0
  x1_2 --> 0
  x2_0 --> 0
  x2_1 --> 0
  x2_2 --> 0
  x2_3 --> 0
  y3_0 --> x2_0^2
  y3_1 --> x2_0*x2_1

```

(continues on next page)

(continued from previous page)

```

y3_2 --> x2_1^2
y3_3 --> x2_0*x2_2
y3_4 --> x2_1*x2_2 + x2_0*x2_3
y3_5 --> x2_2^2
y3_6 --> x2_1*x2_3
y3_7 --> x2_2*x2_3
y3_8 --> x2_3^2

sage: phi
Commutative Differential Graded Algebra morphism:
  From: Commutative Differential Graded Algebra with generators
        ('x1_0', 'x1_1', 'x1_2', 'x2_0', 'x2_1', 'x2_2', 'x2_3',
         'y3_0', 'y3_1', 'y3_2', 'y3_3', 'y3_4', 'y3_5', 'y3_6', 'y3_7', 'y3_8
↪')
        in degrees (1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3)
        over Rational Field with differential:
          x1_0 --> 0
          x1_1 --> 0
          x1_2 --> 0
          x2_0 --> 0
          x2_1 --> 0
          x2_2 --> 0
          x2_3 --> 0
          y3_0 --> x2_0^2
          y3_1 --> x2_0*x2_1
          y3_2 --> x2_1^2
          y3_3 --> x2_0*x2_2
          y3_4 --> x2_1*x2_2 + x2_0*x2_3
          y3_5 --> x2_2^2
          y3_6 --> x2_1*x2_3
          y3_7 --> x2_2*x2_3
          y3_8 --> x2_3^2
  To:   Commutative Differential Graded Algebra with generators
        ('e1', 'e2', 'e3', 'e4', 'e5', 'e6', 'e7')
        in degrees (1, 1, 1, 1, 1, 1, 1) over Rational Field with
↪differential:
          e1 --> e1*e7
          e2 --> e2*e7
          e3 --> -e3*e7
          e4 --> -e4*e7
          e5 --> 0
          e6 --> 0
          e7 --> 0
  Defn: (x1_0, x1_1, x1_2, x2_0, x2_1, x2_2, x2_3,
         y3_0, y3_1, y3_2, y3_3, y3_4, y3_5, y3_6, y3_7, y3_8)
        --> (e5, e6, e7, e1*e3, e2*e3, e1*e4, e2*e4, 0, 0, 0, 0, 0, 0, 0, 0,
↪0)
sage: [B.cohomology(i).dimension() for i in [1..3]]
[3, 7, 13]
sage: [M.cohomology(i).dimension() for i in [1..3]]
[3, 7, 13]

```



```

>>> from sage.all import *
>>> S = GradedCommutativeAlgebra(QQ, degrees = (Integer(1), Integer(1),
↳ Integer(2)), names=('x', 'y', 'z')); (x, y, z) = S._first_ngens(3)
>>> d = S.differential({x:x*y, y:x*y})
>>> R = S.cdg_algebra(d)
>>> p = R.minimal_model()
>>> T = p.domain()
>>> p
Commutative Differential Graded Algebra morphism:
  From: Commutative Differential Graded Algebra
        with generators ('x1_0', 'x2_0') in degrees (1, 2)
        over Rational Field with differential:
            x1_0 --> 0
            x2_0 --> 0
  To:   Commutative Differential Graded Algebra
        with generators ('x', 'y', 'z') in degrees (1, 1, 2)
        over Rational Field with differential:
            x --> x*y
            y --> x*y
            z --> 0
  Defn: (x1_0, x2_0) --> (x - y, z)
>>> R.cohomology(Integer(1))
Free module generated by {[x - y]} over Rational Field
>>> T.cohomology(Integer(1))
Free module generated by {[x1_0]} over Rational Field
>>> [p(g.representative()) for g in T.cohomology(Integer(1)).basis().keys()]
[x - y]
>>> R.cohomology(Integer(2))
Free module generated by {[z]} over Rational Field
>>> T.cohomology(Integer(2))
Free module generated by {[x2_0]} over Rational Field
>>> [p(g.representative()) for g in T.cohomology(Integer(2)).basis().keys()]
[z]

>>> A = GradedCommutativeAlgebra(QQ, names=('e1', 'e2', 'e3', 'e4', 'e5', 'e6
↳ ', 'e7')); (e1, e2, e3, e4, e5, e6, e7) = A._first_ngens(7)
>>> d = A.differential({e1:e1*e7, e2:e2*e7, e3:-e3*e7, e4:-e4*e7})
>>> B = A.cdg_algebra(d)
>>> phi = B.minimal_model(i=Integer(3))
>>> M = phi.domain()
>>> M
Commutative Differential Graded Algebra with generators
('x1_0', 'x1_1', 'x1_2', 'x2_0', 'x2_1', 'x2_2', 'x2_3',
 'y3_0', 'y3_1', 'y3_2', 'y3_3', 'y3_4', 'y3_5', 'y3_6', 'y3_7', 'y3_8')
in degrees (1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3)
over Rational Field with differential:
  x1_0 --> 0
  x1_1 --> 0
  x1_2 --> 0
  x2_0 --> 0
  x2_1 --> 0
  x2_2 --> 0

```

(continues on next page)

(continued from previous page)

```

x2_3 --> 0
y3_0 --> x2_0^2
y3_1 --> x2_0*x2_1
y3_2 --> x2_1^2
y3_3 --> x2_0*x2_2
y3_4 --> x2_1*x2_2 + x2_0*x2_3
y3_5 --> x2_2^2
y3_6 --> x2_1*x2_3
y3_7 --> x2_2*x2_3
y3_8 --> x2_3^2

>>> phi
Commutative Differential Graded Algebra morphism:
  From: Commutative Differential Graded Algebra with generators
        ('x1_0', 'x1_1', 'x1_2', 'x2_0', 'x2_1', 'x2_2', 'x2_3',
         'y3_0', 'y3_1', 'y3_2', 'y3_3', 'y3_4', 'y3_5', 'y3_6', 'y3_7', 'y3_8
↪')
        in degrees (1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3)
        over Rational Field with differential:
          x1_0 --> 0
          x1_1 --> 0
          x1_2 --> 0
          x2_0 --> 0
          x2_1 --> 0
          x2_2 --> 0
          x2_3 --> 0
          y3_0 --> x2_0^2
          y3_1 --> x2_0*x2_1
          y3_2 --> x2_1^2
          y3_3 --> x2_0*x2_2
          y3_4 --> x2_1*x2_2 + x2_0*x2_3
          y3_5 --> x2_2^2
          y3_6 --> x2_1*x2_3
          y3_7 --> x2_2*x2_3
          y3_8 --> x2_3^2
  To:    Commutative Differential Graded Algebra with generators
        ('e1', 'e2', 'e3', 'e4', 'e5', 'e6', 'e7')
        in degrees (1, 1, 1, 1, 1, 1, 1) over Rational Field with
↪differential:
          e1 --> e1*e7
          e2 --> e2*e7
          e3 --> -e3*e7
          e4 --> -e4*e7
          e5 --> 0
          e6 --> 0
          e7 --> 0
  Defn: (x1_0, x1_1, x1_2, x2_0, x2_1, x2_2, x2_3,
        y3_0, y3_1, y3_2, y3_3, y3_4, y3_5, y3_6, y3_7, y3_8)
        --> (e5, e6, e7, e1*e3, e2*e3, e1*e4, e2*e4, 0, 0, 0, 0, 0, 0, 0, 0,
↪0)
>>> [B.cohomology(i).dimension() for i in (ellipsis_range(Integer(1),Ellipsis,
↪Integer(3)))]

```

(continues on next page)

(continued from previous page)

```
[3, 7, 13]
>>> [M.cohomology(i).dimension() for i in (ellipsis_range(Integer(1), Ellipsis,
↪Integer(3)))]
[3, 7, 13]
```

ALGORITHM:

We follow the algorithm described in [Man2019]. It consists in constructing the minimal Sullivan algebra S by iteratively adding generators to it. Start with one closed generator of degree 1 for each element in the basis of the first cohomology of the algebra. Then proceed degree by degree. At each degree d , we keep adding generators of degree $d-1$ whose differential kills the elements in the kernel of the map $H^d(S) \rightarrow H^d(\text{self})$. Once this map is made injective, we add the needed closed generators in degree d to make it surjective.

Warning

The method is not granted to finish (it can't, since the minimal model could be infinitely generated in some degrees). The parameter `max_iterations` controls how many iterations of the method are attempted at each degree. In case they are not enough, an exception is raised. If you think that the result will be finitely generated, you can try to run it again with a higher value for `max_iterations`.

See also

[Wikipedia article Rational_homotopy_theory#Sullivan_algebras](#)

REFERENCES:

- [Fel2001]
- [Man2019]

numerical_invariants (*max_degree=3, max_iterations=3*)

Return the numerical invariants of the algebra, up to degree d . The numerical invariants reflect the number of generators added at each step of the construction of the minimal model.

The numerical invariants are the dimensions of the subsequent Hirsch extensions used at each degree to compute the minimal model.

INPUT:

- `max_degree` – integer (default: 3); the degree up to which the numerical invariants are computed
- `max_iterations` – integer (default: 3); the maximum number of iterations used to compute the minimal model, if it is not already cached

EXAMPLES:

```
sage: A.<e1, e2, e3> = GradedCommutativeAlgebra(QQ)
sage: B = A.cdg_algebra({e3 : e1*e2})
sage: B.minimal_model(4)
Commutative Differential Graded Algebra morphism:
From: Commutative Differential Graded Algebra with
      generators ('x1_0', 'x1_1', 'y1_0') in degrees (1, 1, 1)
      over Rational Field with differential:
      x1_0 --> 0
```

(continues on next page)

(continued from previous page)

```

x1_1 --> 0
y1_0 --> x1_0*x1_1
To: Commutative Differential Graded Algebra with
generators ('e1', 'e2', 'e3') in degrees (1, 1, 1)
over Rational Field with differential:
e1 --> 0
e2 --> 0
e3 --> e1*e2
Defn: (x1_0, x1_1, y1_0) --> (e1, e2, e3)
sage: B.numerical_invariants(2)
{1: [2, 1, 0], 2: [0, 0]}

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, names=('e1', 'e2', 'e3',)); (e1, e2, e3,
↪) = A._first_ngens(3)
>>> B = A.cdg_algebra({e3 : e1*e2})
>>> B.minimal_model(Integer(4))
Commutative Differential Graded Algebra morphism:
From: Commutative Differential Graded Algebra with
generators ('x1_0', 'x1_1', 'y1_0') in degrees (1, 1, 1)
over Rational Field with differential:
x1_0 --> 0
x1_1 --> 0
y1_0 --> x1_0*x1_1
To: Commutative Differential Graded Algebra with
generators ('e1', 'e2', 'e3') in degrees (1, 1, 1)
over Rational Field with differential:
e1 --> 0
e2 --> 0
e3 --> e1*e2
Defn: (x1_0, x1_1, y1_0) --> (e1, e2, e3)
>>> B.numerical_invariants(Integer(2))
{1: [2, 1, 0], 2: [0, 0]}

```

ALGORITHM:

The numerical invariants are stored as the minimal model is constructed.

Warning

The method is not granted to finish (it can't, since the minimal model could be infinitely generated in some degrees). The parameter `max_iterations` controls how many iterations of the method are attempted at each degree. In case they are not enough, an exception is raised. If you think that the result will be finitely generated, you can try to run it again with a higher value for `max_iterations`.

REFERENCES:

For a precise definition and properties, see [Man2019] .

quotient (*I*, *check=True*)

Create the quotient of this algebra by a two-sided ideal $\mathbb{I}$.

INPUT:

- I – a two-sided homogeneous ideal of this algebra
- `check` – boolean (default: `True`); if `True`, check whether I is generated by homogeneous elements

EXAMPLES:

```
sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(1,1,2))
sage: B = A.cdg_algebra({y:x*y, z:x*z})
sage: B.inject_variables()
Defining x, y, z
sage: I = B.ideal([y*z])
sage: C = B.quotient(I)
sage: (y*z).differential()
2*x*y*z
sage: C((y*z).differential())
0
sage: C(y*z)
0
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1),Integer(1),
↳Integer(2)), names=('x', 'y', 'z',)); (x, y, z,) = A._first_ngens(3)
>>> B = A.cdg_algebra({y:x*y, z:x*z})
>>> B.inject_variables()
Defining x, y, z
>>> I = B.ideal([y*z])
>>> C = B.quotient(I)
>>> (y*z).differential()
2*x*y*z
>>> C((y*z).differential())
0
>>> C(y*z)
0
```

It is checked that the differential maps the ideal into itself, to make sure that the quotient inherits a differential structure:

```
sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(1,2,2))
sage: B = A.cdg_algebra({x:y})
sage: B.quotient(B.ideal(y*x))
Traceback (most recent call last):
...
ValueError: the differential does not preserve the ideal
sage: B.quotient(B.ideal(x))
Traceback (most recent call last):
...
ValueError: the differential does not preserve the ideal
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1),Integer(2),
↳Integer(2)), names=('x', 'y', 'z',)); (x, y, z,) = A._first_ngens(3)
>>> B = A.cdg_algebra({x:y})
>>> B.quotient(B.ideal(y*x))
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```

...
ValueError: the differential does not preserve the ideal
>>> B.quotient(B.ideal(x))
Traceback (most recent call last):
...
ValueError: the differential does not preserve the ideal

```

```

class sage.algebras.commutative_dga.DifferentialGCAgebra_multigraded(A, differential)
    Bases: DifferentialGCAgebra, GCAgebra_multigraded

```

A commutative differential multi-graded algebras.

INPUT:

- A – a commutative multi-graded algebra
- differential – a differential

EXAMPLES:

```

sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1,0), (0,1), (0,2)))
sage: B = A.cdg_algebra(differential={a: c})
sage: B.basis((1,0))
[a]
sage: B.basis(1, total=True)
[a, b]
sage: B.cohomology((1,0))
Free module generated by {} over Rational Field
sage: B.cohomology(1, total=True)
Free module generated by {[b]} over Rational Field

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=((Integer(1),Integer(0)),
→ (Integer(0), Integer(1)), (Integer(0),Integer(2))), names=('a', 'b', 'c')); (a,
→ b, c,) = A._first_ngens(3)
>>> B = A.cdg_algebra(differential={a: c})
>>> B.basis((Integer(1),Integer(0)))
[a]
>>> B.basis(Integer(1), total=True)
[a, b]
>>> B.cohomology((Integer(1), Integer(0)))
Free module generated by {} over Rational Field
>>> B.cohomology(Integer(1), total=True)
Free module generated by {[b]} over Rational Field

```

```

class Element(A, rep)

```

Bases: *Element*, *Element*

Element class of a commutative differential multi-graded algebra.

```

coboundaries(n, total=False)

```

The n -th coboundary group of the algebra.

This is a vector space over the base field F , and it is returned as a subspace of the vector space F^d , where the n -th homogeneous component has dimension d .

INPUT:

- n – degree
- `total` – boolean (default: `False`); if `True`, return the coboundaries in total degree n

If n is an integer rather than a multi-index, then the total degree is used in that case as well.

EXAMPLES:

```
sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1,0), (0, 1), (0,2)))
sage: B = A.cdg_algebra(differential={a: c})
sage: B.coboundaries((0,2))
Vector space of degree 1 and dimension 1 over Rational Field
Basis matrix:
[1]
sage: B.coboundaries(2)
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[0 1]
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=((Integer(1),Integer(0)),
↳ (Integer(0), Integer(1)), (Integer(0),Integer(2))), names=('a', 'b', 'c'));
↳ (a, b, c) = A._first_ngens(3)
>>> B = A.cdg_algebra(differential={a: c})
>>> B.coboundaries((Integer(0),Integer(2)))
Vector space of degree 1 and dimension 1 over Rational Field
Basis matrix:
[1]
>>> B.coboundaries(Integer(2))
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[0 1]
```

cocycles (n , `total=False`)

The n -th cocycle group of the algebra.

This is a vector space over the base field F , and it is returned as a subspace of the vector space F^d , where the n -th homogeneous component has dimension d .

INPUT:

- n – degree
- `total` – boolean (default: `False`); if `True`, return the cocycles in total degree n

If n is an integer rather than a multi-index, then the total degree is used in that case as well.

EXAMPLES:

```
sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1,0), (0, 1), (0,2)))
sage: B = A.cdg_algebra(differential={a: c})
sage: B.cocycles((0,1))
Vector space of degree 1 and dimension 1 over Rational Field
Basis matrix:
[1]
sage: B.cocycles((0,1), total=True)
Vector space of degree 2 and dimension 1 over Rational Field
```

(continues on next page)

(continued from previous page)

```
Basis matrix:
[0 1]
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=((Integer(1),Integer(0)),
↳ (Integer(0), Integer(1)), (Integer(0),Integer(2))), names=('a', 'b', 'c'));
↳ (a, b, c,) = A._first_ngens(3)
>>> B = A.cdg_algebra(differential={a: c})
>>> B.cocycles((Integer(0),Integer(1)))
Vector space of degree 1 and dimension 1 over Rational Field
Basis matrix:
[1]
>>> B.cocycles((Integer(0),Integer(1)), total=True)
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[0 1]
```

cohomology (*n*, *total=False*)

The n -th cohomology group of the algebra.

This is a vector space over the base ring, defined as the quotient cocycles/coboundaries. The elements of the quotient are lifted to the vector space of cocycles, and this is described in terms of those lifts.

Compare to `cohomology_raw()`.

INPUT:

- *n* – degree
- *total* – boolean (default: False); if True, return the cohomology in total degree n

If n is an integer rather than a multi-index, then the total degree is used in that case as well.

EXAMPLES:

```
sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1,0), (0, 1), (0,2)))
sage: B = A.cdg_algebra(differential={a: c})
sage: B.cohomology((0,2))
Free module generated by {} over Rational Field

sage: B.cohomology(1)
Free module generated by {[b]} over Rational Field
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=((Integer(1),Integer(0)),
↳ (Integer(0), Integer(1)), (Integer(0),Integer(2))), names=('a', 'b', 'c'));
↳ (a, b, c,) = A._first_ngens(3)
>>> B = A.cdg_algebra(differential={a: c})
>>> B.cohomology((Integer(0),Integer(2)))
Free module generated by {} over Rational Field

>>> B.cohomology(Integer(1))
Free module generated by {[b]} over Rational Field
```


`cohomology_raw(n, total=False)`

The n -th cohomology group of the algebra.

This is a vector space over the base ring, and it is returned as the quotient cocycles/coboundaries.

Compare to `cohomology()`.

INPUT:

- `n` – degree
- `total` – boolean (default: `False`); if `True`, return the cohomology in total degree n

If n is an integer rather than a multi-index, then the total degree is used in that case as well.

EXAMPLES:

```
sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1,0), (0, 1), (0,2)))
sage: B = A.cdg_algebra(differential={a: c})
sage: B.cohomology_raw((0,2))
Vector space quotient V/W of dimension 0 over Rational Field where
V: Vector space of degree 1 and dimension 1 over Rational Field
Basis matrix:
[1]
W: Vector space of degree 1 and dimension 1 over Rational Field
Basis matrix:
[1]

sage: B.cohomology_raw(1)
Vector space quotient V/W of dimension 1 over Rational Field where
V: Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[0 1]
W: Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=((Integer(1),Integer(0)),
↳ (Integer(0), Integer(1)), (Integer(0),Integer(2))), names=('a', 'b', 'c',));
↳ (a, b, c) = A._first_ngens(3)
>>> B = A.cdg_algebra(differential={a: c})
>>> B.cohomology_raw(Integer(0),Integer(2))
Vector space quotient V/W of dimension 0 over Rational Field where
V: Vector space of degree 1 and dimension 1 over Rational Field
Basis matrix:
[1]
W: Vector space of degree 1 and dimension 1 over Rational Field
Basis matrix:
[1]

>>> B.cohomology_raw(Integer(1))
Vector space quotient V/W of dimension 1 over Rational Field where
V: Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[0 1]
```

(continues on next page)

(continued from previous page)

```

W: Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]

```

homology (*n*, *total=False*)

The n -th cohomology group of the algebra.

This is a vector space over the base ring, defined as the quotient cocycles/coboundaries. The elements of the quotient are lifted to the vector space of cocycles, and this is described in terms of those lifts.

Compare to `cohomology_raw()`.

INPUT:

- *n* – degree
- *total* – boolean (default: False); if True, return the cohomology in total degree n

If n is an integer rather than a multi-index, then the total degree is used in that case as well.

EXAMPLES:

```

sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1,0), (0, 1), (0,2)))
sage: B = A.cdg_algebra(differential={a: c})
sage: B.cohomology((0,2))
Free module generated by {} over Rational Field

sage: B.cohomology(1)
Free module generated by {[b]} over Rational Field

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=((Integer(1),Integer(0)),
↪ (Integer(0), Integer(1)), (Integer(0),Integer(2))), names=('a', 'b', 'c'));
↪ (a, b, c,) = A._first_ngens(3)
>>> B = A.cdg_algebra(differential={a: c})
>>> B.cohomology((Integer(0),Integer(2)))
Free module generated by {} over Rational Field

>>> B.cohomology(Integer(1))
Free module generated by {[b]} over Rational Field

```

class `sage.algebras.commutative_dga.Differential_multigraded` (*A*, *im_gens*)

Bases: `Differential`

Differential of a commutative multi-graded algebra.

coboundaries (*n*, *total=False*)

The n -th coboundary group of the algebra.

This is a vector space over the base field F , and it is returned as a subspace of the vector space F^d , where the n -th homogeneous component has dimension d .

INPUT:

- *n* – degree
- *total* – boolean (default: False); if True, return the coboundaries in total degree n

If n is an integer rather than a multi-index, then the total degree is used in that case as well.

EXAMPLES:

```
sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1, 0), (0, 1), (0, 2)))
sage: d = A.differential({a: c})
sage: d.coboundaries((0, 2))
Vector space of degree 1 and dimension 1 over Rational Field
Basis matrix:
[1]
sage: d.coboundaries(2)
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[0 1]
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=((Integer(1), Integer(0)), (Integer(0), Integer(1)), (Integer(0), Integer(2))), names=('a', 'b', 'c'),
); (a, b, c,) = A._first_ngens(3)
>>> d = A.differential({a: c})
>>> d.coboundaries((Integer(0), Integer(2)))
Vector space of degree 1 and dimension 1 over Rational Field
Basis matrix:
[1]
>>> d.coboundaries(Integer(2))
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[0 1]
```

cocycles (n , $total=False$)

The n -th cocycle group of the algebra.

This is a vector space over the base field F , and it is returned as a subspace of the vector space F^d , where the n -th homogeneous component has dimension d .

INPUT:

- n – degree
- $total$ – boolean (default: False); if True, return the cocycles in total degree n

If n is an integer rather than a multi-index, then the total degree is used in that case as well.

EXAMPLES:

```
sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1, 0), (0, 1), (0, 2)))
sage: d = A.differential({a: c})
sage: d.cocycles((0, 1))
Vector space of degree 1 and dimension 1 over Rational Field
Basis matrix:
[1]
sage: d.cocycles((0, 1), total=True)
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[0 1]
```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=((Integer(1), Integer(0)),
↳ (Integer(0), Integer(1)), (Integer(0), Integer(2))), names=('a', 'b', 'c',
↳ )); (a, b, c,) = A._first_ngens(3)
>>> d = A.differential({a: c})
>>> d.cocycles((Integer(0), Integer(1)))
Vector space of degree 1 and dimension 1 over Rational Field
Basis matrix:
[1]
>>> d.cocycles((Integer(0), Integer(1)), total=True)
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[0 1]

```

cohomology (*n*, *total=False*)

The n -th cohomology group of the algebra.

This is a vector space over the base ring, defined as the quotient cocycles/coboundaries. The elements of the quotient are lifted to the vector space of cocycles, and this is described in terms of those lifts.

INPUT:

- *n* – degree
- *total* – boolean (default: False); if True, return the cohomology in total degree n

If n is an integer rather than a multi-index, then the total degree is used in that case as well.

See also

[`cohomology\_raw\(\)`](#)

EXAMPLES:

```

sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1, 0), (0, 1), (0,
↳ 2)))
sage: d = A.differential({a: c})
sage: d.cohomology((0, 2))
Free module generated by {} over Rational Field

sage: d.cohomology(1)
Free module generated by {[b]} over Rational Field

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=((Integer(1), Integer(0)),
↳ (Integer(0), Integer(1)), (Integer(0), Integer(2))), names=('a', 'b', 'c',
↳ )); (a, b, c,) = A._first_ngens(3)
>>> d = A.differential({a: c})
>>> d.cohomology((Integer(0), Integer(2)))
Free module generated by {} over Rational Field

>>> d.cohomology(Integer(1))
Free module generated by {[b]} over Rational Field

```

cohomology_raw(*n*, *total=False*)

The *n*-th cohomology group of the algebra.

This is a vector space over the base ring, and it is returned as the quotient cocycles/coboundaries.

INPUT:

- *n* – degree
- *total* – boolean (default: False); if True, return the cohomology in total degree *n*

If *n* is an integer rather than a multi-index, then the total degree is used in that case as well.

See also

[*cohomology\(\)*](#)

EXAMPLES:

```
sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1, 0), (0, 1), (0, 2)))
sage: d = A.differential({a: c})
sage: d.cohomology_raw((0, 2))
Vector space quotient V/W of dimension 0 over Rational Field where
V: Vector space of degree 1 and dimension 1 over Rational Field
Basis matrix:
[1]
W: Vector space of degree 1 and dimension 1 over Rational Field
Basis matrix:
[1]

sage: d.cohomology_raw(1)
Vector space quotient V/W of dimension 1 over Rational Field where
V: Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[0 1]
W: Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=((Integer(1), Integer(0)),
↳ (Integer(0), Integer(1)), (Integer(0), Integer(2))), names=('a', 'b', 'c',
↳ )); (a, b, c,) = A._first_ngens(3)
>>> d = A.differential({a: c})
>>> d.cohomology_raw((Integer(0), Integer(2)))
Vector space quotient V/W of dimension 0 over Rational Field where
V: Vector space of degree 1 and dimension 1 over Rational Field
Basis matrix:
[1]
W: Vector space of degree 1 and dimension 1 over Rational Field
Basis matrix:
[1]

>>> d.cohomology_raw(Integer(1))
```

(continues on next page)

(continued from previous page)

```

Vector space quotient V/W of dimension 1 over Rational Field where
V: Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[0 1]
W: Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]

```

differential_matrix_multigraded (*n*, *total=False*)

The matrix that gives the differential in degree *n*.

Todo

Rename this to `differential_matrix` once inheritance, overriding, and cached methods work together better. See [Issue #17201](#).

INPUT:

- *n* – degree
- *total* – boolean (default: `False`); if `True`, return the matrix corresponding to total degree *n*

If *n* is an integer rather than a multi-index, then the total degree is used in that case as well.

EXAMPLES:

```

sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1, 0), (0, 1), (0, 2)))
sage: d = A.differential({a: c})
sage: d.differential_matrix_multigraded((1, 0))
[1]
sage: d.differential_matrix_multigraded(1, total=True)
[0 1]
[0 0]
sage: d.differential_matrix_multigraded((1, 0), total=True)
[0 1]
[0 0]
sage: d.differential_matrix_multigraded(1)
[0 1]
[0 0]

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=((Integer(1), Integer(0)), (Integer(0), Integer(1)), (Integer(0), Integer(2))), names=('a', 'b', 'c',
↪)); (a, b, c,) = A._first_ngens(3)
>>> d = A.differential({a: c})
>>> d.differential_matrix_multigraded((Integer(1), Integer(0)))
[1]
>>> d.differential_matrix_multigraded(Integer(1), total=True)
[0 1]
[0 0]
>>> d.differential_matrix_multigraded((Integer(1), Integer(0)), total=True)
[0 1]

```

(continues on next page)

(continued from previous page)

```
[0 0]
>>> d.differential_matrix_multigraded(Integer(1))
[0 1]
[0 0]
```

homology (*n*, *total=False*)

The n -th cohomology group of the algebra.

This is a vector space over the base ring, defined as the quotient cocycles/coboundaries. The elements of the quotient are lifted to the vector space of cocycles, and this is described in terms of those lifts.

INPUT:

- *n* – degree
- *total* – boolean (default: False); if True, return the cohomology in total degree n

If n is an integer rather than a multi-index, then the total degree is used in that case as well.

See also

[`cohomology\_raw\(\)`](#)

EXAMPLES:

```
sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1, 0), (0, 1), (0, 2)))
sage: d = A.differential({a: c})
sage: d.cohomology((0, 2))
Free module generated by {} over Rational Field

sage: d.cohomology(1)
Free module generated by {[b]} over Rational Field
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=((Integer(1), Integer(0)), (Integer(0), Integer(1)), (Integer(0), Integer(2))), names=('a', 'b', 'c',
↪)); (a, b, c,) = A._first_ngens(3)
>>> d = A.differential({a: c})
>>> d.cohomology((Integer(0), Integer(2)))
Free module generated by {} over Rational Field

>>> d.cohomology(Integer(1))
Free module generated by {[b]} over Rational Field
```

class `sage.algebras.commutative_dga.GCAAlgebra` (*base*, *R=None*, *I=None*, *names=None*, *degrees=None*, *category=None*)

Bases: `UniqueRepresentation`, `QuotientRing_nc`

A graded commutative algebra.

INPUT:

- *base* – the base field

- `names` – (optional) names of the generators: a list of strings or a single string with the names separated by commas. If not specified, the generators are named “`x0`”, “`x1`”, ...
- `degrees` – (optional) a tuple or list specifying the degrees of the generators; if omitted, each generator is given degree 1, and if both `names` and `degrees` are omitted, an error is raised.
- `R` – (default: `None`) the ring over which the algebra is defined: if this is specified, the algebra is defined to be R/I .
- `I` – (default: `None`) an ideal in R . It should include, among other relations, the squares of the generators of odd degree

As described in the module-level documentation, these are graded algebras for which oddly graded elements anti-commute and evenly graded elements commute.

The arguments `R` and `I` are primarily for use by the `quotient()` method.

These algebras should be graded over the integers; multi-graded algebras should be constructed using `GCAlgebra_multigraded` instead.

EXAMPLES:

```
sage: A.<a,b> = GradedCommutativeAlgebra(QQ, degrees = (2, 3))
sage: a.degree()
2
sage: B = A.quotient(A.ideal(a**2*b))
sage: B
Graded Commutative Algebra with generators ('a', 'b') in degrees (2, 3)
with relations [a^2*b] over Rational Field
sage: A.basis(7)
[a^2*b]
sage: B.basis(7)
[]
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees = (Integer(2), Integer(3)), names=('a', 'b')); (a, b) = A._first_ngens(2)
>>> a.degree()
2
>>> B = A.quotient(A.ideal(a**Integer(2)*b))
>>> B
Graded Commutative Algebra with generators ('a', 'b') in degrees (2, 3)
with relations [a^2*b] over Rational Field
>>> A.basis(Integer(7))
[a^2*b]
>>> B.basis(Integer(7))
[]
```

Note that the function `GradedCommutativeAlgebra()` can also be used to construct these algebras.

class `Element` (`A`, `rep`)

Bases: `QuotientRingElement`

An element of a graded commutative algebra.

basis_coefficients (`total=False`)

Return the coefficients of this homogeneous element with respect to the basis in its degree.

For example, if this is the sum of the 0-th and 2-nd basis elements, return the list `[1, 0, 1]`.

Raise an error if the element is not homogeneous.

INPUT:

- `total` – boolean (default: `False`); this is only used in the multi-graded case, in which case if `True`, it returns the coefficients with respect to the basis for the total degree of this element

OUTPUT: list of elements of the base field

EXAMPLES:

```
sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(1, 2, 2, 3))
sage: A.basis(3)
[x*y, x*z, t]
sage: (t + 3*x*y).basis_coefficients()
[3, 0, 1]
sage: (t + x).basis_coefficients()
Traceback (most recent call last):
...
ValueError: this element is not homogeneous

sage: B.<c,d> = GradedCommutativeAlgebra(QQ, degrees=((2,0), (0,4)))
sage: B.basis(4)
[c^2, d]
sage: (c^2 - 1/2 * d).basis_coefficients(total=True)
[1, -1/2]
sage: (c^2 - 1/2 * d).basis_coefficients()
Traceback (most recent call last):
...
ValueError: this element is not homogeneous
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1), Integer(2),
↳ Integer(2), Integer(3)), names=('x', 'y', 'z', 't')); (x, y, z, t) = A._first_
↳ A._first_ngens(4)
>>> A.basis(Integer(3))
[x*y, x*z, t]
>>> (t + Integer(3)*x*y).basis_coefficients()
[3, 0, 1]
>>> (t + x).basis_coefficients()
Traceback (most recent call last):
...
ValueError: this element is not homogeneous

>>> B = GradedCommutativeAlgebra(QQ, degrees=((Integer(2), Integer(0)),
↳ (Integer(0), Integer(4))), names=('c', 'd')); (c, d) = B._first_
↳ B._first_ngens(2)
>>> B.basis(Integer(4))
[c^2, d]
>>> (c**Integer(2) - Integer(1)/Integer(2) * d).basis_
↳ coefficients(total=True)
[1, -1/2]
>>> (c**Integer(2) - Integer(1)/Integer(2) * d).basis_coefficients()
Traceback (most recent call last):
...
ValueError: this element is not homogeneous
```

degree (*total=False*)

The degree of this element.

If the element is not homogeneous, this returns the maximum of the degrees of its monomials.

INPUT:

- *total* – ignored, present for compatibility with the multi-graded case

EXAMPLES:

```
sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(1, 2, 3, 3))
sage: e1 = z*t+2*x*y-y^2*z
sage: e1.degree()
7
sage: e1.monomials()
[y^2*z, z*t, x*y]
sage: [i.degree() for i in e1.monomials()]
[7, 6, 3]

sage: A(0).degree()
Traceback (most recent call last):
...
ValueError: the zero element does not have a well-defined degree
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1), Integer(2),
↳Integer(3), Integer(3)), names=('x', 'y', 'z', 't')); (x, y, z, t) =
↳A._first_ngens(4)
>>> e1 = z*t+Integer(2)*x*y-y**Integer(2)*z
>>> e1.degree()
7
>>> e1.monomials()
[y^2*z, z*t, x*y]
>>> [i.degree() for i in e1.monomials()]
[7, 6, 3]

>>> A(Integer(0)).degree()
Traceback (most recent call last):
...
ValueError: the zero element does not have a well-defined degree
```

dict (*copy=True*)

A dictionary that determines the element.

The keys of this dictionary are the tuples of exponents of each monomial, and the values are the corresponding coefficients.

EXAMPLES:

```
sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(1, 2, 2, 3))
sage: elt = x*y - 5*y*z + 7*x*y^2*z^3*t
sage: sorted(elt.monomial_coefficients().items())
[((0, 1, 1, 0), -5), ((1, 1, 0, 0), 1), ((1, 2, 3, 1), 7)]
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1), Integer(2),
```

(continues on next page)

(continued from previous page)

```

↪Integer(2), Integer(3)), names=('x', 'y', 'z', 't')); (x, y, z, t) =
↪A._first_ngens(4)
>>> elt = x*y - Integer(5)*y*z +
↪Integer(7)*x*y**Integer(2)*z**Integer(3)*t
>>> sorted(elt.monomial_coefficients().items())
[((0, 1, 1, 0), -5), ((1, 1, 0, 0), 1), ((1, 2, 3, 1), 7)]

```

dict is an alias:

```

sage: sorted(elt.dict().items())
[((0, 1, 1, 0), -5), ((1, 1, 0, 0), 1), ((1, 2, 3, 1), 7)]

```

```

>>> from sage.all import *
>>> sorted(elt.dict().items())
[((0, 1, 1, 0), -5), ((1, 1, 0, 0), 1), ((1, 2, 3, 1), 7)]

```

homogeneous_parts()

Return the homogeneous parts of the element. The result is given as a dictionary indexed by degree.

EXAMPLES:

```

sage: A.<e1,e2,e3,e4,e5> = GradedCommutativeAlgebra(QQ)
sage: a = e1*e3*e5-3*e2*e3*e5 + e1*e2 -2*e3 + e5
sage: a.homogeneous_parts()
{1: -2*e3 + e5, 2: e1*e2, 3: e1*e3*e5 - 3*e2*e3*e5}

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, names=('e1', 'e2', 'e3', 'e4', 'e5',
↪)); (e1, e2, e3, e4, e5,) = A._first_ngens(5)
>>> a = e1*e3*e5-Integer(3)*e2*e3*e5 + e1*e2 -Integer(2)*e3 + e5
>>> a.homogeneous_parts()
{1: -2*e3 + e5, 2: e1*e2, 3: e1*e3*e5 - 3*e2*e3*e5}

```

is_homogeneous (total=False)

Return True if self is homogeneous and False otherwise.

INPUT:

- total – boolean (default: False); only used in the multi-graded case, in which case if True, check to see if self is homogeneous with respect to total degree

EXAMPLES:

```

sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(1, 2, 3, 3))
sage: e1 = z*t + 2*x*y - y^2*z
sage: e1.degree()
7
sage: e1.monomials()
[y^2*z, z*t, x*y]
sage: [i.degree() for i in e1.monomials()]
[7, 6, 3]
sage: e1.is_homogeneous()
False
sage: em = y^3 - 5*z*t + 3/2*x*y*t
sage: em.is_homogeneous()

```

(continues on next page)

(continued from previous page)

```
True
sage: em.monomials()
[y^3, x*y*t, z*t]
sage: [i.degree() for i in em.monomials()]
[6, 6, 6]
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1), Integer(2),
↳ Integer(3), Integer(3)), names=('x', 'y', 'z', 't')); (x, y, z, t) =
↳ A._first_ngens(4)
>>> e1 = z*t + Integer(2)*x*y - y**Integer(2)*z
>>> e1.degree()
7
>>> e1.monomials()
[y^2*z, z*t, x*y]
>>> [i.degree() for i in e1.monomials()]
[7, 6, 3]
>>> e1.is_homogeneous()
False
>>> em = y**Integer(3) - Integer(5)*z*t + Integer(3)/Integer(2)*x*y*t
>>> em.is_homogeneous()
True
>>> em.monomials()
[y^3, x*y*t, z*t]
>>> [i.degree() for i in em.monomials()]
[6, 6, 6]
```

The element 0 is homogeneous, even though it doesn't have a well-defined degree:

```
sage: A(0).is_homogeneous()
True
```

```
>>> from sage.all import *
>>> A(Integer(0)).is_homogeneous()
True
```

A multi-graded example:

```
sage: B.<c,d> = GradedCommutativeAlgebra(QQ, degrees=((2, 0), (0, 4)))
sage: (c^2 - 1/2 * d).is_homogeneous()
False
sage: (c^2 - 1/2 * d).is_homogeneous(total=True)
True
```

```
>>> from sage.all import *
>>> B = GradedCommutativeAlgebra(QQ, degrees=((Integer(2), Integer(0)),
↳ (Integer(0), Integer(4))), names=('c', 'd')); (c, d) = B._first_
↳ ngens(2)
>>> (c**Integer(2) - Integer(1)/Integer(2) * d).is_homogeneous()
False
>>> (c**Integer(2) - Integer(1)/Integer(2) * d).is_homogeneous(total=True)
True
```

monomial_coefficients (*copy=True*)

A dictionary that determines the element.

The keys of this dictionary are the tuples of exponents of each monomial, and the values are the corresponding coefficients.

EXAMPLES:

```
sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(1, 2, 2, 3))
sage: elt = x*y - 5*y*z + 7*x*y^2*z^3*t
sage: sorted(elt.monomial_coefficients().items())
[((0, 1, 1, 0), -5), ((1, 1, 0, 0), 1), ((1, 2, 3, 1), 7)]
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1), Integer(2),
↳Integer(2), Integer(3)), names=('x', 'y', 'z', 't')); (x, y, z, t) = A._
↳A._first_ngens(4)
>>> elt = x*y - Integer(5)*y*z +
↳Integer(7)*x*y**Integer(2)*z**Integer(3)*t
>>> sorted(elt.monomial_coefficients().items())
[((0, 1, 1, 0), -5), ((1, 1, 0, 0), 1), ((1, 2, 3, 1), 7)]
```

dict is an alias:

```
sage: sorted(elt.dict().items())
[((0, 1, 1, 0), -5), ((1, 1, 0, 0), 1), ((1, 2, 3, 1), 7)]
```

```
>>> from sage.all import *
>>> sorted(elt.dict().items())
[((0, 1, 1, 0), -5), ((1, 1, 0, 0), 1), ((1, 2, 3, 1), 7)]
```

basis (*n*)

Return a basis of the *n*-th homogeneous component of *self*.

EXAMPLES:

```
sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(1, 2, 2, 3))
sage: A.basis(2)
[y, z]
sage: A.basis(3)
[x*y, x*z, t]
sage: A.basis(4)
[y^2, y*z, z^2, x*t]
sage: A.basis(5)
[x*y^2, x*y*z, x*z^2, y*t, z*t]
sage: A.basis(6)
[y^3, y^2*z, y*z^2, z^3, x*y*t, x*z*t]
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1), Integer(2),
↳Integer(2), Integer(3)), names=('x', 'y', 'z', 't')); (x, y, z, t) = A._
↳first_ngens(4)
>>> A.basis(Integer(2))
[y, z]
```

(continues on next page)

(continued from previous page)

```

>>> A.basis(Integer(3))
[x*y, x*z, t]
>>> A.basis(Integer(4))
[y^2, y*z, z^2, x*t]
>>> A.basis(Integer(5))
[x*y^2, x*y*z, x*z^2, y*t, z*t]
>>> A.basis(Integer(6))
[y^3, y^2*z, y*z^2, z^3, x*y*t, x*z*t]

```

cdg_algebra (*differential*)

Construct a differential graded commutative algebra from `self` by specifying a differential.

INPUT:

- `differential` – dictionary defining a differential or a map defining a valid differential

The keys of the dictionary are generators of the algebra, and the associated values are their targets under the differential. Any generators which are not specified are assumed to have zero differential. Alternatively, the differential can be defined using the `differential()` method; see below for an example.

See also

`differential()`

EXAMPLES:

```

sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=(1, 1, 1))
sage: B = A.cdg_algebra({a: b*c, b: a*c})
sage: B
Commutative Differential Graded Algebra with generators ('a', 'b', 'c')
in degrees (1, 1, 1) over Rational Field with differential:
  a --> b*c
  b --> a*c
  c --> 0

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1), Integer(1),
↳ Integer(1)), names=('a', 'b', 'c')); (a, b, c) = A._first_ngens(3)
>>> B = A.cdg_algebra({a: b*c, b: a*c})
>>> B
Commutative Differential Graded Algebra with generators ('a', 'b', 'c')
in degrees (1, 1, 1) over Rational Field with differential:
  a --> b*c
  b --> a*c
  c --> 0

```

Note that differential can also be a map:

```

sage: d = A.differential({a: b*c, b: a*c})
sage: d
Differential of Graded Commutative Algebra with generators ('a', 'b', 'c')
in degrees (1, 1, 1) over Rational Field
Defn: a --> b*c

```

(continues on next page)

(continued from previous page)

```

        b --> a*c
        c --> 0
sage: A.cdg_algebra(d) is B
True

```

```

>>> from sage.all import *
>>> d = A.differential({a: b*c, b: a*c})
>>> d
Differential of Graded Commutative Algebra with generators ('a', 'b', 'c')
in degrees (1, 1, 1) over Rational Field
Defn: a --> b*c
      b --> a*c
      c --> 0
>>> A.cdg_algebra(d) is B
True

```

differential (*diff*)

Construct a differential on self.

INPUT:

- *diff* – dictionary defining a differential

The keys of the dictionary are generators of the algebra, and the associated values are their targets under the differential. Any generators which are not specified are assumed to have zero differential.

EXAMPLES:

```

sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(1, 1, 2))
sage: A.differential({y:x*y, x: x*y})
Differential of Graded Commutative Algebra with generators ('x', 'y', 'z')
in degrees (1, 1, 2) over Rational Field
Defn: x --> x*y
      y --> x*y
      z --> 0
sage: B.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=(1, 2, 2))
sage: d = B.differential({b:a*c, c:a*c})
sage: d(b*c)
a*b*c + a*c^2

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1), Integer(1),
↳ Integer(2)), names=('x', 'y', 'z',)); (x, y, z,) = A._first_ngens(3)
>>> A.differential({y:x*y, x: x*y})
Differential of Graded Commutative Algebra with generators ('x', 'y', 'z')
in degrees (1, 1, 2) over Rational Field
Defn: x --> x*y
      y --> x*y
      z --> 0
>>> B = GradedCommutativeAlgebra(QQ, degrees=(Integer(1), Integer(2),
↳ Integer(2)), names=('a', 'b', 'c',)); (a, b, c,) = B._first_ngens(3)
>>> d = B.differential({b:a*c, c:a*c})
>>> d(b*c)
a*b*c + a*c^2

```

quotient (*I*, *check=True*)

Create the quotient of this algebra by a two-sided ideal *I*.

INPUT:

- *I* – a two-sided homogeneous ideal of this algebra
- *check* – boolean (default: `True`); if `True`, check whether *I* is generated by homogeneous elements

EXAMPLES:

```
sage: A.<x,y,z,t> = GradedCommutativeAlgebra(GF(5), degrees=(2, 2, 3, 4))
sage: I = A.ideal([x*t+z^2, x*y - t])
sage: B = A.quotient(I); B
Graded Commutative Algebra with generators ('x', 'y', 'z', 't')
in degrees (2, 2, 3, 4) with relations [x*t, x*y - t]
over Finite Field of size 5
sage: B(x*t)
0
sage: B(x*y)
t
sage: A.basis(7)
[x^2*z, x*y*z, y^2*z, z*t]
sage: B.basis(7)
[x^2*z, y^2*z, z*t]
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(GF(Integer(5)), degrees=(Integer(2), Integer(2), Integer(3), Integer(4)), names=('x', 'y', 'z', 't')); (x, y, z, t) = A._first_ngens(4)
>>> I = A.ideal([x*t+z**Integer(2), x*y - t])
>>> B = A.quotient(I); B
Graded Commutative Algebra with generators ('x', 'y', 'z', 't')
in degrees (2, 2, 3, 4) with relations [x*t, x*y - t]
over Finite Field of size 5
>>> B(x*t)
0
>>> B(x*y)
t
>>> A.basis(Integer(7))
[x^2*z, x*y*z, y^2*z, z*t]
>>> B.basis(Integer(7))
[x^2*z, y^2*z, z*t]
```

class sage.algebras.commutative_dga.GCAAlgebraHomset (*R*, *S*, *category=None*)

Bases: `RingHomset_generic`

Set of morphisms between two graded commutative algebras.

Note

Homsets (and thus morphisms) have only been implemented when the base fields are the same for the domain and codomain.

EXAMPLES:


```

sage: A.<x,y> = GradedCommutativeAlgebra(QQ, degrees=(1,2))
sage: H = Hom(A,A)
sage: H([x,y]) == H.identity()
True
sage: H([x,x]) == H.identity()
False

sage: A.<w,x> = GradedCommutativeAlgebra(QQ, degrees=(1,2))
sage: B.<y,z> = GradedCommutativeAlgebra(QQ, degrees=(1,1))
sage: H = Hom(A,B)
sage: H([y,0])
Graded Commutative Algebra morphism:
  From: Graded Commutative Algebra with generators ('w', 'x')
         in degrees (1, 2) over Rational Field
  To:   Graded Commutative Algebra with generators ('y', 'z')
         in degrees (1, 1) over Rational Field
  Defn: (w, x) --> (y, 0)
sage: H([y,y*z])
Graded Commutative Algebra morphism:
  From: Graded Commutative Algebra with generators ('w', 'x')
         in degrees (1, 2) over Rational Field
  To:   Graded Commutative Algebra with generators ('y', 'z')
         in degrees (1, 1) over Rational Field
  Defn: (w, x) --> (y, y*z)

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1),Integer(2)), names=('x',
↪ 'y')); (x, y) = A._first_ngens(2)
>>> H = Hom(A,A)
>>> H([x,y]) == H.identity()
True
>>> H([x,x]) == H.identity()
False

>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1),Integer(2)), names=('w',
↪ 'x')); (w, x) = A._first_ngens(2)
>>> B = GradedCommutativeAlgebra(QQ, degrees=(Integer(1),Integer(1)), names=('y',
↪ 'z')); (y, z) = B._first_ngens(2)
>>> H = Hom(A,B)
>>> H([y,Integer(0)])
Graded Commutative Algebra morphism:
  From: Graded Commutative Algebra with generators ('w', 'x')
         in degrees (1, 2) over Rational Field
  To:   Graded Commutative Algebra with generators ('y', 'z')
         in degrees (1, 1) over Rational Field
  Defn: (w, x) --> (y, 0)
>>> H([y,y*z])
Graded Commutative Algebra morphism:
  From: Graded Commutative Algebra with generators ('w', 'x')
         in degrees (1, 2) over Rational Field
  To:   Graded Commutative Algebra with generators ('y', 'z')
         in degrees (1, 1) over Rational Field
  Defn: (w, x) --> (y, y*z)

```

identity()

Construct the identity morphism of this homset.

EXAMPLES:

```
sage: A.<x,y> = GradedCommutativeAlgebra(QQ, degrees=(1,2))
sage: H = Hom(A,A)
sage: H([x,y]) == H.identity()
True
sage: H([x,x]) == H.identity()
False
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1),Integer(2)), names=(
↳ 'x', 'y',)); (x, y,) = A._first_ngens(2)
>>> H = Hom(A,A)
>>> H([x,y]) == H.identity()
True
>>> H([x,x]) == H.identity()
False
```

zero()

Construct the “zero” morphism of this homset: the map sending each generator to zero.

EXAMPLES:

```
sage: A.<x,y> = GradedCommutativeAlgebra(QQ, degrees=(1,2))
sage: B.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=(1,1,1))
sage: zero = Hom(A,B).zero()
sage: zero(x) == zero(y) == 0
True
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1),Integer(2)), names=(
↳ 'x', 'y',)); (x, y,) = A._first_ngens(2)
>>> B = GradedCommutativeAlgebra(QQ, degrees=(Integer(1),Integer(1),
↳ Integer(1)), names=('a', 'b', 'c',)); (a, b, c,) = B._first_ngens(3)
>>> zero = Hom(A,B).zero()
>>> zero(x) == zero(y) == Integer(0)
True
```

```
class sage.algebras.commutative_dga.GCAlgebraMorphism (parent, im_gens, check=True)
```

Bases: `RingHomomorphism_im_gens`

Create a morphism between two *graded commutative algebras*.

INPUT:

- `parent` – the parent homset
- `im_gens` – the images, in the codomain, of the generators of the domain
- `check` – boolean (default: `True`); check whether the proposed map is actually an algebra map; if the domain and codomain have differentials, also check that the map respects those.

EXAMPLES:

```

sage: A.<x,y> = GradedCommutativeAlgebra(QQ)
sage: H = Hom(A,A)
sage: f = H([y,x])
sage: f
Graded Commutative Algebra endomorphism of Graded Commutative Algebra
with generators ('x', 'y') in degrees (1, 1) over Rational Field
Defn: (x, y) --> (y, x)
sage: f(x*y)
-x*y

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, names=('x', 'y')); (x, y) = A._first_
↳ngens(2)
>>> H = Hom(A,A)
>>> f = H([y,x])
>>> f
Graded Commutative Algebra endomorphism of Graded Commutative Algebra
with generators ('x', 'y') in degrees (1, 1) over Rational Field
Defn: (x, y) --> (y, x)
>>> f(x*y)
-x*y

```

is_graded(total=False)

Return True if this morphism is graded.

That is, return True if $f(x)$ is zero, or if $f(x)$ is homogeneous and has the same degree as x , for each generator x .

INPUT:

- total – boolean (default: False); if True, use the total degree to determine whether the morphism is graded (relevant only in the multigraded case)

EXAMPLES:

```

sage: C.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=(1,1,2))
sage: H = Hom(C,C)
sage: H([a, b, a*b + 2*a]).is_graded()
False
sage: H([a, b, a*b]).is_graded()
True

sage: A.<w,x> = GradedCommutativeAlgebra(QQ, degrees=((1,0), (1,0)))
sage: B.<y,z> = GradedCommutativeAlgebra(QQ, degrees=((1,0), (0,1)))
sage: H = Hom(A,B)
sage: H([y,0]).is_graded()
True
sage: H([z,z]).is_graded()
False
sage: H([z,z]).is_graded(total=True)
True

```

```

>>> from sage.all import *
>>> C = GradedCommutativeAlgebra(QQ, degrees=(Integer(1), Integer(1),

```

(continues on next page)

(continued from previous page)

```

↪Integer(2)), names=('a', 'b', 'c',)); (a, b, c,) = C._first_ngens(3)
>>> H = Hom(C,C)
>>> H([a, b, a*b + Integer(2)*a]).is_graded()
False
>>> H([a, b, a*b]).is_graded()
True

>>> A = GradedCommutativeAlgebra(QQ, degrees=((Integer(1),Integer(0)),↪
↪(Integer(1),Integer(0))), names=('w', 'x',)); (w, x,) = A._first_ngens(2)
>>> B = GradedCommutativeAlgebra(QQ, degrees=((Integer(1),Integer(0)),↪
↪(Integer(0),Integer(1))), names=('y', 'z',)); (y, z,) = B._first_ngens(2)
>>> H = Hom(A,B)
>>> H([y,Integer(0)]).is_graded()
True
>>> H([z,z]).is_graded()
False
>>> H([z,z]).is_graded(total=True)
True

```

```

class sage.algebras.commutative_dga.GCAlgebra_multigraded(base, degrees, names=None, R=None,
                                                             I=None, category=None)

```

Bases: *GCAlgebra*

A multi-graded commutative algebra.

INPUT:

- *base* – the base field
- *degrees* – tuple or list specifying the degrees of the generators
- *names* – (optional) names of the generators: a list of strings or a single string with the names separated by commas; if not specified, the generators are named $x_0, x_1, \dots$
- *R* – (optional) the ring over which the algebra is defined
- *I* – (optional) an ideal in *R*; it should include, among other relations, the squares of the generators of odd degree

When defining such an algebra, each entry of *degrees* should be a list, tuple, or element of an additive (free) abelian group. Regardless of how the user specifies the degrees, Sage converts them to group elements.

The arguments *R* and *I* are primarily for use by the *GCAlgebra.quotient()* method.

EXAMPLES:

```

sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1,0), (0,1), (1,1)))
sage: A
Graded Commutative Algebra with generators ('a', 'b', 'c')
in degrees ((1, 0), (0, 1), (1, 1)) over Rational Field
sage: a**2
0
sage: c.degree(total=True)
2
sage: c**2
c^2

```

(continues on next page)

(continued from previous page)

```
sage: c.degree()
(1, 1)
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=((Integer(1),Integer(0)),
↳ (Integer(0),Integer(1)), (Integer(1),Integer(1))), names=('a', 'b', 'c')); (a,
↳ b, c,) = A._first_ngens(3)
>>> A
Graded Commutative Algebra with generators ('a', 'b', 'c')
in degrees ((1, 0), (0, 1), (1, 1)) over Rational Field
>>> a**Integer(2)
0
>>> c.degree(total=True)
2
>>> c**Integer(2)
c^2
>>> c.degree()
(1, 1)
```

Although the degree of `c` was defined using a Python tuple, it is returned as an element of an additive abelian group, and so it can be manipulated via arithmetic operations:

```
sage: type(c.degree())
<class 'sage.groups.additive_abelian.additive_abelian_group.AdditiveAbelianGroup_
↳ fixed_gens_with_category.element_class'>
sage: 2 * c.degree()
(2, 2)
sage: (a*b).degree() == a.degree() + b.degree()
True
```

```
>>> from sage.all import *
>>> type(c.degree())
<class 'sage.groups.additive_abelian.additive_abelian_group.AdditiveAbelianGroup_
↳ fixed_gens_with_category.element_class'>
>>> Integer(2) * c.degree()
(2, 2)
>>> (a*b).degree() == a.degree() + b.degree()
True
```

The `basis()` method and the `Element.degree()` method both accept the boolean keyword `total`. If `True`, use the total degree:

```
sage: A.basis(2, total=True)
[a*b, c]
sage: c.degree(total=True)
2
```

```
>>> from sage.all import *
>>> A.basis(Integer(2), total=True)
[a*b, c]
>>> c.degree(total=True)
2
```

class `Element` (*A*, *rep*)

Bases: `Element`

degree (*total=False*)

Return the degree of this element.

INPUT:

- *total* – if `True`, return the total degree, an integer; otherwise, return the degree as an element of an additive free abelian group

If not requesting the total degree, raise an error if the element is not homogeneous.

EXAMPLES:

```
sage: A.<a,b,c> = GradedCommutativeAlgebra(GF(2),
....:                                     degrees=((1,0), (0,1), (1,1)))
sage: (a**2*b).degree()
(2, 1)
sage: (a**2*b).degree(total=True)
3
sage: (a**2*b + c).degree()
Traceback (most recent call last):
...
ValueError: this element is not homogeneous
sage: (a**2*b + c).degree(total=True)
3
sage: A(0).degree()
Traceback (most recent call last):
...
ValueError: the zero element does not have a well-defined degree
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(GF(Integer(2)),
...                               degrees=((Integer(1), Integer(0)),
...   ↪ (Integer(0), Integer(1)), (Integer(1), Integer(1))), names=('a', 'b', 'c'
...   ↪ ',)); (a, b, c,) = A._first_ngens(3)
>>> (a**Integer(2)*b).degree()
(2, 1)
>>> (a**Integer(2)*b).degree(total=True)
3
>>> (a**Integer(2)*b + c).degree()
Traceback (most recent call last):
...
ValueError: this element is not homogeneous
>>> (a**Integer(2)*b + c).degree(total=True)
3
>>> A(Integer(0)).degree()
Traceback (most recent call last):
...
ValueError: the zero element does not have a well-defined degree
```

basis (*n*, *total=False*)

Basis in degree *n*.

- *n* – degree or integer
- *total* – boolean (default: `False`); if `True`, return the basis in total degree *n*

If n is an integer rather than a multi-index, then the total degree is used in that case as well.

EXAMPLES:

```
sage: A.<a,b,c> = GradedCommutativeAlgebra(GF(2), degrees=((1,0), (0,1), (1,
↪1)))
sage: A.basis((1,1))
[a*b, c]
sage: A.basis(2, total=True)
[a^2, a*b, b^2, c]
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(GF(Integer(2)), degrees=((Integer(1),
↪Integer(0)), (Integer(0),Integer(1)), (Integer(1),Integer(1))), names=('a',
↪'b', 'c',)); (a, b, c,) = A._first_ngens(3)
>>> A.basis((Integer(1),Integer(1)))
[a*b, c]
>>> A.basis(Integer(2), total=True)
[a^2, a*b, b^2, c]
```

Since 2 is not a multi-index, we don't need to specify that `total` is `True`:

```
sage: A.basis(2)
[a^2, a*b, b^2, c]
```

```
>>> from sage.all import *
>>> A.basis(Integer(2))
[a^2, a*b, b^2, c]
```

If `total` is `True`, then n can still be a tuple, list, etc., and its total degree is used instead:

```
sage: A.basis((1,1), total=True)
[a^2, a*b, b^2, c]
```

```
>>> from sage.all import *
>>> A.basis((Integer(1),Integer(1)), total=True)
[a^2, a*b, b^2, c]
```

cdg_algebra(*differential*)

Construct a differential graded commutative algebra from `self` by specifying a differential.

INPUT:

- `differential` – dictionary defining a differential or a map defining a valid differential

The keys of the dictionary are generators of the algebra, and the associated values are their targets under the differential. Any generators which are not specified are assumed to have zero differential. Alternatively, the differential can be defined using the `differential()` method; see below for an example.

See also

`differential()`

EXAMPLES:

```

sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1,0), (0, 1), (0,2)))
sage: A.cdg_algebra({a: c})
Commutative Differential Graded Algebra with generators ('a', 'b', 'c')
in degrees ((1, 0), (0, 1), (0, 2)) over Rational Field with differential:
a --> c
b --> 0
c --> 0
sage: d = A.differential({a: c})
sage: A.cdg_algebra(d)
Commutative Differential Graded Algebra with generators ('a', 'b', 'c')
in degrees ((1, 0), (0, 1), (0, 2)) over Rational Field with differential:
a --> c
b --> 0
c --> 0

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=((Integer(1),Integer(0)),
↪ (Integer(0), Integer(1)), (Integer(0),Integer(2))), names=('a', 'b', 'c',));
↪ (a, b, c,) = A._first_ngens(3)
>>> A.cdg_algebra({a: c})
Commutative Differential Graded Algebra with generators ('a', 'b', 'c')
in degrees ((1, 0), (0, 1), (0, 2)) over Rational Field with differential:
a --> c
b --> 0
c --> 0
>>> d = A.differential({a: c})
>>> A.cdg_algebra(d)
Commutative Differential Graded Algebra with generators ('a', 'b', 'c')
in degrees ((1, 0), (0, 1), (0, 2)) over Rational Field with differential:
a --> c
b --> 0
c --> 0

```

differential (*diff*)

Construct a differential on self.

INPUT:

- *diff* – dictionary defining a differential

The keys of the dictionary are generators of the algebra, and the associated values are their targets under the differential. Any generators which are not specified are assumed to have zero differential.

EXAMPLES:

```

sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1,0), (0, 1), (0,2)))
sage: A.differential({a: c})
Differential of Graded Commutative Algebra with generators ('a', 'b', 'c')
in degrees ((1, 0), (0, 1), (0, 2)) over Rational Field
Defn: a --> c
      b --> 0
      c --> 0

```



```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=((Integer(1), Integer(0)),
↳ (Integer(0), Integer(1)), (Integer(0), Integer(2))), names=('a', 'b', 'c'));
↳ (a, b, c) = A._first_ngens(3)
>>> A.differential({a: c})
Differential of Graded Commutative Algebra with generators ('a', 'b', 'c')
in degrees ((1, 0), (0, 1), (0, 2)) over Rational Field
Defn: a --> c
      b --> 0
      c --> 0

```

quotient (*I*, *check=True*)

Create the quotient of this algebra by a two-sided ideal *I*.

INPUT:

- *I* – a two-sided homogeneous ideal of this algebra
- *check* – boolean (default: True); if True, check whether *I* is generated by homogeneous elements

EXAMPLES:

```

sage: A.<x,y,z,t> = GradedCommutativeAlgebra(GF(5), degrees=(2, 2, 3, 4))
sage: I = A.ideal([x*t+z^2, x*y - t])
sage: B = A.quotient(I)
sage: B
Graded Commutative Algebra with generators ('x', 'y', 'z', 't')
in degrees (2, 2, 3, 4) with relations [x*t, x*y - t]
over Finite Field of size 5
sage: B(x*t)
0
sage: B(x*y)
t
sage: A.basis(7)
[x^2*z, x*y*z, y^2*z, z*t]
sage: B.basis(7)
[x^2*z, y^2*z, z*t]

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(GF(Integer(5)), degrees=(Integer(2),
↳ Integer(2), Integer(3), Integer(4)), names=('x', 'y', 'z', 't')); (x, y, z,
↳ t,) = A._first_ngens(4)
>>> I = A.ideal([x*t+z**Integer(2), x*y - t])
>>> B = A.quotient(I)
>>> B
Graded Commutative Algebra with generators ('x', 'y', 'z', 't')
in degrees (2, 2, 3, 4) with relations [x*t, x*y - t]
over Finite Field of size 5
>>> B(x*t)
0
>>> B(x*y)
t
>>> A.basis(Integer(7))
[x^2*z, x*y*z, y^2*z, z*t]

```

(continues on next page)

(continued from previous page)

```
>>> B.basis(Integer(7))
[x^2*z, y^2*z, z*t]
```

```
sage.algebras.commutative_dga.GradedCommutativeAlgebra(ring, names=None, degrees=None,
max_degree=None, **kwargs)
```

A graded commutative algebra.

INPUT:

There are two ways to call this. The first way defines a free graded commutative algebra:

- `ring` – the base field over which to work
- `names` – names of the generators. You may also use Sage's `A.<x,y,...> = ...` syntax to define the names. If no names are specified, the generators are named `x0`, `x1`, ...
- `degrees` – degrees of the generators; if this is omitted, the degree of each generator is 1, and if both `names` and `degrees` are omitted, an error is raised
- `max_degree` – the maximal degree of the graded algebra. If omitted, no maximal degree is assumed and an instance of `GCAAlgebra` is returned. Otherwise, an instance of `sage.algebras.commutative_graded_algebra.GradedCommutativeAlgebraWithMaxDeg` is created.

Once such an algebra has been defined, one can use its associated methods to take a quotient, impose a differential, etc. See the examples below.

The second way takes a graded commutative algebra and imposes relations:

- `ring` – a graded commutative algebra
- `relations` – list or tuple of elements of `ring`

EXAMPLES:

Defining a graded commutative algebra:

```
sage: GradedCommutativeAlgebra(QQ, 'x, y, z')
Graded Commutative Algebra with generators ('x', 'y', 'z')
in degrees (1, 1, 1) over Rational Field
sage: GradedCommutativeAlgebra(QQ, degrees=(2, 3, 4))
Graded Commutative Algebra with generators ('x0', 'x1', 'x2')
in degrees (2, 3, 4) over Rational Field
```

```
>>> from sage.all import *
>>> GradedCommutativeAlgebra(QQ, 'x, y, z')
Graded Commutative Algebra with generators ('x', 'y', 'z')
in degrees (1, 1, 1) over Rational Field
>>> GradedCommutativeAlgebra(QQ, degrees=(Integer(2), Integer(3), Integer(4)))
Graded Commutative Algebra with generators ('x0', 'x1', 'x2')
in degrees (2, 3, 4) over Rational Field
```

As usual in Sage, the `A.<...>` notation defines both the algebra and the generator names:

```
sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(1, 1, 2))
sage: x^2
0
sage: y*x # Odd classes anticommute.
-x*y
```

(continues on next page)

(continued from previous page)

```

sage: z*y # z is central since it is in degree 2.
y*z
sage: (x*y*z**3).degree()
8
sage: A.basis(3) # basis of homogeneous degree 3 elements
[x*z, y*z]

```

```

>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(1), Integer(1), Integer(2)),
↳ names=('x', 'y', 'z')); (x, y, z) = A._first_ngens(3)
>>> x**Integer(2)
0
>>> y*x # Odd classes anticommute.
-x*y
>>> z*y # z is central since it is in degree 2.
y*z
>>> (x*y*z**Integer(3)).degree()
8
>>> A.basis(Integer(3)) # basis of homogeneous degree 3 elements
[x*z, y*z]

```

Defining a quotient:

```

sage: I = A.ideal(x*z)
sage: AQ = A.quotient(I); AQ
Graded Commutative Algebra with generators ('x', 'y', 'z')
in degrees (1, 1, 2) with relations [x*z] over Rational Field
sage: AQ.basis(3)
[y*z]

```

```

>>> from sage.all import *
>>> I = A.ideal(x*z)
>>> AQ = A.quotient(I); AQ
Graded Commutative Algebra with generators ('x', 'y', 'z')
in degrees (1, 1, 2) with relations [x*z] over Rational Field
>>> AQ.basis(Integer(3))
[y*z]

```

Note that `AQ` has no specified differential. This is reflected in its print representation: `AQ` is described as a “graded commutative algebra” – the word “differential” is missing. Also, it has no default differential:

```

sage: AQ.differential()
Traceback (most recent call last):
...
TypeError: ...differential() missing 1 required positional argument:
'diff'

```

```

>>> from sage.all import *
>>> AQ.differential()
Traceback (most recent call last):
...

```

(continues on next page)

(continued from previous page)

```
TypeError: ...differential() missing 1 required positional argument:
'diff'
```

Now we add a differential to AQ:

```
sage: B = AQ.cdg_algebra({z: y*z}); B
Commutative Differential Graded Algebra with generators ('x', 'y', 'z')
in degrees (1, 1, 2) with relations [x*z] over Rational Field with differential:
  x --> 0
  y --> 0
  z --> y*z
sage: B.differential()
Differential of Commutative Differential Graded Algebra with generators
('x', 'y', 'z') in degrees (1, 1, 2) with relations [x*z] over Rational Field
Defn: x --> 0
      y --> 0
      z --> y*z
sage: B.cohomology(1)
Free module generated by {[x], [y]} over Rational Field
sage: B.cohomology(2)
Free module generated by {[x*y]} over Rational Field
```

```
>>> from sage.all import *
>>> B = AQ.cdg_algebra({z: y*z}); B
Commutative Differential Graded Algebra with generators ('x', 'y', 'z')
in degrees (1, 1, 2) with relations [x*z] over Rational Field with differential:
  x --> 0
  y --> 0
  z --> y*z
>>> B.differential()
Differential of Commutative Differential Graded Algebra with generators
('x', 'y', 'z') in degrees (1, 1, 2) with relations [x*z] over Rational Field
Defn: x --> 0
      y --> 0
      z --> y*z
>>> B.cohomology(Integer(1))
Free module generated by {[x], [y]} over Rational Field
>>> B.cohomology(Integer(2))
Free module generated by {[x*y]} over Rational Field
```

We compute algebra generators for cohomology in a range of degrees. This cohomology algebra appears to be finitely generated:

```
sage: B.cohomology_generators(15)
{1: [x, y]}
```

```
>>> from sage.all import *
>>> B.cohomology_generators(Integer(15))
{1: [x, y]}
```

We can construct multi-graded rings as well. We work in characteristic 2 for a change, so the algebras here are honestly commutative:

```

sage: C.<a,b,c,d> = GradedCommutativeAlgebra(GF(2),
.....:                                     degrees=((1,0), (1,1), (0,2), (0,3)))
sage: D = C.cdg_algebra(differential={a: c, b: d}); D
Commutative Differential Graded Algebra with generators ('a', 'b', 'c', 'd')
in degrees ((1, 0), (1, 1), (0, 2), (0, 3)) over Finite Field of size 2
with differential:
  a --> c
  b --> d
  c --> 0
  d --> 0

```

```

>>> from sage.all import *
>>> C = GradedCommutativeAlgebra(GF(Integer(2)),
...                               degrees=((Integer(1),Integer(0)),
->(Integer(1),Integer(1)), (Integer(0),Integer(2)), (Integer(0),Integer(3))),
->names=('a', 'b', 'c', 'd')); (a, b, c, d,) = C._first_ngens(4)
>>> D = C.cdg_algebra(differential={a: c, b: d}); D
Commutative Differential Graded Algebra with generators ('a', 'b', 'c', 'd')
in degrees ((1, 0), (1, 1), (0, 2), (0, 3)) over Finite Field of size 2
with differential:
  a --> c
  b --> d
  c --> 0
  d --> 0

```

We can examine D using both total degrees and multidegrees. Use tuples, lists, vectors, or elements of additive abelian groups to specify degrees:

```

sage: D.basis(3)          # basis in total degree 3
[a^3, a*b, a*c, d]
sage: D.basis((1,2))     # basis in degree (1,2)
[a*c]
sage: D.basis([1,2])
[a*c]
sage: D.basis(vector([1,2]))
[a*c]
sage: G = AdditiveAbelianGroup([0,0]); G
Additive abelian group isomorphic to Z + Z
sage: D.basis(G(vector([1,2])))
[a*c]

```

```

>>> from sage.all import *
>>> D.basis(Integer(3))    # basis in total degree 3
[a^3, a*b, a*c, d]
>>> D.basis((Integer(1),Integer(2))) # basis in degree (1,2)
[a*c]
>>> D.basis([Integer(1),Integer(2)])
[a*c]
>>> D.basis(vector([Integer(1),Integer(2)]))
[a*c]
>>> G = AdditiveAbelianGroup([Integer(0),Integer(0)]); G
Additive abelian group isomorphic to Z + Z

```

(continues on next page)

(continued from previous page)

```
>>> D.basis(G(vector([Integer(1), Integer(2)])))
[a*c]
```

At this point, a , for example, is an element of C . We can redefine it so that it is instead an element of D in several ways, for instance using `gens()` method:

```
sage: a, b, c, d = D.gens()
sage: a.differential()
c
```

```
>>> from sage.all import *
>>> a, b, c, d = D.gens()
>>> a.differential()
c
```

Or the `inject_variables()` method:

```
sage: D.inject_variables()
Defining a, b, c, d
sage: (a*b).differential()
b*c + a*d
sage: (a*b*c**2).degree()
(2, 5)
```

```
>>> from sage.all import *
>>> D.inject_variables()
Defining a, b, c, d
>>> (a*b).differential()
b*c + a*d
>>> (a*b*c**Integer(2)).degree()
(2, 5)
```

Degrees are returned as elements of additive abelian groups:

```
sage: (a*b*c**2).degree() in G
True

sage: (a*b*c**2).degree(total=True) # total degree
7
sage: D.cohomology(4)
Free module generated by {[a^4], [b^2]} over Finite Field of size 2
sage: D.cohomology((2,2))
Free module generated by {[b^2]} over Finite Field of size 2
```

```
>>> from sage.all import *
>>> (a*b*c**Integer(2)).degree() in G
True

>>> (a*b*c**Integer(2)).degree(total=True) # total degree
7
>>> D.cohomology(Integer(4))
Free module generated by {[a^4], [b^2]} over Finite Field of size 2
```

(continues on next page)

(continued from previous page)

```
>>> D.cohomology((Integer(2),Integer(2)))
Free module generated by {[b^2]} over Finite Field of size 2
```

Graded algebra with maximal degree:

```
sage: A.<p,e> = GradedCommutativeAlgebra(QQ, degrees=(4,2), max_degree=6); A
Graded commutative algebra with generators ('p', 'e') in degrees (4, 2)
with maximal degree 6
sage: p^2
0
```

```
>>> from sage.all import *
>>> A = GradedCommutativeAlgebra(QQ, degrees=(Integer(4),Integer(2)), max_
↪degree=Integer(6), names=('p', 'e',)); (p, e,) = A._first_ngens(2); A
Graded commutative algebra with generators ('p', 'e') in degrees (4, 2)
with maximal degree 6
>>> p**Integer(2)
0
```

sage.algebras.commutative_dga.**exterior_algebra_basis**(degrees)

Basis of an exterior algebra in degree n, where the generators are in degrees degrees.

INPUT:

- n – integer
- degrees – iterable of integers

Return list of lists, each list representing exponents for the corresponding generators. (So each list consists of 0s and 1s.)

EXAMPLES:

```
sage: from sage.algebras.commutative_dga import exterior_algebra_basis
sage: exterior_algebra_basis(1, (1,3,1))
[[0, 0, 1], [1, 0, 0]]
sage: exterior_algebra_basis(4, (1,3,1))
[[0, 1, 1], [1, 1, 0]]
sage: exterior_algebra_basis(10, (1,5,1,1))
[]
```

```
>>> from sage.all import *
>>> from sage.algebras.commutative_dga import exterior_algebra_basis
>>> exterior_algebra_basis(Integer(1), (Integer(1),Integer(3),Integer(1)))
[[0, 0, 1], [1, 0, 0]]
>>> exterior_algebra_basis(Integer(4), (Integer(1),Integer(3),Integer(1)))
[[0, 1, 1], [1, 1, 0]]
>>> exterior_algebra_basis(Integer(10), (Integer(1),Integer(5),Integer(1),
↪Integer(1)))
[]
```

sage.algebras.commutative_dga.**sorting_keys**(element)

Auxiliary function to sort the elements of a basis of a Cohomology group.

It is needed to ensure that elements of a cohomology group are represented in a consistent way.

INPUT:

- element – a CohomologyClass

OUTPUT: its coordinates in the corresponding `cohomology_raw` quotient vector space

EXAMPLES:

```
sage: from sage.algebras.commutative_dga import sorting_keys
sage: A.<e1,e2,e3,e4,e5> = GradedCommutativeAlgebra(QQ)
sage: B = A.cdg_algebra({e5:e1*e2+e3*e4})
sage: B.inject_variables()
Defining e1, e2, e3, e4, e5
sage: C = B.cohomology(3)
sage: [sorting_keys(e1) for e1 in C.basis().keys()]
[[1, 0, 0, 0, 0],
 [0, 1, 0, 0, 0],
 [0, 0, 1, 0, 0],
 [0, 0, 0, 1, 0],
 [0, 0, 0, 0, 1]]
```

```
>>> from sage.all import *
>>> from sage.algebras.commutative_dga import sorting_keys
>>> A = GradedCommutativeAlgebra(QQ, names=('e1', 'e2', 'e3', 'e4', 'e5')); (e1,
↪e2, e3, e4, e5,) = A._first_ngens(5)
>>> B = A.cdg_algebra({e5:e1*e2+e3*e4})
>>> B.inject_variables()
Defining e1, e2, e3, e4, e5
>>> C = B.cohomology(Integer(3))
>>> [sorting_keys(e1) for e1 in C.basis().keys()]
[[1, 0, 0, 0, 0],
 [0, 1, 0, 0, 0],
 [0, 0, 1, 0, 0],
 [0, 0, 0, 1, 0],
 [0, 0, 0, 0, 1]]
```

`sage.algebras.commutative_dga.total_degree(deg)`

Total degree of `deg`.

INPUT:

- `deg` – an element of a free abelian group

In fact, `deg` could be an integer, a Python int, a list, a tuple, a vector, etc. This function returns the sum of the components of `deg`.

EXAMPLES:

```
sage: from sage.algebras.commutative_dga import total_degree
sage: total_degree(12)
12
sage: total_degree(range(5))
10
sage: total_degree(vector(range(5)))
10
sage: G = AdditiveAbelianGroup((0,0))
sage: x = G.gen(0); y = G.gen(1)
```

(continues on next page)

(continued from previous page)

```
sage: 3*x+4*y
(3, 4)
sage: total_degree(3*x+4*y)
7
```

```
>>> from sage.all import *
>>> from sage.algebras.commutative_dga import total_degree
>>> total_degree(Integer(12))
12
>>> total_degree(range(Integer(5)))
10
>>> total_degree(vector(range(Integer(5))))
10
>>> G = AdditiveAbelianGroup((Integer(0), Integer(0)))
>>> x = G.gen(Integer(0)); y = G.gen(Integer(1))
>>> Integer(3)*x+Integer(4)*y
(3, 4)
>>> total_degree(Integer(3)*x+Integer(4)*y)
7
```


VARIOUS ASSOCIATIVE ALGEBRAS

8.1 Associated Graded Algebras To Filtered Algebras

AUTHORS:

- Travis Scrimshaw (2014-10-08): Initial version

class sage.algebras.associated_graded.**AssociatedGradedAlgebra** (*A*, category=None)

Bases: CombinatorialFreeModule

The associated graded algebra/module $\text{gr } A$ of a filtered algebra/module with basis A .

Let A be a filtered module over a commutative ring R . Let $(F_i)_{i \in I}$ be the filtration of A , with I being a totally ordered set. Define

$$G_i = F_i / \sum_{j < i} F_j$$

for every $i \in I$, and then

$$\text{gr } A = \bigoplus_{i \in I} G_i.$$

There are canonical projections $p_i : F_i \rightarrow G_i$ for every $i \in I$. Moreover $\text{gr } A$ is naturally a graded R -module with G_i being the i -th graded component. This graded R -module is known as the *associated graded module* (or, for short, just *graded module*) of A .

Now, assume that A (endowed with the filtration $(F_i)_{i \in I}$) is not just a filtered R -module, but also a filtered R -algebra. Let $u \in G_i$ and $v \in G_j$, and let $u' \in F_i$ and $v' \in F_j$ be lifts of u and v , respectively (so that $u = p_i(u')$ and $v = p_j(v')$). Then, we define a multiplication $*$ on $\text{gr } A$ (not to be mistaken for the multiplication of the original algebra A) by

$$u * v = p_{i+j}(u'v').$$

The *associated graded algebra* (or, for short, just *graded algebra*) of A is the graded algebra $\text{gr } A$ (endowed with this multiplication).

Now, assume that A is a filtered R -algebra with basis. Let $(b_x)_{x \in X}$ be the basis of A , and consider the partition $X = \bigsqcup_{i \in I} X_i$ of the set X , which is part of the data of a filtered algebra with basis. We know (see `FilteredModulesWithBasis`) that A (being a filtered R -module with basis) is canonically (when the basis is considered to be part of the data) isomorphic to $\text{gr } A$ as an R -module. Therefore the k -th graded component G_k can be identified with the span of $(b_x)_{x \in X_k}$, or equivalently the k -th homogeneous component of A . Suppose that $u'v' = \sum_{k \leq i+j} m_k$ where $m_k \in G_k$ (which has been identified with the k -th homogeneous component of A). Then $u * v = m_{i+j}$. We also note that the choice of identification of G_k with the k -th homogeneous component of A depends on the given basis.

The basis $(b_x)_{x \in X}$ of A gives rise to a basis of $\text{gr } A$. This latter basis is still indexed by the elements of X , and consists of the images of the b_x under the R -module isomorphism from A to $\text{gr } A$. It makes $\text{gr } A$ into a graded R -algebra with basis.

In this class, the R -module isomorphism from A to $\text{gr } A$ is implemented as `to_graded_conversion()` and also as the default conversion from A to $\text{gr } A$. Its inverse map is implemented as `from_graded_conversion()`. The projection $p_i : F_i \rightarrow G_i$ is implemented as `projection() (i)`.

INPUT:

- A – a filtered module (or algebra) with basis

OUTPUT:

The associated graded module of A , if A is just a filtered R -module. The associated graded algebra of A , if A is a filtered R -algebra.

EXAMPLES:

Associated graded module of a filtered module:

```
sage: A = Modules(QQ).WithBasis().Filtered().example()
sage: grA = A.graded_algebra()
sage: grA.category()
Category of graded vector spaces with basis over Rational Field
sage: x = A.basis()[Partition([3,2,1])]
sage: grA(x)
Bbar[[3, 2, 1]]
```

```
>>> from sage.all import *
>>> A = Modules(QQ).WithBasis().Filtered().example()
>>> grA = A.graded_algebra()
>>> grA.category()
Category of graded vector spaces with basis over Rational Field
>>> x = A.basis()[Partition([Integer(3), Integer(2), Integer(1)])]
>>> grA(x)
Bbar[[3, 2, 1]]
```

Associated graded algebra of a filtered algebra:

```
sage: A = Algebras(QQ).WithBasis().Filtered().example()
sage: grA = A.graded_algebra()
sage: grA.category()
Category of graded algebras with basis over Rational Field
sage: x,y,z = [grA.algebra_generators()[s] for s in ['x','y','z']]
sage: x
bar(U['x'])
sage: y * x + z
bar(U['x']*U['y']) + bar(U['z'])
sage: A(y) * A(x) + A(z)
U['x']*U['y']
```

```
>>> from sage.all import *
>>> A = Algebras(QQ).WithBasis().Filtered().example()
>>> grA = A.graded_algebra()
>>> grA.category()
```

(continues on next page)

(continued from previous page)

```

Category of graded algebras with basis over Rational Field
>>> x,y,z = [grA.algebra_generators()[s] for s in ['x','y','z']]
>>> x
bar(U['x'])
>>> y * x + z
bar(U['x']*U['y']) + bar(U['z'])
>>> A(y) * A(x) + A(z)
U['x']*U['y']

```

We note that the conversion between A and grA is the canonical $\mathbb{Q}\mathbb{Q}$ -module isomorphism stemming from the fact that the underlying $\mathbb{Q}\mathbb{Q}$ -modules of A and grA are isomorphic:

```

sage: grA(A.an_element())
bar(U['x']^2*U['y']^2*U['z']^3) + 2*bar(U['x']) + 3*bar(U['y']) + bar(1)
sage: elt = A.an_element() + A.algebra_generators()['x'] + 2
sage: grelt = grA(elt); grelt
bar(U['x']^2*U['y']^2*U['z']^3) + 3*bar(U['x']) + 3*bar(U['y']) + 3*bar(1)
sage: A(grelt) == elt
True

```

```

>>> from sage.all import *
>>> grA(A.an_element())
bar(U['x']^2*U['y']^2*U['z']^3) + 2*bar(U['x']) + 3*bar(U['y']) + bar(1)
>>> elt = A.an_element() + A.algebra_generators()['x'] + Integer(2)
>>> grelt = grA(elt); grelt
bar(U['x']^2*U['y']^2*U['z']^3) + 3*bar(U['x']) + 3*bar(U['y']) + 3*bar(1)
>>> A(grelt) == elt
True

```

Todo

The algebra A must currently be an instance of (a subclass of) `CombinatorialFreeModule`. This should work with any filtered algebra with a basis.

Todo

Implement a version of associated graded algebra for filtered algebras without a distinguished basis.

REFERENCES:

- [Wikipedia article Filtered_algebra#Associated_graded_algebra](#)

`algebra_generators()`

Return the algebra generators of `self`.

This assumes that the algebra generators of A provided by its `algebra_generators` method are homogeneous.

EXAMPLES:

```
sage: A = Algebras(QQ).WithBasis().Filtered().example()
sage: grA = A.graded_algebra()
sage: grA.algebra_generators()
Finite family {'x': bar(U['x']), 'y': bar(U['y']), 'z': bar(U['z'])}
```

```
>>> from sage.all import *
>>> A = Algebras(QQ).WithBasis().Filtered().example()
>>> grA = A.graded_algebra()
>>> grA.algebra_generators()
Finite family {'x': bar(U['x']), 'y': bar(U['y']), 'z': bar(U['z'])}
```

degree_on_basis(x)

Return the degree of the basis element indexed by x .

EXAMPLES:

```
sage: A = Algebras(QQ).WithBasis().Filtered().example()
sage: grA = A.graded_algebra()
sage: all(A.degree_on_basis(x) == grA.degree_on_basis(x)
....:      for g in grA.algebra_generators() for x in g.support())
True
```

```
>>> from sage.all import *
>>> A = Algebras(QQ).WithBasis().Filtered().example()
>>> grA = A.graded_algebra()
>>> all(A.degree_on_basis(x) == grA.degree_on_basis(x)
...     for g in grA.algebra_generators() for x in g.support())
True
```

gen(*args, **kws)

Return a generator of self .

EXAMPLES:

```
sage: A = Algebras(QQ).WithBasis().Filtered().example()
sage: grA = A.graded_algebra()
sage: grA.gen('x')
bar(U['x'])
```

```
>>> from sage.all import *
>>> A = Algebras(QQ).WithBasis().Filtered().example()
>>> grA = A.graded_algebra()
>>> grA.gen('x')
bar(U['x'])
```

one_basis()

Return the basis index of the element 1 of $\text{gr } A$.

This assumes that the unity 1 of A belongs to F_0 .

EXAMPLES:

```
sage: A = Algebras(QQ).WithBasis().Filtered().example()
sage: grA = A.graded_algebra()
```

(continues on next page)

(continued from previous page)

```
sage: grA.one_basis()
1
```

```
>>> from sage.all import *
>>> A = Algebras(QQ).WithBasis().Filtered().example()
>>> grA = A.graded_algebra()
>>> grA.one_basis()
1
```

product_on_basis(x, y)

Return the product on basis elements given by the indices x and y .

EXAMPLES:

```
sage: A = Algebras(QQ).WithBasis().Filtered().example()
sage: grA = A.graded_algebra()
sage: G = grA.algebra_generators()
sage: x,y,z = G['x'], G['y'], G['z']
sage: x * y # indirect doctest
bar(U['x']*U['y'])
sage: y * x
bar(U['x']*U['y'])
sage: z * y * x
bar(U['x']*U['y']*U['z'])
```

```
>>> from sage.all import *
>>> A = Algebras(QQ).WithBasis().Filtered().example()
>>> grA = A.graded_algebra()
>>> G = grA.algebra_generators()
>>> x,y,z = G['x'], G['y'], G['z']
>>> x * y # indirect doctest
bar(U['x']*U['y'])
>>> y * x
bar(U['x']*U['y'])
>>> z * y * x
bar(U['x']*U['y']*U['z'])
```

8.2 Cellular Basis

Cellular algebras are a class of algebras introduced by Graham and Lehrer [GrLe1996]. The `CellularBasis` class provides a general framework for implementing cellular algebras and their cell modules and simple modules.

Let R be a commutative ring. A R -algebra A is a *cellular algebra* if it has a *cell datum*, which is a tuple (Λ, i, M, C) , where Λ is finite poset with order $\geq$, if $\mu \in \Lambda$ then $T(\mu)$ is a finite set and

$$C: \coprod_{\mu \in \Lambda} T(\mu) \times T(\mu) \longrightarrow A; (\mu, s, t) \mapsto c_{st}^{\mu} \text{ is an injective map}$$

such that the following holds:

- The set $\{c_{st}^{\mu} \mid \mu \in \Lambda, s, t \in T(\mu)\}$ is a basis of A .

- If $a \in A$ and $\mu \in \Lambda$, $s, t \in T(\mu)$ then:

$$ac_{st}^\mu = \sum_{u \in T(\mu)} r_a(s, u) c_{ut}^\mu \pmod{A^{>\mu}},$$

where $A^{>\mu}$ is spanned by

$$\{c_{ab}^\nu \mid \nu > \mu \text{ and } a, b \in T(\nu)\}.$$

Moreover, the scalar $r_a(s, u)$ depends only on a, s and u and, in particular, is independent of t .

- The map $\iota: A \longrightarrow A; c_{st}^\mu \mapsto c_{ts}^\mu$ is an algebra anti-isomorphism.

A *cellular basis* for A is any basis of the form $\{c_{st}^\mu \mid \mu \in \Lambda, s, t \in T(\mu)\}$.

Note that the scalars $r_a(s, u) \in R$ depend only if a, s and u and, in particular, they do not depend on t . It follows from the definition of a cell datum that $A^{>\mu}$ is a two-sided ideal of A . More importantly, if $\mu \in \Lambda$ then the `CellModule` C^μ is the free R -module with basis $\{c_s^\mu \mid \mu \in \Lambda, s \in T(\mu)\}$ and with A -action:

$$ac_s^\mu = \sum_{u \in T(\mu)} r_a(s, u) c_u^\mu,$$

where the scalars $r_a(s, u)$ are those appearing in the definition of the cell datum. It follows from the cellular basis axioms that C^μ comes equipped with a bilinear form $\langle \cdot, \cdot \rangle$ that is determined by:

$$c_{st}^\mu c_u^\mu = \langle c_s^\mu, c_t^\mu \rangle c_u^\mu.$$

The *radical* of C^μ is the A -submodule $\text{rad } C^\mu = \{x \in C^\mu \mid \langle x, y \rangle = 0\}$. Hence, $D^\mu = C^\mu / \text{rad } C^\mu$ is also an A -module. It is not difficult to show that $\{D^\mu \mid D^\mu \neq 0\}$ is a complete set of pairwise non-isomorphic A -modules. Hence, a cell datum for A gives an explicit construction of the irreducible A -modules. The module `simple_module()` D^μ is either zero or absolutely irreducible.

EXAMPLES:

We compute a cellular basis and do some basic computations:

```
sage: S = SymmetricGroupAlgebra(QQ, 3)
sage: C = S.cellular_basis()
sage: C
Cellular basis of Symmetric group algebra of order 3
over Rational Field
```

```
>>> from sage.all import *
>>> S = SymmetricGroupAlgebra(QQ, Integer(3))
>>> C = S.cellular_basis()
>>> C
Cellular basis of Symmetric group algebra of order 3
over Rational Field
```

See also

`CellModule`

AUTHOR:

- Travis Scrimshaw (2015-11-5): Initial version

REFERENCES:

- [GrLe1996]
- [KX1998]
- [Mat1999]
- [Wikipedia article Cellular_algebra](#)
- <http://webusers.imj-prg.fr/~bernhard.keller/ictp2006/lecturenotes/xi.pdf>

class sage.algebras.cellular_basis.**CellularBasis** (*A, to_algebra=None, from_algebra=None, **kwargs*)

Bases: `CombinatorialFreeModule`

The cellular basis of a cellular algebra, in the sense of Graham and Lehrer [GrLe1996].

INPUT:

- A – the cellular algebra

EXAMPLES:

We compute a cellular basis and do some basic computations:

```
sage: S = SymmetricGroupAlgebra(QQ, 3)
sage: C = S.cellular_basis()
sage: C
Cellular basis of Symmetric group algebra of order 3
over Rational Field
sage: len(C.basis())
6
sage: len(S.basis())
6
sage: a,b,c,d,e,f = C.basis()
sage: f
C([3], [[1, 2, 3]], [[1, 2, 3]])
sage: c
C([2, 1], [[1, 3], [2]], [[1, 2], [3]])
sage: d
C([2, 1], [[1, 2], [3]], [[1, 3], [2]])
sage: f * f
C([3], [[1, 2, 3]], [[1, 2, 3]])
sage: f * c
0
sage: d * c
C([2, 1], [[1, 2], [3]], [[1, 2], [3]])
sage: c * d
C([2, 1], [[1, 3], [2]], [[1, 3], [2]])
sage: S(f)
1/6*[1, 2, 3] + 1/6*[1, 3, 2] + 1/6*[2, 1, 3] + 1/6*[2, 3, 1]
+ 1/6*[3, 1, 2] + 1/6*[3, 2, 1]
sage: S(d)
1/4*[1, 3, 2] - 1/4*[2, 3, 1] + 1/4*[3, 1, 2] - 1/4*[3, 2, 1]
sage: B = list(S.basis())
sage: B[2]
[2, 1, 3]
sage: C(B[2])
```

(continues on next page)

(continued from previous page)

```
-C([1, 1, 1], [[1], [2], [3]], [[1], [2], [3]])
+ C([2, 1], [[1, 2], [3]], [[1, 2], [3]])
- C([2, 1], [[1, 3], [2]], [[1, 3], [2]])
+ C([3], [[1, 2, 3]], [[1, 2, 3]])
```

```
>>> from sage.all import *
>>> S = SymmetricGroupAlgebra(QQ, Integer(3))
>>> C = S.cellular_basis()
>>> C
Cellular basis of Symmetric group algebra of order 3
over Rational Field
>>> len(C.basis())
6
>>> len(S.basis())
6
>>> a,b,c,d,e,f = C.basis()
>>> f
C([3], [[1, 2, 3]], [[1, 2, 3]])
>>> c
C([2, 1], [[1, 3], [2]], [[1, 2], [3]])
>>> d
C([2, 1], [[1, 2], [3]], [[1, 3], [2]])
>>> f * f
C([3], [[1, 2, 3]], [[1, 2, 3]])
>>> f * c
0
>>> d * c
C([2, 1], [[1, 2], [3]], [[1, 2], [3]])
>>> c * d
C([2, 1], [[1, 3], [2]], [[1, 3], [2]])
>>> S(f)
1/6*[1, 2, 3] + 1/6*[1, 3, 2] + 1/6*[2, 1, 3] + 1/6*[2, 3, 1]
+ 1/6*[3, 1, 2] + 1/6*[3, 2, 1]
>>> S(d)
1/4*[1, 3, 2] - 1/4*[2, 3, 1] + 1/4*[3, 1, 2] - 1/4*[3, 2, 1]
>>> B = list(S.basis())
>>> B[Integer(2)]
[2, 1, 3]
>>> C(B[Integer(2)])
-C([1, 1, 1], [[1], [2], [3]], [[1], [2], [3]])
+ C([2, 1], [[1, 2], [3]], [[1, 2], [3]])
- C([2, 1], [[1, 3], [2]], [[1, 3], [2]])
+ C([3], [[1, 2, 3]], [[1, 2, 3]])
```

cell_module_indices (*la*)

Return the indices of the cell module of `self` indexed by `la`.

This is the finite set $M(\lambda)$.

EXAMPLES:

```
sage: S = SymmetricGroupAlgebra(QQ, 3)
sage: C = S.cellular_basis()
```

(continues on next page)

(continued from previous page)

```
sage: C.cell_module_indices([2,1])
Standard tableaux of shape [2, 1]
```

```
>>> from sage.all import *
>>> S = SymmetricGroupAlgebra(QQ, Integer(3))
>>> C = S.cellular_basis()
>>> C.cell_module_indices([Integer(2), Integer(1)])
Standard tableaux of shape [2, 1]
```

cell_poset()

Return the cell poset of self.

EXAMPLES:

```
sage: S = SymmetricGroupAlgebra(QQ, 3)
sage: C = S.cellular_basis()
sage: C.cell_poset()
Finite poset containing 3 elements
```

```
>>> from sage.all import *
>>> S = SymmetricGroupAlgebra(QQ, Integer(3))
>>> C = S.cellular_basis()
>>> C.cell_poset()
Finite poset containing 3 elements
```

cellular_basis()

Return the cellular basis of self, which is self.

EXAMPLES:

```
sage: S = SymmetricGroupAlgebra(QQ, 3)
sage: C = S.cellular_basis()
sage: C.cellular_basis() is C
True
```

```
>>> from sage.all import *
>>> S = SymmetricGroupAlgebra(QQ, Integer(3))
>>> C = S.cellular_basis()
>>> C.cellular_basis() is C
True
```

cellular_basis_of()

Return the defining algebra of self.

EXAMPLES:

```
sage: S = SymmetricGroupAlgebra(QQ, 3)
sage: C = S.cellular_basis()
sage: C.cellular_basis_of() is S
True
```

```
>>> from sage.all import *
>>> S = SymmetricGroupAlgebra(QQ, Integer(3))
>>> C = S.cellular_basis()
>>> C.cellular_basis_of() is S
True
```

one()

Return the element 1 in self.

EXAMPLES:

```
sage: S = SymmetricGroupAlgebra(QQ, 3)
sage: C = S.cellular_basis()
sage: C.one()
C([1, 1, 1], [[1], [2], [3]], [[1], [2], [3]])
+ C([2, 1], [[1, 2], [3]], [[1, 2], [3]])
+ C([2, 1], [[1, 3], [2]], [[1, 3], [2]])
+ C([3], [[1, 2, 3]], [[1, 2, 3]])
```

```
>>> from sage.all import *
>>> S = SymmetricGroupAlgebra(QQ, Integer(3))
>>> C = S.cellular_basis()
>>> C.one()
C([1, 1, 1], [[1], [2], [3]], [[1], [2], [3]])
+ C([2, 1], [[1, 2], [3]], [[1, 2], [3]])
+ C([2, 1], [[1, 3], [2]], [[1, 3], [2]])
+ C([3], [[1, 2, 3]], [[1, 2, 3]])
```

product_on_basis(x, y)

Return the product of basis indices by x and y.

EXAMPLES:

```
sage: S = SymmetricGroupAlgebra(QQ, 3)
sage: C = S.cellular_basis()
sage: la = Partition([2, 1])
sage: s = StandardTableau([[1, 2], [3]])
sage: t = StandardTableau([[1, 3], [2]])
sage: C.product_on_basis((la, s, t), (la, s, t))
0
```

```
>>> from sage.all import *
>>> S = SymmetricGroupAlgebra(QQ, Integer(3))
>>> C = S.cellular_basis()
>>> la = Partition([Integer(2), Integer(1)])
>>> s = StandardTableau([[Integer(1), Integer(2)], [Integer(3)]])
>>> t = StandardTableau([[Integer(1), Integer(3)], [Integer(2)]])
>>> C.product_on_basis((la, s, t), (la, s, t))
0
```

8.3 Q-Systems

AUTHORS:

- Travis Scrimshaw (2013-10-08): Initial version
- Travis Scrimshaw (2017-12-08): Added twisted Q-systems

class sage.algebras.q_system.QSystem(base_ring, cartan_type, level, twisted)

Bases: CombinatorialFreeModule

A Q-system.

Let $\mathfrak{g}$ be a tamely-laced symmetrizable Kac-Moody algebra with index set I and Cartan matrix $(C_{ab})_{a,b \in I}$ over a field k . Follow the presentation given in [HKOTY1999], an unrestricted Q-system is a k -algebra in infinitely many variables $Q_m^{(a)}$, where $a \in I$ and $m \in \mathbf{Z}_{>0}$, that satisfies the relations

$$\left(Q_m^{(a)}\right)^2 = Q_{m+1}^{(a)} Q_{m-1}^{(a)} + \prod_{b \sim a} \prod_{k=0}^{-C_{ab}-1} Q_{\left\lfloor \frac{mC_{ba}-k}{C_{ab}} \right\rfloor}^{(b)},$$

with $Q_0^{(a)} := 1$. Q-systems can be considered as T-systems where we forget the spectral parameter u and for $\mathfrak{g}$ of finite type, have a solution given by the characters of Kirillov-Reshetikhin modules (again without the spectral parameter) for an affine Kac-Moody algebra $\widehat{\mathfrak{g}}$ with $\mathfrak{g}$ as its classical subalgebra. See [KNS2011] for more information.

Q-systems have a natural bases given by polynomials of the fundamental representations $Q_1^{(a)}$, for $a \in I$. As such, we consider the Q-system as generated by $\{Q_1^{(a)}\}_{a \in I}$.

There is also a level ℓ restricted Q-system (with unit boundary condition) given by setting $Q_{d_a \ell}^{(a)} = 1$, where d_a are the entries of the symmetrizing matrix for the dual type of $\mathfrak{g}$.

Similarly, for twisted affine types (we omit type $A_{2n}^{(2)}$), we can define the *twisted Q-system* by using the relation:

$$(Q_m^{(a)})^2 = Q_{m+1}^{(a)} Q_{m-1}^{(a)} + \prod_{b \neq a} (Q_m^{(b)})^{-C_{ba}}.$$

See [Wil2013] for more information.

EXAMPLES:

We begin by constructing a Q-system and doing some basic computations in type A_4 :

```
sage: Q = QSystem(QQ, ['A', 4])
sage: Q.Q(3, 1)
Q^(3) [1]
sage: Q.Q(1, 2)
Q^(1) [1]^2 - Q^(2) [1]
sage: Q.Q(3, 3)
-Q^(1) [1]*Q^(3) [1] + Q^(1) [1]*Q^(4) [1]^2 + Q^(2) [1]^2
- 2*Q^(2) [1]*Q^(3) [1]*Q^(4) [1] + Q^(3) [1]^3
sage: x = Q.Q(1, 1) + Q.Q(2, 1); x
Q^(1) [1] + Q^(2) [1]
sage: x * x
Q^(1) [1]^2 + 2*Q^(1) [1]*Q^(2) [1] + Q^(2) [1]^2
```

```
>>> from sage.all import *
>>> Q = QSystem(QQ, ['A', Integer(4)])
```

(continues on next page)

(continued from previous page)

```

>>> Q.Q(Integer(3), Integer(1))
Q^(3) [1]
>>> Q.Q(Integer(1), Integer(2))
Q^(1) [1]^2 - Q^(2) [1]
>>> Q.Q(Integer(3), Integer(3))
-Q^(1) [1]*Q^(3) [1] + Q^(1) [1]*Q^(4) [1]^2 + Q^(2) [1]^2
- 2*Q^(2) [1]*Q^(3) [1]*Q^(4) [1] + Q^(3) [1]^3
>>> x = Q.Q(Integer(1), Integer(1)) + Q.Q(Integer(2), Integer(1)); x
Q^(1) [1] + Q^(2) [1]
>>> x * x
Q^(1) [1]^2 + 2*Q^(1) [1]*Q^(2) [1] + Q^(2) [1]^2

```

Next we do some basic computations in type C_4 :

```

sage: Q = QSystem(QQ, ['C', 4])
sage: Q.Q(4, 1)
Q^(4) [1]
sage: Q.Q(1, 2)
Q^(1) [1]^2 - Q^(2) [1]
sage: Q.Q(2, 3)
Q^(1) [1]^2*Q^(4) [1] - 2*Q^(1) [1]*Q^(2) [1]*Q^(3) [1]
+ Q^(2) [1]^3 - Q^(2) [1]*Q^(4) [1] + Q^(3) [1]^2
sage: Q.Q(3, 3)
Q^(1) [1]*Q^(4) [1]^2 - 2*Q^(2) [1]*Q^(3) [1]*Q^(4) [1] + Q^(3) [1]^3

```

```

>>> from sage.all import *
>>> Q = QSystem(QQ, ['C', Integer(4)])
>>> Q.Q(Integer(4), Integer(1))
Q^(4) [1]
>>> Q.Q(Integer(1), Integer(2))
Q^(1) [1]^2 - Q^(2) [1]
>>> Q.Q(Integer(2), Integer(3))
Q^(1) [1]^2*Q^(4) [1] - 2*Q^(1) [1]*Q^(2) [1]*Q^(3) [1]
+ Q^(2) [1]^3 - Q^(2) [1]*Q^(4) [1] + Q^(3) [1]^2
>>> Q.Q(Integer(3), Integer(3))
Q^(1) [1]*Q^(4) [1]^2 - 2*Q^(2) [1]*Q^(3) [1]*Q^(4) [1] + Q^(3) [1]^3

```

We compare that with the twisted Q-system of type $A_7^{(2)}$:

```

sage: Q = QSystem(QQ, ['A', 7, 2], twisted=True)
sage: Q.Q(4, 1)
Q^(4) [1]
sage: Q.Q(1, 2)
Q^(1) [1]^2 - Q^(2) [1]
sage: Q.Q(2, 3)
Q^(1) [1]^2*Q^(4) [1] - 2*Q^(1) [1]*Q^(2) [1]*Q^(3) [1]
+ Q^(2) [1]^3 - Q^(2) [1]*Q^(4) [1] + Q^(3) [1]^2
sage: Q.Q(3, 3)
-Q^(1) [1]*Q^(3) [1]^2 + Q^(1) [1]*Q^(4) [1]^2 + Q^(2) [1]^2*Q^(3) [1]
- 2*Q^(2) [1]*Q^(3) [1]*Q^(4) [1] + Q^(3) [1]^3

```

```

>>> from sage.all import *
>>> Q = QSystem(QQ, ['A', Integer(7), Integer(2)], twisted=True)
>>> Q.Q(Integer(4), Integer(1))
Q^(4) [1]
>>> Q.Q(Integer(1), Integer(2))
Q^(1) [1]^2 - Q^(2) [1]
>>> Q.Q(Integer(2), Integer(3))
Q^(1) [1]^2*Q^(4) [1] - 2*Q^(1) [1]*Q^(2) [1]*Q^(3) [1]
+ Q^(2) [1]^3 - Q^(2) [1]*Q^(4) [1] + Q^(3) [1]^2
>>> Q.Q(Integer(3), Integer(3))
-Q^(1) [1]*Q^(3) [1]^2 + Q^(1) [1]*Q^(4) [1]^2 + Q^(2) [1]^2*Q^(3) [1]
- 2*Q^(2) [1]*Q^(3) [1]*Q^(4) [1] + Q^(3) [1]^3

```

REFERENCES:

- [HKOTY1999]
- [KNS2011]

class ElementBases: `IndexedFreeModuleElement`

An element of a Q-system.

 $Q(a, m)$ Return the generator $Q_m^{(a)}$ of self.

EXAMPLES:

```

sage: Q = QSystem(QQ, ['A', 8])
sage: Q.Q(2, 1)
Q^(2) [1]
sage: Q.Q(6, 2)
-Q^(5) [1]*Q^(7) [1] + Q^(6) [1]^2
sage: Q.Q(7, 3)
-Q^(5) [1]*Q^(7) [1] + Q^(5) [1]*Q^(8) [1]^2 + Q^(6) [1]^2
- 2*Q^(6) [1]*Q^(7) [1]*Q^(8) [1] + Q^(7) [1]^3
sage: Q.Q(1, 0)
1

```

```

>>> from sage.all import *
>>> Q = QSystem(QQ, ['A', Integer(8)])
>>> Q.Q(Integer(2), Integer(1))
Q^(2) [1]
>>> Q.Q(Integer(6), Integer(2))
-Q^(5) [1]*Q^(7) [1] + Q^(6) [1]^2
>>> Q.Q(Integer(7), Integer(3))
-Q^(5) [1]*Q^(7) [1] + Q^(5) [1]*Q^(8) [1]^2 + Q^(6) [1]^2
- 2*Q^(6) [1]*Q^(7) [1]*Q^(8) [1] + Q^(7) [1]^3
>>> Q.Q(Integer(1), Integer(0))
1

```

Twisted Q-system:

```

sage: Q = QSystem(QQ, ['D', 4, 3], twisted=True)
sage: Q.Q(1, 2)
Q^(1) [1]^2 - Q^(2) [1]
sage: Q.Q(2, 2)
-Q^(1) [1]^3 + Q^(2) [1]^2
sage: Q.Q(2, 3)
3*Q^(1) [1]^4 - 2*Q^(1) [1]^3*Q^(2) [1] - 3*Q^(1) [1]^2*Q^(2) [1]
+ Q^(2) [1]^2 + Q^(2) [1]^3
sage: Q.Q(1, 4)
-2*Q^(1) [1]^2 + 2*Q^(1) [1]^3 + Q^(1) [1]^4
- 3*Q^(1) [1]^2*Q^(2) [1] + Q^(2) [1] + Q^(2) [1]^2

```

```

>>> from sage.all import *
>>> Q = QSystem(QQ, ['D', Integer(4), Integer(3)], twisted=True)
>>> Q.Q(Integer(1), Integer(2))
Q^(1) [1]^2 - Q^(2) [1]
>>> Q.Q(Integer(2), Integer(2))
-Q^(1) [1]^3 + Q^(2) [1]^2
>>> Q.Q(Integer(2), Integer(3))
3*Q^(1) [1]^4 - 2*Q^(1) [1]^3*Q^(2) [1] - 3*Q^(1) [1]^2*Q^(2) [1]
+ Q^(2) [1]^2 + Q^(2) [1]^3
>>> Q.Q(Integer(1), Integer(4))
-2*Q^(1) [1]^2 + 2*Q^(1) [1]^3 + Q^(1) [1]^4
- 3*Q^(1) [1]^2*Q^(2) [1] + Q^(2) [1] + Q^(2) [1]^2

```

algebra_generators()

Return the algebra generators of self.

EXAMPLES:

```

sage: Q = QSystem(QQ, ['A', 4])
sage: Q.algebra_generators()
Finite family {1: Q^(1) [1], 2: Q^(2) [1], 3: Q^(3) [1], 4: Q^(4) [1]}

sage: Q = QSystem(QQ, ['D', 4, 3], twisted=True)
sage: Q.algebra_generators()
Finite family {1: Q^(1) [1], 2: Q^(2) [1]}

```

```

>>> from sage.all import *
>>> Q = QSystem(QQ, ['A', Integer(4)])
>>> Q.algebra_generators()
Finite family {1: Q^(1) [1], 2: Q^(2) [1], 3: Q^(3) [1], 4: Q^(4) [1]}

>>> Q = QSystem(QQ, ['D', Integer(4), Integer(3)], twisted=True)
>>> Q.algebra_generators()
Finite family {1: Q^(1) [1], 2: Q^(2) [1]}

```

cartan_type()

Return the Cartan type of self.

EXAMPLES:


```
sage: Q = QSystem(QQ, ['A', 4])
sage: Q.cartan_type()
['A', 4]

sage: Q = QSystem(QQ, ['D', 4, 3], twisted=True)
sage: Q.cartan_type()
['G', 2, 1]^* relabelled by {0: 0, 1: 2, 2: 1}
```

```
>>> from sage.all import *
>>> Q = QSystem(QQ, ['A', Integer(4)])
>>> Q.cartan_type()
['A', 4]

>>> Q = QSystem(QQ, ['D', Integer(4), Integer(3)], twisted=True)
>>> Q.cartan_type()
['G', 2, 1]^* relabelled by {0: 0, 1: 2, 2: 1}
```

dimension()

Return the dimension of `self`, which is ∞ .

EXAMPLES:

```
sage: F = QSystem(QQ, ['A', 4])
sage: F.dimension()
+Infinity
```

```
>>> from sage.all import *
>>> F = QSystem(QQ, ['A', Integer(4)])
>>> F.dimension()
+Infinity
```

gens()

Return the generators of `self`.

EXAMPLES:

```
sage: Q = QSystem(QQ, ['A', 4])
sage: Q.gens()
(Q^(1) [1], Q^(2) [1], Q^(3) [1], Q^(4) [1])
```

```
>>> from sage.all import *
>>> Q = QSystem(QQ, ['A', Integer(4)])
>>> Q.gens()
(Q^(1) [1], Q^(2) [1], Q^(3) [1], Q^(4) [1])
```

index_set()

Return the index set of `self`.

EXAMPLES:

```
sage: Q = QSystem(QQ, ['A', 4])
sage: Q.index_set()
(1, 2, 3, 4)
```

(continues on next page)

(continued from previous page)

```
sage: Q = QSystem(QQ, ['D', 4, 3], twisted=True)
sage: Q.index_set()
(1, 2)
```

```
>>> from sage.all import *
>>> Q = QSystem(QQ, ['A', Integer(4)])
>>> Q.index_set()
(1, 2, 3, 4)

>>> Q = QSystem(QQ, ['D', Integer(4), Integer(3)], twisted=True)
>>> Q.index_set()
(1, 2)
```

level()

Return the restriction level of `self` or `None` if the system is unrestricted.

EXAMPLES:

```
sage: Q = QSystem(QQ, ['A', 4])
sage: Q.level()

sage: Q = QSystem(QQ, ['A', 4], 5)
sage: Q.level()
5
```

```
>>> from sage.all import *
>>> Q = QSystem(QQ, ['A', Integer(4)])
>>> Q.level()

>>> Q = QSystem(QQ, ['A', Integer(4)], Integer(5))
>>> Q.level()
5
```

one_basis()

Return the basis element indexing 1.

EXAMPLES:

```
sage: Q = QSystem(QQ, ['A', 4])
sage: Q.one_basis()
1
sage: Q.one_basis().parent() is Q._indices
True
```

```
>>> from sage.all import *
>>> Q = QSystem(QQ, ['A', Integer(4)])
>>> Q.one_basis()
1
>>> Q.one_basis().parent() is Q._indices
True
```

`sage.algebras.q_system.is_tamely_laced(ct)`

Check if the Cartan type `ct` is tamely-laced.

A (symmetrizable) Cartan type with index set I is *tamely-laced* if $A_{ij} < -1$ implies $d_i = -A_{ji} = 1$ for all $i, j \in I$, where $(d_i)_{i \in I}$ is the diagonal matrix symmetrizing the Cartan matrix $(A_{ij})_{i,j \in I}$.

EXAMPLES:

```
sage: from sage.algebras.q_system import is_tamely_laced
sage: all(is_tamely_laced(ct)
.....:     for ct in CartanType.samples(crystallographic=True, finite=True))
True
sage: for ct in CartanType.samples(crystallographic=True, affine=True):
.....:     if not is_tamely_laced(ct):
.....:         print(ct)
['A', 1, 1]
['BC', 1, 2]
['BC', 5, 2]
['BC', 1, 2]^*
['BC', 5, 2]^*
sage: cm = CartanMatrix([[2,-1,0,0],[-3,2,-2,-2],[0,-1,2,-1],[0,-1,-1,2]])
sage: is_tamely_laced(cm)
True
```

```
>>> from sage.all import *
>>> from sage.algebras.q_system import is_tamely_laced
>>> all(is_tamely_laced(ct)
...     for ct in CartanType.samples(crystallographic=True, finite=True))
True
>>> for ct in CartanType.samples(crystallographic=True, affine=True):
...     if not is_tamely_laced(ct):
...         print(ct)
['A', 1, 1]
['BC', 1, 2]
['BC', 5, 2]
['BC', 1, 2]^*
['BC', 5, 2]^*
>>> cm = CartanMatrix([[Integer(2),-Integer(1),Integer(0),Integer(0)],[-
↪Integer(3),Integer(2),-Integer(2),-Integer(2)],[Integer(0),-Integer(1),
↪Integer(2),-Integer(1)],[Integer(0),-Integer(1),-Integer(1),Integer(2)]]
>>> is_tamely_laced(cm)
True
```

8.4 q -Commuting Polynomials

AUTHORS:

- Travis Scrimshaw (2022-08-23): Initial version
- Travis Scrimshaw (2023-02-10): Added Laurent polynomials

class `sage.algebras.q_commuting_polynomials.qCommutingLaurentPolynomials( $q, B, names$ )`

Bases: `qCommutingPolynomials_generic`

The algebra of q -commuting Laurent polynomials.

Let R be a commutative ring, and fix an element $q \in R$. Let $B = (B_{xy})_{x,y \in I}$ be a skew-symmetric bilinear form with index set I . Let $R[I]_{q,B}$ denote the Laurent polynomial ring in the variables I such that we have the q -commuting relation for $x, y \in I$:

$$yx = q^{B_{xy}} \cdot xy.$$

This is a graded R -algebra with a natural basis given by monomials written in increasing order with respect to some total order on I .

EXAMPLES:

```
sage: q = ZZ['q'].fraction_field().gen()
sage: R.<x,y> = algebras.qCommutingLaurentPolynomials(q)
```

```
>>> from sage.all import *
>>> q = ZZ['q'].fraction_field().gen()
>>> R = algebras.qCommutingLaurentPolynomials(q, names=('x', 'y',)); (x, y,) = R._
↳first_ngens(2)
```

We verify a case of the q -binomial theorem using inverse variables:

```
sage: f = (x^-1 + y^-1)^10
sage: all(f[b] == q_binomial(10, -b.list()[0]) for b in f.support())
True
```

```
>>> from sage.all import *
>>> f = (x**Integer(1) + y**Integer(1))**Integer(10)
>>> all(f[b] == q_binomial(Integer(10), -b.list()[Integer(0)]) for b in f.
↳support())
True
```

We now do a computation with a non-standard B matrix:

```
sage: B = matrix([[0,1,2],[-1,0,3],[-2,-3,0]])
sage: B
[ 0  1  2]
[-1  0  3]
[-2 -3  0]
sage: q = ZZ['q'].gen()
sage: R.<x,y,z> = algebras.qCommutingLaurentPolynomials(q, B)
sage: y^-1 * x
1/q*x*y^-1
sage: z^-1 * x
1/q^2*x*z^-1
sage: z^-1 * y^-1
q^3*y^-1*z^-1

sage: f = (x + z^-1)^10
sage: all(f[b] == q_binomial(10, b.list()[0], q^-2) for b in f.support())
True

sage: f = (y^-1 + z^-1)^10
sage: all(f[b] == q_binomial(10, -b.list()[1], q^3) for b in f.support())
True
```

```

>>> from sage.all import *
>>> B = matrix([[Integer(0), Integer(1), Integer(2)], [-Integer(1), Integer(0),
↳ Integer(3)], [-Integer(2), -Integer(3), Integer(0)]])
>>> B
[ 0  1  2]
[-1  0  3]
[-2 -3  0]
>>> q = ZZ['q'].gen()
>>> R = algebras.qCommutingLaurentPolynomials(q, B, names=('x', 'y', 'z')); (x, y, z) = R._first_ngens(3)
>>> y**Integer(1) * x
1/q*x*y^-1
>>> z**Integer(1) * x
1/q^2*x*z^-1
>>> z**Integer(1) * y**Integer(1)
q^3*y^-1*z^-1

>>> f = (x + z**Integer(1))**Integer(10)
>>> all(f[b] == q_binomial(Integer(10), b.list()[Integer(0)], q**Integer(2)) for b in f.support())
True

>>> f = (y**Integer(1) + z**Integer(1))**Integer(10)
>>> all(f[b] == q_binomial(Integer(10), -b.list()[Integer(1)], q**Integer(3)) for b in f.support())
True

```

class ElementBases: `IndexedFreeModuleElement`**one_basis()**

Return the basis index of the element 1.

EXAMPLES:

```

sage: q = ZZ['q'].fraction_field().gen()
sage: R.<x,y,z> = algebras.qCommutingPolynomials(q)
sage: R.one_basis()
1

```

```

>>> from sage.all import *
>>> q = ZZ['q'].fraction_field().gen()
>>> R = algebras.qCommutingPolynomials(q, names=('x', 'y', 'z')); (x, y, z) = R._first_ngens(3)
>>> R.one_basis()
1

```

product_on_basis(x, y)Return the product of two monomials given by `x` and `y`.

EXAMPLES:

```

sage: q = ZZ['q'].fraction_field().gen()
sage: R.<x,y> = algebras.qCommutingLaurentPolynomials(q)

```

(continues on next page)

(continued from previous page)

```

sage: R.product_on_basis(x.leading_support(), y.leading_support())
x*y
sage: R.product_on_basis(y.leading_support(), x.leading_support())
q*x*y

sage: x * y
x*y
sage: y * x
q*x*y
sage: y^2 * x
q^2*x*y^2
sage: y * x^2
q^2*x^2*y
sage: y^-2 * x
1/q^2*x*y^-2
sage: y * x^-2
1/q^2*x^-2*y
sage: x * y * x
q*x^2*y
sage: x * y * ~x
1/q*y
sage: y^2 * x^2
q^4*x^2*y^2
sage: y^-2 * x^2
1/q^4*x^2*y^-2
sage: y^-2 * x^-2
q^4*x^-2*y^-2
sage: (x + y)^4
x^4 + (q^3+q^2+q+1)*x^3*y + (q^4+q^3+2*q^2+q+1)*x^2*y^2 + (q^3+q^2+q+1)*x*y^3 +
+ y^4

```

```

>>> from sage.all import *
>>> q = ZZ['q'].fraction_field().gen()
>>> R = algebras.qCommutingLaurentPolynomials(q, names=('x', 'y',)); (x, y,) =
+ R._first_ngens(2)
>>> R.product_on_basis(x.leading_support(), y.leading_support())
x*y
>>> R.product_on_basis(y.leading_support(), x.leading_support())
q*x*y

>>> x * y
x*y
>>> y * x
q*x*y
>>> y**Integer(2) * x
q^2*x*y^2
>>> y * x**Integer(2)
q^2*x^2*y
>>> y**-Integer(2) * x
1/q^2*x*y^-2
>>> y * x**-Integer(2)
1/q^2*x^-2*y

```

(continues on next page)

(continued from previous page)

```

>>> x * y * x
q*x^2*y
>>> x * y * ~x
1/q*y
>>> y**Integer(2) * x**Integer(2)
q^4*x^2*y^2
>>> y**Integer(2) * x**Integer(2)
1/q^4*x^2*y^2
>>> y**Integer(2) * x**Integer(2)
q^4*x^2*y^2
>>> (x + y)**Integer(4)
x^4 + (q^3+q^2+q+1)*x^3*y + (q^4+q^3+2*q^2+q+1)*x^2*y^2 + (q^3+q^2+q+1)*x*y^3 +
+ y^4

```

With a non-standard B matrix:

```

sage: B = matrix([[0,1,2],[-1,0,3],[-2,-3,0]])
sage: q = ZZ['q'].fraction_field().gen()
sage: R.<x,y,z> = algebras.qCommutingLaurentPolynomials(q, B=B)
sage: x * y
x*y
sage: y * x^2
q^2*x^2*y
sage: z^2 * x
q^4*x*z^2
sage: z^2 * x^3
q^12*x^3*z^2
sage: z^2 * y
q^6*y*z^2
sage: z^2 * y^3
q^18*y^3*z^2
sage: x * y^-1
x*y^-1
sage: y * x^-2
1/q^2*x^-2*y
sage: z^-2 * x
1/q^4*x*z^-2
sage: z^-2 * x^-3
q^12*x^-3*z^-2
sage: z^2 * y^-1
1/q^6*y^-1*z^2
sage: z^2 * y^-3
1/q^18*y^-3*z^2

```

```

>>> from sage.all import *
>>> B = matrix([[Integer(0),Integer(1),Integer(2)],[-Integer(1),Integer(0),
Integer(3)],[-Integer(2),-Integer(3),Integer(0)]])
>>> q = ZZ['q'].fraction_field().gen()
>>> R = algebras.qCommutingLaurentPolynomials(q, B=B, names=('x', 'y', 'z'));
(x, y, z) = R._first_ngens(3)
>>> x * y
x*y

```

(continues on next page)

(continued from previous page)

```

>>> y * x**Integer(2)
q^2*x^2*y
>>> z**Integer(2) * x
q^4*x*z^2
>>> z**Integer(2) * x**Integer(3)
q^12*x^3*z^2
>>> z**Integer(2) * y
q^6*y*z^2
>>> z**Integer(2) * y**Integer(3)
q^18*y^3*z^2
>>> x * y**Integer(-1)
x*y^-1
>>> y * x**Integer(-2)
1/q^2*x^-2*y
>>> z**Integer(-2) * x
1/q^4*x*z^-2
>>> z**Integer(-2) * x**Integer(3)
q^12*x^-3*z^-2
>>> z**Integer(2) * y**Integer(-1)
1/q^6*y^-1*z^2
>>> z**Integer(2) * y**Integer(-3)
1/q^18*y^-3*z^2

```

class sage.algebras.q_commuting_polynomials.qCommutingPolynomials(*q, B, names*)

Bases: *qCommutingPolynomials_generic*

The algebra of q -commuting polynomials.

Let R be a commutative ring, and fix an element $q \in R$. Let $B = (B_{xy})_{x,y \in I}$ be a skew-symmetric bilinear form with index set I . Let $R[I]_{q,B}$ denote the polynomial ring in the variables I such that we have the q -commuting relation for $x, y \in I$:

$$yx = q^{B_{xy}} \cdot xy.$$

This is a graded R -algebra with a natural basis given by monomials written in increasing order with respect to some total order on I .

When $B_{xy} = 1$ and $B_{yx} = -1$ for all $x < y$, then we have a q -analog of the classical binomial coefficient theorem:

$$(x + y)^n = \sum_{k=0}^n \binom{n}{k}_q x^k y^{n-k}.$$

EXAMPLES:

```

sage: q = ZZ['q'].fraction_field().gen()
sage: R.<x,y> = algebras.qCommutingPolynomials(q)

```

```

>>> from sage.all import *
>>> q = ZZ['q'].fraction_field().gen()
>>> R = algebras.qCommutingPolynomials(q, names=('x', 'y')); (x, y) = R._first_
  ↪ ngens(2)

```

We verify a case of the q -binomial theorem:


```
sage: f = (x + y)^10
sage: all(f[b] == q_binomial(10, b.list()[0]) for b in f.support())
True
```

```
>>> from sage.all import *
>>> f = (x + y)**Integer(10)
>>> all(f[b] == q_binomial(Integer(10), b.list()[Integer(0)]) for b in f.
↳support())
True
```

We now do a computation with a non-standard B matrix:

```
sage: B = matrix([[0,1,2],[-1,0,3],[-2,-3,0]])
sage: B
[ 0  1  2]
[-1  0  3]
[-2 -3  0]
sage: q = ZZ['q'].gen()
sage: R.<x,y,z> = algebras.qCommutingPolynomials(q, B)
sage: y * x
q*x*y
sage: z * x
q^2*x*z
sage: z * y
q^3*y*z

sage: f = (x + z)^10
sage: all(f[b] == q_binomial(10, b.list()[0], q^2) for b in f.support())
True

sage: f = (y + z)^10
sage: all(f[b] == q_binomial(10, b.list()[1], q^3) for b in f.support())
True
```

```
>>> from sage.all import *
>>> B = matrix([[Integer(0),Integer(1),Integer(2)],[-Integer(1),Integer(0),
↳Integer(3)],[-Integer(2),-Integer(3),Integer(0)]])
>>> B
[ 0  1  2]
[-1  0  3]
[-2 -3  0]
>>> q = ZZ['q'].gen()
>>> R = algebras.qCommutingPolynomials(q, B, names=('x', 'y', 'z')); (x, y, z)↳
↳= R._first_ngens(3)
>>> y * x
q*x*y
>>> z * x
q^2*x*z
>>> z * y
q^3*y*z

>>> f = (x + z)**Integer(10)
```

(continues on next page)

(continued from previous page)

```

>>> all(f[b] == q_binomial(Integer(10), b.list()[Integer(0)], q**Integer(2)) for
↳ b in f.support())
True

>>> f = (y + z)**Integer(10)
>>> all(f[b] == q_binomial(Integer(10), b.list()[Integer(1)], q**Integer(3)) for
↳ b in f.support())
True

```

one_basis()

Return the basis index of the element 1.

EXAMPLES:

```

sage: q = ZZ['q'].fraction_field().gen()
sage: R.<x,y,z> = algebras.qCommutingPolynomials(q)
sage: R.one_basis()
1

```

```

>>> from sage.all import *
>>> q = ZZ['q'].fraction_field().gen()
>>> R = algebras.qCommutingPolynomials(q, names=('x', 'y', 'z')); (x, y, z)
↳ R._first_ngens(3)
>>> R.one_basis()
1

```

product_on_basis(x, y)

Return the product of two monomials given by x and y.

EXAMPLES:

```

sage: q = ZZ['q'].fraction_field().gen()
sage: R.<x,y> = algebras.qCommutingPolynomials(q)
sage: R.product_on_basis(x.leading_support(), y.leading_support())
x*y
sage: R.product_on_basis(y.leading_support(), x.leading_support())
q*x*y

sage: x * y
x*y
sage: y * x
q*x*y
sage: y^2 * x
q^2*x*y^2
sage: y * x^2
q^2*x^2*y
sage: x * y * x
q*x^2*y
sage: y^2 * x^2
q^4*x^2*y^2
sage: (x + y)^2
x^2 + (q+1)*x*y + y^2
sage: (x + y)^3

```

(continues on next page)

(continued from previous page)

```

x^3 + (q^2+q+1)*x^2*y + (q^2+q+1)*x*y^2 + y^3
sage: (x + y)^4
x^4 + (q^3+q^2+q+1)*x^3*y + (q^4+q^3+2*q^2+q+1)*x^2*y^2 + (q^3+q^2+q+1)*x*y^3_
↪ + y^4

```

```

>>> from sage.all import *
>>> q = ZZ['q'].fraction_field().gen()
>>> R = algebras.qCommutingPolynomials(q, names=('x', 'y',)); (x, y,) = R._
↪ first_ngens(2)
>>> R.product_on_basis(x.leading_support(), y.leading_support())
x*y
>>> R.product_on_basis(y.leading_support(), x.leading_support())
q*x*y

>>> x * y
x*y
>>> y * x
q*x*y
>>> y**Integer(2) * x
q^2*x*y^2
>>> y * x**Integer(2)
q^2*x^2*y
>>> x * y * x
q*x^2*y
>>> y**Integer(2) * x**Integer(2)
q^4*x^2*y^2
>>> (x + y)**Integer(2)
x^2 + (q+1)*x*y + y^2
>>> (x + y)**Integer(3)
x^3 + (q^2+q+1)*x^2*y + (q^2+q+1)*x*y^2 + y^3
>>> (x + y)**Integer(4)
x^4 + (q^3+q^2+q+1)*x^3*y + (q^4+q^3+2*q^2+q+1)*x^2*y^2 + (q^3+q^2+q+1)*x*y^3_
↪ + y^4

```

With a non-standard B matrix:

```

sage: B = matrix([[0,1,2],[ -1,0,3],[ -2,-3,0]])
sage: q = ZZ['q'].fraction_field().gen()
sage: R.<x,y,z> = algebras.qCommutingPolynomials(q, B=B)
sage: x * y
x*y
sage: y * x^2
q^2*x^2*y
sage: z^2 * x
q^4*x*z^2
sage: z^2 * x^3
q^12*x^3*z^2
sage: z^2 * y
q^6*y*z^2
sage: z^2 * y^3
q^18*y^3*z^2

```

```

>>> from sage.all import *
>>> B = matrix([[Integer(0), Integer(1), Integer(2)], [-Integer(1), Integer(0),
↳ Integer(3)], [-Integer(2), -Integer(3), Integer(0)]])
>>> q = ZZ['q'].fraction_field().gen()
>>> R = algebras.qCommutingPolynomials(q, B=B, names=('x', 'y', 'z',)); (x, y,
↳ z,) = R._first_ngens(3)
>>> x * y
x*y
>>> y * x**Integer(2)
q^2*x^2*y
>>> z**Integer(2) * x
q^4*x*z^2
>>> z**Integer(2) * x**Integer(3)
q^12*x^3*z^2
>>> z**Integer(2) * y
q^6*y*z^2
>>> z**Integer(2) * y**Integer(3)
q^18*y^3*z^2

```

class sage.algebras.q_commuting_polynomials.**qCommutingPolynomials_generic**(*q, B, indices, names*)

Bases: `CombinatorialFreeModule`

Base class for algebra of q -commuting (Laurent, etc.) polynomials.

Let R be a commutative ring, and fix an element $q \in R$. Let $B = (B_{xy})_{x,y \in I}$ be a skew-symmetric bilinear form with index set I . Let $R[I]_{q,B}$ denote the polynomial ring in the variables I such that we have the q -commuting relation for $x, y \in I$:

$$yx = q^{B_{xy}} \cdot xy.$$

This is a graded R -algebra with a natural basis given by monomials written in increasing order with respect to some total order on I .

algebra_generators()

Return the algebra generators of `self`.

EXAMPLES:

```

sage: q = ZZ['q'].fraction_field().gen()
sage: R.<x,y,z> = algebras.qCommutingPolynomials(q)
sage: R.algebra_generators()
Finite family {'x': x, 'y': y, 'z': z}

```

```

>>> from sage.all import *
>>> q = ZZ['q'].fraction_field().gen()
>>> R = algebras.qCommutingPolynomials(q, names=('x', 'y', 'z',)); (x, y, z,) =
↳ R._first_ngens(3)
>>> R.algebra_generators()
Finite family {'x': x, 'y': y, 'z': z}

```

degree_on_basis(m)

Return the degree of the monomial index by m .

EXAMPLES:

```

sage: q = ZZ['q'].fraction_field().gen()
sage: R.<x,y,z> = algebras.qCommutingPolynomials(q)
sage: R.degree_on_basis(R.one_basis())
0
sage: f = (x + y)^3 + z^3
sage: f.degree()
3

```

```

>>> from sage.all import *
>>> q = ZZ['q'].fraction_field().gen()
>>> R = algebras.qCommutingPolynomials(q, names=('x', 'y', 'z')); (x, y, z) ↪
↪ R._first_ngens(3)
>>> R.degree_on_basis(R.one_basis())
0
>>> f = (x + y)**Integer(3) + z**Integer(3)
>>> f.degree()
3

```

dimension()

Return the dimension of self, which is ∞ .

EXAMPLES:

```

sage: q = ZZ['q'].fraction_field().gen()
sage: R.<x,y,z> = algebras.qCommutingPolynomials(q)
sage: R.dimension()
+Infinity

```

```

>>> from sage.all import *
>>> q = ZZ['q'].fraction_field().gen()
>>> R = algebras.qCommutingPolynomials(q, names=('x', 'y', 'z')); (x, y, z) ↪
↪ R._first_ngens(3)
>>> R.dimension()
+Infinity

```

gen(i)

Return the i -generator of self.

EXAMPLES:

```

sage: q = ZZ['q'].fraction_field().gen()
sage: R.<x,y,z> = algebras.qCommutingPolynomials(q)
sage: R.gen(0)
x
sage: R.gen(2)
z

```

```

>>> from sage.all import *
>>> q = ZZ['q'].fraction_field().gen()
>>> R = algebras.qCommutingPolynomials(q, names=('x', 'y', 'z')); (x, y, z) ↪
↪ R._first_ngens(3)
>>> R.gen(Integer(0))
x

```

(continues on next page)

(continued from previous page)

```
>>> R.gen(Integer(2))
z
```

gens()Return the generators of `self`.

EXAMPLES:

```
sage: q = ZZ['q'].fraction_field().gen()
sage: R.<x,y,z> = algebras.qCommutingPolynomials(q)
sage: R.gens()
(x, y, z)
```

```
>>> from sage.all import *
>>> q = ZZ['q'].fraction_field().gen()
>>> R = algebras.qCommutingPolynomials(q, names=('x', 'y', 'z')); (x, y, z) ↪
↪ R._first_ngens(3)
>>> R.gens()
(x, y, z)
```

q()Return the parameter q .

EXAMPLES:

```
sage: q = ZZ['q'].fraction_field().gen()
sage: R.<x,y,z> = algebras.qCommutingPolynomials(q)
sage: R.q() == q
True
```

```
>>> from sage.all import *
>>> q = ZZ['q'].fraction_field().gen()
>>> R = algebras.qCommutingPolynomials(q, names=('x', 'y', 'z')); (x, y, z) ↪
↪ R._first_ngens(3)
>>> R.q() == q
True
```

8.5 Splitting Algebras

Splitting algebras have been considered by Dan Laksov, Anders Thorup, Torsten Ekedahl and others (see references below) in order to study intersection theory of Grassmann and other flag schemes. Similarly as *splitting fields* they can be considered as extensions of rings containing all the roots of a given monic polynomial over that ring under the assumption that its Galois group is the symmetric group of order equal to the polynomial's degree.

Thus they can be used as a tool to express elements of a ring generated by n indeterminates in terms of symmetric functions in these indeterminates.

This realization of splitting algebras follows the approach of a recursive quotient ring construction splitting off some linear factor of the polynomial in each recursive step. Accordingly it is inherited from `PolynomialQuotientRing_domain`.

AUTHORS:

- Sebastian Oehms (April 2020): initial version

```
class sage.algebras.splitting_algebra.SplittingAlgebra (monic_polynomial, names='X',  
                                                    iterate=True, warning=True)
```

Bases: `PolynomialQuotientRing_domain`

For a given monic polynomial $p(t)$ of degree n over a commutative ring R , the splitting algebra is the universal R -algebra in which $p(t)$ has n roots, or, more precisely, over which $p(t)$ factors,

$$p(t) = (t - \xi_1) \cdots (t - \xi_n).$$

This class creates an algebra as extension over the base ring of a given polynomial p such that p splits into linear factors over that extension. It is assumed (and not checked in general) that the Galois group of p is the symmetric Group $S(n)$. The construction is recursive (following [LT2012], 1.3).

INPUT:

- `monic_polynomial` – the monic polynomial which should be split
- `names` – names for the indeterminates to be adjoined to the base ring of `monic_polynomial`
- `warning` – boolean (default: `True`); can be used (by setting to `False`) to suppress a warning which will be thrown whenever it cannot be checked that the Galois group of `monic_polynomial` is maximal

EXAMPLES:

```
sage: from sage.algebras.splitting_algebra import SplittingAlgebra
sage: Lc.<w> = LaurentPolynomialRing(ZZ)
sage: PabLc.<u,v> = Lc[]; t = polygen(PabLc)
sage: S.<x, y> = SplittingAlgebra(t^3 - u*t^2 + v*t - w)
doctest:...: UserWarning: Assuming x^3 - u*x^2 + v*x - w to have maximal
                        Galois group!

sage: roots = S.splitting_roots(); roots
[x, y, -y - x + u]
sage: all(t^3 - u*t^2 + v*t - w == 0 for t in roots)
True
sage: xi = ~x; xi
(w^-1)*x^2 + ((-w^-1)*u)*x + (w^-1)*v
sage: ~xi == x
True
sage: ~y
((-w^-1)*x)*y + (-w^-1)*x^2 + ((w^-1)*u)*x
sage: zi = ((w^-1)*x)*y; ~zi
-y - x + u

sage: cp3 = cyclotomic_polynomial(3).change_ring(GF(5))
sage: CR3.<e3> = SplittingAlgebra(cp3)
sage: CR3.is_field()
True
sage: CR3.cardinality()
25
sage: F.<a> = cp3.splitting_field()
sage: F.cardinality()
25
sage: E3 = cp3.change_ring(F).roots()[0][0]; E3
3*a + 3
sage: f = CR3.hom([E3]); f
```

(continues on next page)

(continued from previous page)

Ring morphism:

```

From: Splitting Algebra of  $x^2 + x + 1$ 
      with roots  $[e3, 4*e3 + 4]$ 
      over Finite Field of size 5
To:   Finite Field in  $a$  of size  $5^2$ 
Defn:  $e3 \mapsto 3*a + 3$ 

```

```

>>> from sage.all import *
>>> from sage.algebras.splitting_algebra import SplittingAlgebra
>>> Lc = LaurentPolynomialRing(ZZ, names=('w',)); (w,) = Lc._first_ngens(1)
>>> PabLc = Lc['u, v']; (u, v,) = PabLc._first_ngens(2); t = polygen(PabLc)
>>> S = SplittingAlgebra(t**Integer(3) - u*t**Integer(2) + v*t - w, names=('x', 'y',
↪ 't')); (x, y,) = S._first_ngens(2)
doctest:...: UserWarning: Assuming  $x^3 - u*x^2 + v*x - w$  to have maximal
                        Galois group!

>>> roots = S.splitting_roots(); roots
[x, y, -y - x + u]
>>> all(t**Integer(3) - u*t**Integer(2) + v*t - w == Integer(0) for t in roots)
True
>>> xi = ~x; xi
(w^-1)*x^2 + ((-w^-1)*u)*x + (w^-1)*v
>>> ~xi == x
True
>>> ~y
((-w^-1)*x)*y + (-w^-1)*x^2 + ((w^-1)*u)*x
>>> zi = ((w**Integer(1))*x)*y; ~zi
-y - x + u

>>> cp3 = cyclotomic_polynomial(Integer(3)).change_ring(GF(Integer(5)))
>>> CR3 = SplittingAlgebra(cp3, names=('e3',)); (e3,) = CR3._first_ngens(1)
>>> CR3.is_field()
True
>>> CR3.cardinality()
25
>>> F = cp3.splitting_field(names=('a',)); (a,) = F._first_ngens(1)
>>> F.cardinality()
25
>>> E3 = cp3.change_ring(F).roots()[Integer(0)][Integer(0)]; E3
3*a + 3
>>> f = CR3.hom([E3]); f
Ring morphism:
  From: Splitting Algebra of  $x^2 + x + 1$ 
        with roots  $[e3, 4*e3 + 4]$ 
        over Finite Field of size 5
  To:   Finite Field in  $a$  of size  $5^2$ 
  Defn:  $e3 \mapsto 3*a + 3$ 

```

REFERENCES:

- [EL2002]
- [Lak2010]

- [Tho2011]
- [LT2012]

Element

alias of *SplittingAlgebraElement*

defining_polynomial()

Return the defining polynomial of *self*.

EXAMPLES:

```
sage: from sage.algebras.splitting_algebra import SplittingAlgebra
sage: L.<u, v, w> = LaurentPolynomialRing(ZZ)
sage: x = polygen(L)
sage: S = SplittingAlgebra(x^3 - u*x^2 + v*x - w, ('X', 'Y'))
sage: S.defining_polynomial()
x^3 - u*x^2 + v*x - w
```

```
>>> from sage.all import *
>>> from sage.algebras.splitting_algebra import SplittingAlgebra
>>> L = LaurentPolynomialRing(ZZ, names=('u', 'v', 'w')); (u, v, w) = L._
↳first_ngens(3)
>>> x = polygen(L)
>>> S = SplittingAlgebra(x**Integer(3) - u*x**Integer(2) + v*x - w, ('X', 'Y'
↳'))
>>> S.defining_polynomial()
x^3 - u*x^2 + v*x - w
```

hom (*im_gens*, *codomain=None*, *check=True*, *base_map=None*)

This version keeps track with the special recursive structure of *SplittingAlgebra*

Type `Ring.hom?` to see the general documentation of this method. Here you see just special examples for the current class.

EXAMPLES:

```
sage: from sage.algebras.splitting_algebra import SplittingAlgebra
sage: L.<u, v, w> = LaurentPolynomialRing(ZZ); x = polygen(L)
sage: S = SplittingAlgebra(x^3 - u*x^2 + v*x - w, ('X', 'Y'))
sage: P.<x, y, z> = PolynomialRing(ZZ)
sage: F = FractionField(P)
sage: im_gens = [F(g) for g in [y, x, x + y + z, x*y+x*z+y*z, x*y*z]]
sage: f = S.hom(im_gens)
sage: f(u), f(v), f(w)
(x + y + z, x*y + x*z + y*z, x*y*z)
sage: roots = S.splitting_roots(); roots
[X, Y, -Y - X + u]
sage: [f(r) for r in roots]
[x, y, z]
```

```
>>> from sage.all import *
>>> from sage.algebras.splitting_algebra import SplittingAlgebra
>>> L = LaurentPolynomialRing(ZZ, names=('u', 'v', 'w')); (u, v, w) = L._
↳first_ngens(3); x = polygen(L)
```

(continues on next page)

(continued from previous page)

```

>>> S = SplittingAlgebra(x**Integer(3) - u*x**Integer(2) + v*x - w, ('X', 'Y'
↳'))
>>> P = PolynomialRing(ZZ, names=('x', 'y', 'z',)); (x, y, z,) = P._first_
↳ngens(3)
>>> F = FractionField(P)
>>> im_gens = [F(g) for g in [y, x, x + y + z, x*y+x*z+y*z, x*y*z]]
>>> f = S.hom(im_gens)
>>> f(u), f(v), f(w)
(x + y + z, x*y + x*z + y*z, x*y*z)
>>> roots = S.splitting_roots(); roots
[X, Y, -Y - X + u]
>>> [f(r) for r in roots]
[x, y, z]

```

is_completely_split()

Return True if the defining polynomial of `self` splits into linear factors over `self`.

EXAMPLES:

```

sage: from sage.algebras.splitting_algebra import SplittingAlgebra
sage: L.<u, v, w> = LaurentPolynomialRing(ZZ); x = polygen(L)
sage: S.<a,b> = SplittingAlgebra(x^3 - u*x^2 + v*x - w)
sage: S.is_completely_split()
True
sage: S.base_ring().is_completely_split()
False

```

```

>>> from sage.all import *
>>> from sage.algebras.splitting_algebra import SplittingAlgebra
>>> L = LaurentPolynomialRing(ZZ, names=('u', 'v', 'w',)); (u, v, w,) = L._
↳first_ngens(3); x = polygen(L)
>>> S = SplittingAlgebra(x**Integer(3) - u*x**Integer(2) + v*x - w, names=('a'
↳', 'b',)); (a, b,) = S._first_ngens(2)
>>> S.is_completely_split()
True
>>> S.base_ring().is_completely_split()
False

```

lifting_map()

Return a section map from `self` to the cover ring. It is implemented according to the same named method of `QuotientRing_nc`.

EXAMPLES:

```

sage: from sage.algebras.splitting_algebra import SplittingAlgebra
sage: x = polygen(ZZ)
sage: S = SplittingAlgebra(x^2+1, ('I',))
sage: lift = S.lifting_map()
sage: lift(5)
5
sage: r1, r2 = S.splitting_roots()
sage: lift(r1)
I

```

```

>>> from sage.all import *
>>> from sage.algebras.splitting_algebra import SplittingAlgebra
>>> x = polygen(ZZ)
>>> S = SplittingAlgebra(x**Integer(2)+Integer(1), ('I',))
>>> lift = S.lifting_map()
>>> lift(Integer(5))
5
>>> r1, r2 = S.splitting_roots()
>>> lift(r1)
I

```

scalar_base_ring()

Return the ring of scalars of `self` (considered as an algebra).

EXAMPLES:

```

sage: from sage.algebras.splitting_algebra import SplittingAlgebra
sage: L.<u, v, w> = LaurentPolynomialRing(ZZ)
sage: x = polygen(L)
sage: S = SplittingAlgebra(x^3 - u*x^2 + v*x - w, ('X', 'Y'))
sage: S.base_ring()
Factorization Algebra of x^3 - u*x^2 + v*x - w with roots [X]
over Multivariate Laurent Polynomial Ring in u, v, w over Integer Ring
sage: S.scalar_base_ring()
Multivariate Laurent Polynomial Ring in u, v, w over Integer Ring

```

```

>>> from sage.all import *
>>> from sage.algebras.splitting_algebra import SplittingAlgebra
>>> L = LaurentPolynomialRing(ZZ, names=('u', 'v', 'w',)); (u, v, w,) = L._
↳first_ngens(3)
>>> x = polygen(L)
>>> S = SplittingAlgebra(x**Integer(3) - u*x**Integer(2) + v*x - w, ('X', 'Y
↳'))
>>> S.base_ring()
Factorization Algebra of x^3 - u*x^2 + v*x - w with roots [X]
over Multivariate Laurent Polynomial Ring in u, v, w over Integer Ring
>>> S.scalar_base_ring()
Multivariate Laurent Polynomial Ring in u, v, w over Integer Ring

```

splitting_roots()

Return the roots of the split equation.

EXAMPLES:

```

sage: from sage.algebras.splitting_algebra import SplittingAlgebra
sage: x = polygen(ZZ)
sage: S = SplittingAlgebra(x^2+1, ('I',))
sage: S.splitting_roots()
[I, -I]

```

```

>>> from sage.all import *
>>> from sage.algebras.splitting_algebra import SplittingAlgebra
>>> x = polygen(ZZ)

```

(continues on next page)

(continued from previous page)

```
>>> S = SplittingAlgebra(x**Integer(2)+Integer(1), ('I',))
>>> S.splitting_roots()
[I, -I]
```

class sage.algebras.splitting_algebra.**SplittingAlgebraElement** (*parent, polynomial, check=True*)

Bases: `PolynomialQuotientRingElement`

Element class for *SplittingAlgebra*.

EXAMPLES:

```
sage: from sage.algebras.splitting_algebra import SplittingAlgebra
sage: cp6 = cyclotomic_polynomial(6)
sage: CR6.<e6> = SplittingAlgebra(cp6)
sage: type(e6)
<class 'sage.algebras.splitting_algebra.SplittingAlgebra_with_category.element_
↳class'>

sage: type(CR6(5))
<class 'sage.algebras.splitting_algebra.SplittingAlgebra_with_category.element_
↳class'>
```

```
>>> from sage.all import *
>>> from sage.algebras.splitting_algebra import SplittingAlgebra
>>> cp6 = cyclotomic_polynomial(Integer(6))
>>> CR6 = SplittingAlgebra(cp6, names=('e6',)); (e6,) = CR6._first_ngens(1)
>>> type(e6)
<class 'sage.algebras.splitting_algebra.SplittingAlgebra_with_category.element_
↳class'>

>>> type(CR6(Integer(5)))
<class 'sage.algebras.splitting_algebra.SplittingAlgebra_with_category.element_
↳class'>
```

dict (*copy=True*)

Return the dictionary of *self* according to its lift to the cover.

EXAMPLES:

```
sage: from sage.algebras.splitting_algebra import SplittingAlgebra
sage: CR3.<e3> = SplittingAlgebra(cyclotomic_polynomial(3))
sage: f = e3 + 42
sage: f.monomial_coefficients()
{0: 42, 1: 1}
```

```
>>> from sage.all import *
>>> from sage.algebras.splitting_algebra import SplittingAlgebra
>>> CR3 = SplittingAlgebra(cyclotomic_polynomial(Integer(3)), names=('e3',));
↳(e3,) = CR3._first_ngens(1)
>>> f = e3 + Integer(42)
>>> f.monomial_coefficients()
{0: 42, 1: 1}
```

dict is an alias:

```
sage: f.dict()
{0: 42, 1: 1}
```

```
>>> from sage.all import *
>>> f.dict()
{0: 42, 1: 1}
```

is_unit()

Return True if self is invertible.

EXAMPLES:

```
sage: from sage.algebras.splitting_algebra import SplittingAlgebra
sage: CR3.<e3> = SplittingAlgebra(cyclotomic_polynomial(3))
sage: e3.is_unit()
True
```

```
>>> from sage.all import *
>>> from sage.algebras.splitting_algebra import SplittingAlgebra
>>> CR3 = SplittingAlgebra(cyclotomic_polynomial(Integer(3)), names=('e3',));
↳ (e3,) = CR3._first_ngens(1)
>>> e3.is_unit()
True
```

monomial_coefficients (*copy=True*)

Return the dictionary of self according to its lift to the cover.

EXAMPLES:

```
sage: from sage.algebras.splitting_algebra import SplittingAlgebra
sage: CR3.<e3> = SplittingAlgebra(cyclotomic_polynomial(3))
sage: f = e3 + 42
sage: f.monomial_coefficients()
{0: 42, 1: 1}
```

```
>>> from sage.all import *
>>> from sage.algebras.splitting_algebra import SplittingAlgebra
>>> CR3 = SplittingAlgebra(cyclotomic_polynomial(Integer(3)), names=('e3',));
↳ (e3,) = CR3._first_ngens(1)
>>> f = e3 + Integer(42)
>>> f.monomial_coefficients()
{0: 42, 1: 1}
```

dict is an alias:

```
sage: f.dict()
{0: 42, 1: 1}
```

```
>>> from sage.all import *
>>> f.dict()
{0: 42, 1: 1}
```

`sage.algebras.splitting_algebra.solve_with_extension` (*monic_polynomial*, *root_names=None*, *var='x'*, *flatten=False*, *warning=True*)

Return all roots of a monic polynomial in its base ring or in an appropriate extension ring, as far as possible.

INPUT:

- `monic_polynomial` – the monic polynomial whose roots should be created
- `root_names` – names for the indeterminates needed to define the splitting algebra of the `monic_polynomial` (if necessary and possible)
- `var` – (default: `'x'`) for the indeterminate needed to define the splitting field of the `monic_polynomial` (if necessary and possible)
- `flatten` – boolean (default: `True`); if `True` the roots will not be given as a list of pairs (`root`, `multiplicity`) but as a list of roots repeated according to their multiplicity
- `warning` – boolean (default: `True`); can be used (by setting to `False`) to suppress a warning which will be thrown whenever it cannot be checked that the Galois group of `monic_polynomial` is maximal

OUTPUT:

List of tuples (`root`, `multiplicity`) respectively list of roots repeated according to their multiplicity if option `flatten` is `True`.

EXAMPLES:

```
sage: from sage.algebras.splitting_algebra import solve_with_extension
sage: t = polygen(ZZ)
sage: p = t^2 - 2*t + 1
sage: solve_with_extension(p, flatten=True)
[1, 1]
sage: solve_with_extension(p)
[(1, 2)]

sage: cp5 = cyclotomic_polynomial(5, var='T').change_
      ↪ring(UniversalCyclotomicField())
sage: solve_with_extension(cp5)
[(E(5), 1), (E(5)^4, 1), (E(5)^2, 1), (E(5)^3, 1)]
sage: _[0][0].parent()
Universal Cyclotomic Field
```

```
>>> from sage.all import *
>>> from sage.algebras.splitting_algebra import solve_with_extension
>>> t = polygen(ZZ)
>>> p = t**Integer(2) - Integer(2)*t + Integer(1)
>>> solve_with_extension(p, flatten=True)
[1, 1]
>>> solve_with_extension(p)
[(1, 2)]

>>> cp5 = cyclotomic_polynomial(Integer(5), var='T').change_
      ↪ring(UniversalCyclotomicField())
>>> solve_with_extension(cp5)
[(E(5), 1), (E(5)^4, 1), (E(5)^2, 1), (E(5)^3, 1)]
>>> _[Integer(0)][Integer(0)].parent()
Universal Cyclotomic Field
```

NON-ASSOCIATIVE ALGEBRAS

9.1 Octonion Algebras

AUTHORS:

- Travis Scrimshaw (2023-05-06): Initial version

class `sage.algebras.octonion_algebra.Octonion`

Bases: *Octonion_generic*

An octonion.

This is an element of the octonion algebra with parameters $a = b = c = -1$, which is a classical octonion number.

norm()

Return the norm of `self`.

The norm of an octonion x is $\|x\| = \sqrt{xx^*}$, where x^* is the `conjugate()` of x .

See also

This is the square root of `quadratic_form()`.

EXAMPLES:

```
sage: O = OctonionAlgebra(QQ)
sage: elt = sum(i * b for i, b in enumerate(O.basis(), start=2))
sage: elt.norm()
↳needs sage.symbolic
2*sqrt(71)
sage: elt = sum(O.basis())
sage: elt.norm()
↳needs sage.symbolic
2*sqrt(2)
```

```
>>> from sage.all import *
>>> O = OctonionAlgebra(QQ)
>>> elt = sum(i * b for i, b in enumerate(O.basis(), start=Integer(2)))
>>> elt.norm()
↳needs sage.symbolic
2*sqrt(71)
```

(continues on next page)

(continued from previous page)

```
>>> elt = sum(O.basis())
>>> elt.norm()
↳needs sage.symbolic
2*sqrt(2)
```

#_

quadratic_form()

Return the quadratic form of `self`.

The octonion algebra has a distinguished quadratic form given by $N(x) = xx^*$, where x^* is the `conjugate()` of x .

EXAMPLES:

```
sage: O = OctonionAlgebra(QQ)
sage: elt = sum(O.basis()); elt
1 + i + j + k + l + li + lj + lk
sage: elt.quadratic_form()
8
sage: elt * elt.conjugate()
8
```

```
>>> from sage.all import *
>>> O = OctonionAlgebra(QQ)
>>> elt = sum(O.basis()); elt
1 + i + j + k + l + li + lj + lk
>>> elt.quadratic_form()
8
>>> elt * elt.conjugate()
8
```

class `sage.algebras.octonion_algebra.OctonionAlgebra` ($R, a, b, c, names$)

Bases: `UniqueRepresentation, Parent`

The octonion algebra.

Let R be a commutative ring of characteristic not equal to 2. The *octonion algebra* with parameters a, b, c is a non-associative non-commutative unital 8-dimensional R -algebra that is a deformation of the usual octonions, which are when $a = b = c = -1$. The octonions were originally constructed by Graves and independently discovered by Cayley (due to being published first, these are sometimes called the Cayley numbers) and can also be built from the Cayley-Dickson construction with the quaternions.

We use the multiplication table from [Scha1996]. The octonion algebra $\mathbf{O}_{a,b,c}(R)$ is a composition (Hurwitz) algebra, which means it is also an alternative algebra as it satisfies $x^2y = (xx)y = x(xy)$ and $yx^2 = y(xx) = (yx)x$ for all $x, y \in \mathbf{O}_{a,b,c}$.

EXAMPLES:

We first create the classical octonions and perform some basic computations:

```
sage: O = OctonionAlgebra(QQ)
sage: O
Octonion algebra over Rational Field
sage: i, j, k, l = O.gens()
sage: i * j * k
1
```

(continues on next page)

(continued from previous page)

```

sage: k * j * i
-1
sage: (i * k) * l
-1j
sage: i * (k * l)
lj
sage: elt = sum(O.basis())
sage: elt^2
-6 + 2*i + 2*j + 2*k + 2*l + 2*li + 2*lj + 2*lk
sage: prod(O.basis())
1
sage: (i + l)^2
-2
sage: (1 + l) * (1 + l).conjugate()
2
sage: S = O.some_elements()
sage: B = O.basis()
sage: S.extend(x * (i + j/2 - 5*k/3) for x in O.some_elements())
sage: all((x * x) * y == (x * (x * y)) for x in S for y in S)
True
sage: all(y * (x * x) == (y * x) * x for x in S for y in S)
True
sage: all((x + x.conjugate()) / 2 == x.real_part() for x in S)
True
sage: all((x - x.conjugate()) / 2 == x.imag_part() for x in S)
True
sage: all(sum((b*x)*b for b in B) == -6 * x.conjugate() for x in S)
True

```

```

>>> from sage.all import *
>>> O = OctonionAlgebra(QQ)
>>> O
Octonion algebra over Rational Field
>>> i, j, k, l = O.gens()
>>> i * j * k
1
>>> k * j * i
-1
>>> (i * k) * l
-1j
>>> i * (k * l)
lj
>>> elt = sum(O.basis())
>>> elt**Integer(2)
-6 + 2*i + 2*j + 2*k + 2*l + 2*li + 2*lj + 2*lk
>>> prod(O.basis())
1
>>> (i + l)**Integer(2)
-2
>>> (Integer(1) + l) * (Integer(1) + l).conjugate()
2
>>> S = O.some_elements()

```

(continues on next page)

(continued from previous page)

```

>>> B = O.basis()
>>> S.extend(x * (i + j/Integer(2) - Integer(5)*k/Integer(3)) for x in O.some_
↳elements())
>>> all((x * x) * y == (x * (x * y)) for x in S for y in S)
True
>>> all(y * (x * x) == (y * x) * x for x in S for y in S)
True
>>> all((x + x.conjugate()) / Integer(2) == x.real_part() for x in S)
True
>>> all((x - x.conjugate()) / Integer(2) == x.imag_part() for x in S)
True
>>> all(sum((b*x)*b for b in B) == -Integer(6) * x.conjugate() for x in S)
True

```

We construct the (rescaled) E_8 lattice as the integral octonions, which we verify by constructing 240 shortest length elements in the lattice (see also [Wikipedia article E8_lattice#Integral_octonions](#)):

```

sage: m = (i + j + k + 1) / 2
sage: basis = [i, j, i*j, i^2, m, i * m, j * m, (i * j) * m]
sage: basis
[i,
 j,
 -k,
 -1,
 1/2*i + 1/2*j + 1/2*k + 1/2*1,
 -1/2 + 1/2*j - 1/2*k - 1/2*1i,
 -1/2 - 1/2*i + 1/2*k - 1/2*1j,
 1/2 - 1/2*i + 1/2*j + 1/2*1k]
sage: matrix([vector(b) for b in basis]).rank()
8
sage: [b.norm() for b in basis]
[1, 1, 1, 1, 1, 1, 1, 1]
sage: roots = set(basis)
sage: roots.update(-b for b in basis)
sage: new_roots = set(roots) # make a copy
sage: while new_roots:
....:     prev_roots = new_roots
....:     new_roots = set()
....:     for a in prev_roots:
....:         for b in roots:
....:             c = a + b
....:             if c.quadratic_form() != 1 or c in roots:
....:                 continue
....:             new_roots.update([c, -c])
....:             roots.update(new_roots)
sage: len(roots)
240

```

```

>>> from sage.all import *
>>> m = (i + j + k + 1) / Integer(2)
>>> basis = [i, j, i*j, i**Integer(2), m, i * m, j * m, (i * j) * m]
>>> basis

```

(continues on next page)

(continued from previous page)

```

[i,
 j,
 -k,
 -1,
 1/2*i + 1/2*j + 1/2*k + 1/2*l,
 -1/2 + 1/2*j - 1/2*k - 1/2*li,
 -1/2 - 1/2*i + 1/2*k - 1/2*lj,
 1/2 - 1/2*i + 1/2*j + 1/2*lk]
>>> matrix([vector(b) for b in basis]).rank()
8
>>> [b.norm() for b in basis]
[1, 1, 1, 1, 1, 1, 1, 1]
>>> roots = set(basis)
>>> roots.update(-b for b in basis)
>>> new_roots = set(roots) # make a copy
>>> while new_roots:
...     prev_roots = new_roots
...     new_roots = set()
...     for a in prev_roots:
...         for b in roots:
...             c = a + b
...             if c.quadratic_form() != Integer(1) or c in roots:
...                 continue
...             new_roots.update([c, -c])
...     roots.update(new_roots)
>>> len(roots)
240

```

A classical construction of the Lie algebra of type G_2 is the Lie algebra of all derivations of $\mathbf{O}$ (as the automorphism group is the Lie group of type G_2). We verify that the derivations have the correct dimension:

```

sage: len(O.derivations_basis())
14

```

```

>>> from sage.all import *
>>> len(O.derivations_basis())
14

```

We can construct the split octonions by taking the parameter $c = 1$:

```

sage: SO = OctonionAlgebra(QQ, c=1)
sage: SO
Octonion algebra over Rational Field with parameters (-1, -1, 1)
sage: i, j, k, l = SO.gens()
sage: i^2 == j^2 == k^2 == -1
True
sage: l^2
1
sage: (i + l)^2
0
sage: (1 + l) * (1 + l).conjugate()
0

```

```

>>> from sage.all import *
>>> SO = OctonionAlgebra(QQ, c=Integer(1))
>>> SO
Octonion algebra over Rational Field with parameters (-1, -1, 1)
>>> i, j, k, l = SO.gens()
>>> i**Integer(2) == j**Integer(2) == k**Integer(2) == -Integer(1)
True
>>> l**Integer(2)
1
>>> (i + l)**Integer(2)
0
>>> (Integer(1) + l) * (Integer(1) + l).conjugate()
0

```

REFERENCES:

- [Scha1996]
- [Wikipedia article octonion](#)
- [Wikipedia article Split-octonion](#)
- [Wikipedia article Hurwitz's_theorem_\(composition_algebras\)](#)

Element

alias of *Octonion_generic*

basis()

Return the basis of *self*.

EXAMPLES:

```

sage: O = OctonionAlgebra(ZZ)
sage: O.basis()
Family (1, i, j, k, l, li, lj, lk)

```

```

>>> from sage.all import *
>>> O = OctonionAlgebra(ZZ)
>>> O.basis()
Family (1, i, j, k, l, li, lj, lk)

```

gens()

Return the generators of *self*.

EXAMPLES:

```

sage: O = OctonionAlgebra(ZZ)
sage: O.gens()
(i, j, k, l)

```

```

>>> from sage.all import *
>>> O = OctonionAlgebra(ZZ)
>>> O.gens()
(i, j, k, l)

```

one_basis()

Return the index for the basis element of 1.

EXAMPLES:

```
sage: O = OctonionAlgebra(ZZ)
sage: O.one_basis()
0
```

```
>>> from sage.all import *
>>> O = OctonionAlgebra(ZZ)
>>> O.one_basis()
0
```

some_elements()

Return some elements of self.

EXAMPLES:

```
sage: O = OctonionAlgebra(ZZ)
sage: O.some_elements()
[2, 1, i, j, k, l, li, lj, lk,
 2 + 3*i + 4*j + 5*k + 6*l + 7*li + 8*lj + 9*lk,
 -2*j + 3*k - li - lj,
 8 - 7*i + 2*j + 13*k - 18*l + 45*li - 40*lj + 5*lk]
sage: O = OctonionAlgebra(Zmod(6))
sage: O.some_elements()
[2, 1, i, j, k, l, li, lj, lk,
 2 + 3*i + 4*j + 5*k + li + 2*lj + 3*lk,
 4*j + 3*k + 5*li + 5*lj,
 2 + 5*i + 2*j + k + 3*li + 2*lj + 5*lk]
```

```
>>> from sage.all import *
>>> O = OctonionAlgebra(ZZ)
>>> O.some_elements()
[2, 1, i, j, k, l, li, lj, lk,
 2 + 3*i + 4*j + 5*k + 6*l + 7*li + 8*lj + 9*lk,
 -2*j + 3*k - li - lj,
 8 - 7*i + 2*j + 13*k - 18*l + 45*li - 40*lj + 5*lk]
>>> O = OctonionAlgebra(Zmod(Integer(6)))
>>> O.some_elements()
[2, 1, i, j, k, l, li, lj, lk,
 2 + 3*i + 4*j + 5*k + li + 2*lj + 3*lk,
 4*j + 3*k + 5*li + 5*lj,
 2 + 5*i + 2*j + k + 3*li + 2*lj + 5*lk]
```

class sage.algebras.octonion_algebra.Octonion_generic

Bases: AlgebraElement

An octonion with generic parameters.

abs()

Return the absolute value of self.

This is equal to the `norm()`.

Warning

If any of the parameters $a, b, c \neq 0$, then this does not define a metric.

EXAMPLES:

```
sage: O = OctonionAlgebra(QQ, 1, 3, 7)
sage: elt = sum(i * b for i, b in enumerate(O.basis(), start=2))
sage: elt.abs()
↳needs sage.symbolic
2*sqrt(-61)
sage: elt = sum(O.basis())
sage: elt.abs()
0
```

```
>>> from sage.all import *
>>> O = OctonionAlgebra(QQ, Integer(1), Integer(3), Integer(7))
>>> elt = sum(i * b for i, b in enumerate(O.basis(), start=Integer(2)))
>>> elt.abs()
↳needs sage.symbolic
2*sqrt(-61)
>>> elt = sum(O.basis())
>>> elt.abs()
0
```

conjugate()

Return the conjugate of `self`.

EXAMPLES:

```
sage: O = OctonionAlgebra(QQ)
sage: elt = sum(O.basis()); elt
1 + i + j + k + l + li + lj + lk
sage: elt.conjugate()
1 - i - j - k - l - li - lj - lk
```

```
>>> from sage.all import *
>>> O = OctonionAlgebra(QQ)
>>> elt = sum(O.basis()); elt
1 + i + j + k + l + li + lj + lk
>>> elt.conjugate()
1 - i - j - k - l - li - lj - lk
```

dict (copy=False)

Return `self` as a `dict` with keys being indices for the basis and the values being the corresponding nonzero coefficients.

INPUT:

- `copy` – ignored

EXAMPLES:

```
sage: O = OctonionAlgebra(QQ)
sage: x = O([2/7, 0, 0, 0, 2/3, 0, -5, 0])
sage: x.monomial_coefficients()
{0: 2/7, 4: 2/3, 6: -5}
```

```
>>> from sage.all import *
>>> O = OctonionAlgebra(QQ)
>>> x = O([Integer(2)/Integer(7), Integer(0), Integer(0), Integer(0),
Integer(2)/Integer(3), Integer(0), -Integer(5), Integer(0)])
>>> x.monomial_coefficients()
{0: 2/7, 4: 2/3, 6: -5}
```

imag_part()

Return the imaginary part of `self`.

OUTPUT: the imaginary part of `self` as an element in the octonion algebra

EXAMPLES:

```
sage: O = OctonionAlgebra(QQ)
sage: elt = sum(i * b for i, b in enumerate(O.basis(), start=2)); elt
2 + 3*i + 4*j + 5*k + 6*l + 7*li + 8*lj + 9*lk
sage: elt.imag_part()
3*i + 4*j + 5*k + 6*l + 7*li + 8*lj + 9*lk
```

```
>>> from sage.all import *
>>> O = OctonionAlgebra(QQ)
>>> elt = sum(i * b for i, b in enumerate(O.basis(), start=Integer(2))); elt
2 + 3*i + 4*j + 5*k + 6*l + 7*li + 8*lj + 9*lk
>>> elt.imag_part()
3*i + 4*j + 5*k + 6*l + 7*li + 8*lj + 9*lk
```

is_unit()

Return if `self` is a unit or not.

EXAMPLES:

```
sage: O = OctonionAlgebra(ZZ)
sage: x = O([1, 0, 1, 0, 0, 0, 0, 0])
sage: x.quadratic_form()
2
sage: x.is_unit()
False
sage: O([1, 0, -1, 0, 0, 0, 0, 0]).is_unit()
False
sage: x = O([1, 0, 0, 0, 1, 0, 0, 0])
sage: x.quadratic_form()
2
sage: x.is_unit()
False
sage: x = O([0, 0, 0, 0, 1, 0, 0, 0])
sage: x.quadratic_form()
1
sage: x.is_unit()
True
```

(continues on next page)

(continued from previous page)

```

True

sage: O = OctonionAlgebra(ZZ, -1, 1, 2)
sage: x = O([1, 0, 1, 0, 0, 0, 0, 0])
sage: x.quadratic_form()
0
sage: x.is_unit()
False
sage: O([1, 0, -1, 0, 0, 0, 0, 0]).is_unit()
False
sage: x = O([1, 0, 0, 0, 1, 0, 0, 0])
sage: x.quadratic_form()
-1
sage: x.is_unit()
True
sage: x = O([0, 0, 0, 0, 1, 0, 0, 0])
sage: x.quadratic_form()
-2
sage: x.is_unit()
False

```

```

>>> from sage.all import *
>>> O = OctonionAlgebra(ZZ)
>>> x = O([Integer(1), Integer(0), Integer(1), Integer(0), Integer(0),
↳ Integer(0), Integer(0), Integer(0)])
>>> x.quadratic_form()
2
>>> x.is_unit()
False
>>> O([Integer(1), Integer(0), -Integer(1), Integer(0), Integer(0),
↳ Integer(0), Integer(0), Integer(0)]).is_unit()
False
>>> x = O([Integer(1), Integer(0), Integer(0), Integer(0), Integer(1),
↳ Integer(0), Integer(0), Integer(0)])
>>> x.quadratic_form()
2
>>> x.is_unit()
False
>>> x = O([Integer(0), Integer(0), Integer(0), Integer(0), Integer(1),
↳ Integer(0), Integer(0), Integer(0)])
>>> x.quadratic_form()
1
>>> x.is_unit()
True

>>> O = OctonionAlgebra(ZZ, -Integer(1), Integer(1), Integer(2))
>>> x = O([Integer(1), Integer(0), Integer(1), Integer(0), Integer(0),
↳ Integer(0), Integer(0), Integer(0)])
>>> x.quadratic_form()
0
>>> x.is_unit()
False

```

(continues on next page)

(continued from previous page)

```

>>> O([Integer(1), Integer(0), -Integer(1), Integer(0), Integer(0),
↪Integer(0), Integer(0), Integer(0)]).is_unit()
False
>>> x = O([Integer(1), Integer(0), Integer(0), Integer(0), Integer(1),
↪Integer(0), Integer(0), Integer(0)])
>>> x.quadratic_form()
-1
>>> x.is_unit()
True
>>> x = O([Integer(0), Integer(0), Integer(0), Integer(0), Integer(1),
↪Integer(0), Integer(0), Integer(0)])
>>> x.quadratic_form()
-2
>>> x.is_unit()
False

```

monomial_coefficients (*copy=False*)

Return *self* as a dict with keys being indices for the basis and the values being the corresponding nonzero coefficients.

INPUT:

- *copy* – ignored

EXAMPLES:

```

sage: O = OctonionAlgebra(QQ)
sage: x = O([2/7, 0, 0, 0, 2/3, 0, -5, 0])
sage: x.monomial_coefficients()
{0: 2/7, 4: 2/3, 6: -5}

```

```

>>> from sage.all import *
>>> O = OctonionAlgebra(QQ)
>>> x = O([Integer(2)/Integer(7), Integer(0), Integer(0), Integer(0),
↪Integer(2)/Integer(3), Integer(0), -Integer(5), Integer(0)])
>>> x.monomial_coefficients()
{0: 2/7, 4: 2/3, 6: -5}

```

norm()

Return the norm of *self*.

The norm of an octonion x is $\|x\| = \sqrt{xx^*}$, where x^* is the *conjugate()* of x .

See also

This is the square root of *quadratic_form()*.

Warning

If any of the parameters $a, b, c \neq 0$, then this is not an actual norm.

EXAMPLES:

```

sage: O = OctonionAlgebra(QQ, 1, 3, 7)
sage: elt = sum(i * b for i, b in enumerate(O.basis(), start=2))
sage: elt.norm()
↪needs sage.symbolic
2*sqrt(-61)
sage: elt = sum(O.basis())
sage: elt.norm()
0

```

```

>>> from sage.all import *
>>> O = OctonionAlgebra(QQ, Integer(1), Integer(3), Integer(7))
>>> elt = sum(i * b for i, b in enumerate(O.basis(), start=Integer(2)))
>>> elt.norm()
↪needs sage.symbolic
2*sqrt(-61)
>>> elt = sum(O.basis())
>>> elt.norm()
0

```

quadratic_form()

Return the quadratic form of `self`.

The octonion algebra has a distinguished quadratic form given by $N(x) = xx^*$, where x^* is the *conjugate()* of x .

EXAMPLES:

```

sage: O = OctonionAlgebra(QQ, 1, 3, 7)
sage: elt = sum(O.basis())
sage: elt.quadratic_form()
0
sage: elt * elt.conjugate()
0

```

```

>>> from sage.all import *
>>> O = OctonionAlgebra(QQ, Integer(1), Integer(3), Integer(7))
>>> elt = sum(O.basis())
>>> elt.quadratic_form()
0
>>> elt * elt.conjugate()
0

```

real_part()

Return the real part of `self`.

OUTPUT: the real part of `self` as an element in the base ring

EXAMPLES:

```

sage: O = OctonionAlgebra(QQ)
sage: elt = sum(i * b for i, b in enumerate(O.basis(), start=2)); elt
2 + 3*i + 4*j + 5*k + 6*l + 7*li + 8*lj + 9*lk
sage: r = elt.real_part(); r
2

```

(continues on next page)

(continued from previous page)

```
sage: r.parent() is QQ
True
```

```
>>> from sage.all import *
>>> O = OctonionAlgebra(QQ)
>>> elt = sum(i * b for i, b in enumerate(O.basis(), start=Integer(2))); elt
2 + 3*i + 4*j + 5*k + 6*l + 7*li + 8*lj + 9*lk
>>> r = elt.real_part(); r
2
>>> r.parent() is QQ
True
```

vector (*new_base_ring=None*)

Return self as a vector in R^8 , where R is the base ring of self.

EXAMPLES:

```
sage: O = OctonionAlgebra(QQ)
sage: elt = sum(i * b for i, b in enumerate(O.basis(), start=2))
sage: elt.vector()
(2, 3, 4, 5, 6, 7, 8, 9)
```

```
>>> from sage.all import *
>>> O = OctonionAlgebra(QQ)
>>> elt = sum(i * b for i, b in enumerate(O.basis(), start=Integer(2)))
>>> elt.vector()
(2, 3, 4, 5, 6, 7, 8, 9)
```

9.2 Lie Algebras

9.2.1 Abelian Lie Algebras

AUTHORS:

- Travis Scrimshaw (2016-06-07): Initial version

class sage.algebras.lie_algebras.abelian.**AbelianLieAlgebra** (*R, names, index_set, category, **kws*)

Bases: *LieAlgebraWithStructureCoefficients*

An abelian Lie algebra.

A Lie algebra $\mathfrak{g}$ is abelian if $[x, y] = 0$ for all $x, y \in \mathfrak{g}$.

EXAMPLES:

```
sage: L.<x, y> = LieAlgebra(QQ, abelian=True)
sage: L.bracket(x, y)
0
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, abelian=True, names=('x', 'y')); (x, y) = L._first_
    ↪ngens(2)
```

(continues on next page)

(continued from previous page)

```
>>> L.bracket(x, y)
0
```

```
class Element
```

```
    Bases: Element
```

```
is_abelian()
```

```
    Return True since self is an abelian Lie algebra.
```

```
    EXAMPLES:
```

```
sage: L = LieAlgebra(QQ, 3, 'x', abelian=True)
sage: L.is_abelian()
True
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(3), 'x', abelian=True)
>>> L.is_abelian()
True
```

```
is_nilpotent()
```

```
    Return True since self is an abelian Lie algebra.
```

```
    EXAMPLES:
```

```
sage: L = LieAlgebra(QQ, 3, 'x', abelian=True)
sage: L.is_abelian()
True
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(3), 'x', abelian=True)
>>> L.is_abelian()
True
```

```
is_solvable()
```

```
    Return True since self is an abelian Lie algebra.
```

```
    EXAMPLES:
```

```
sage: L = LieAlgebra(QQ, 3, 'x', abelian=True)
sage: L.is_abelian()
True
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(3), 'x', abelian=True)
>>> L.is_abelian()
True
```

```
class sage.algebras.lie_algebras.abelian.InfiniteDimensionalAbelianLieAlgebra(R,
                                                                              index_set,
                                                                              prefix='L',
                                                                              **kws)
```

```
    Bases: InfinitelyGeneratedLieAlgebra, IndexedGenerators
```

An infinite dimensional abelian Lie algebra.

A Lie algebra $\mathfrak{g}$ is abelian if $[x, y] = 0$ for all $x, y \in \mathfrak{g}$.

class `Element`

Bases: `LieAlgebraElement`

dimension()

Return the dimension of `self`, which is ∞ .

EXAMPLES:

```
sage: L = lie_algebras.abelian(QQ, index_set=ZZ)
sage: L.dimension()
+Infinity
```

```
>>> from sage.all import *
>>> L = lie_algebras.abelian(QQ, index_set=ZZ)
>>> L.dimension()
+Infinity
```

is_abelian()

Return True since `self` is an abelian Lie algebra.

EXAMPLES:

```
sage: L = lie_algebras.abelian(QQ, index_set=ZZ)
sage: L.is_abelian()
True
```

```
>>> from sage.all import *
>>> L = lie_algebras.abelian(QQ, index_set=ZZ)
>>> L.is_abelian()
True
```

is_nilpotent()

Return True since `self` is an abelian Lie algebra.

EXAMPLES:

```
sage: L = lie_algebras.abelian(QQ, index_set=ZZ)
sage: L.is_abelian()
True
```

```
>>> from sage.all import *
>>> L = lie_algebras.abelian(QQ, index_set=ZZ)
>>> L.is_abelian()
True
```

is_solvable()

Return True since `self` is an abelian Lie algebra.

EXAMPLES:

```
sage: L = lie_algebras.abelian(QQ, index_set=ZZ)
sage: L.is_abelian()
True
```

```
>>> from sage.all import *
>>> L = lie_algebras.abelian(QQ, index_set=ZZ)
>>> L.is_abelian()
True
```

9.2.2 Affine Lie Algebras

AUTHORS:

- Travis Scrimshaw (2013-05-03): Initial version

```
class sage.algebras.lie_algebras.affine_lie_algebra.AffineLieAlgebra(g, cartan_type, names,
                                                                    kac_moody)
```

Bases: *FinitelyGeneratedLieAlgebra*

An (untwisted) affine Lie algebra.

Note that the derived subalgebra of the Kac-Moody algebra is the affine Lie algebra.

INPUT:

Can be one of the following:

- a base ring and an affine Cartan type: constructs the affine (Kac-Moody) Lie algebra of the classical Lie algebra in the bracket representation over the base ring
- a classical Lie algebra: constructs the corresponding affine (Kac-Moody) Lie algebra

There is the optional argument `kac_moody`, which can be set to `False` to obtain the affine Lie algebra instead of the affine Kac-Moody algebra.

See also

- *UntwistedAffineLieAlgebra*
- *TwistedAffineLieAlgebra*

REFERENCES:

- [Ka1990]

basis()

Return the basis of `self`.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['D', 4, 1])
sage: B = g.basis()
sage: al = RootSystem(['D', 4]).root_lattice().simple_roots()
sage: B[al[1]+al[2]+al[4], 4]
(E[alpha[1] + alpha[2] + alpha[4]])#t^4
sage: B[-al[1]-2*al[2]-al[3]-al[4], 2]
(E[-alpha[1] - 2*alpha[2] - alpha[3] - alpha[4]])#t^2
```

(continues on next page)

(continued from previous page)

```

sage: B[al[4],-2]
(E[alpha[4]])#t^-2
sage: B['c']
c
sage: B['d']
d

sage: g = LieAlgebra(QQ, cartan_type=['D', 4, 2], kac_moody=False)
sage: B = g.basis()
sage: it = iter(B)
sage: [next(it) for _ in range(3)]
[c, (E[alpha[1]])#t^0, (E[alpha[2]])#t^0]
sage: B['c']
c
sage: B['d']
0

```

```

>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['D', Integer(4), Integer(1)])
>>> B = g.basis()
>>> al = RootSystem(['D', Integer(4)]).root_lattice().simple_roots()
>>> B[al[Integer(1)]+al[Integer(2)]+al[Integer(4)], Integer(4)]
(E[alpha[1] + alpha[2] + alpha[4]])#t^4
>>> B[-al[Integer(1)]-Integer(2)*al[Integer(2)]-al[Integer(3)]-al[Integer(4)],
↪ Integer(2)]
(E[-alpha[1] - 2*alpha[2] - alpha[3] - alpha[4]])#t^2
>>> B[al[Integer(4)], -Integer(2)]
(E[alpha[4]])#t^-2
>>> B['c']
c
>>> B['d']
d

>>> g = LieAlgebra(QQ, cartan_type=['D', Integer(4), Integer(2)], kac_
↪ moody=False)
>>> B = g.basis()
>>> it = iter(B)
>>> [next(it) for _ in range(Integer(3))]
[c, (E[alpha[1]])#t^0, (E[alpha[2]])#t^0]
>>> B['c']
c
>>> B['d']
0

```

c()Return the canonical central element c of self.

EXAMPLES:

```

sage: g = LieAlgebra(QQ, cartan_type=['A', 3, 1])
sage: g.c()
c

```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['A', Integer(3), Integer(1)])
>>> g.c()
c
```

cartan_type()

Return the Cartan type of self.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['C', 3, 1])
sage: g.cartan_type()
['C', 3, 1]
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['C', Integer(3), Integer(1)])
>>> g.cartan_type()
['C', 3, 1]
```

classical()

Return the classical Lie algebra of self.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['F', 4, 1])
sage: g.classical()
Lie algebra of ['F', 4] in the Chevalley basis

sage: so5 = lie_algebras.so(QQ, 5, 'matrix')
sage: A = so5.affine()
sage: A.classical() == so5
True
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['F', Integer(4), Integer(1)])
>>> g.classical()
Lie algebra of ['F', 4] in the Chevalley basis

>>> so5 = lie_algebras.so(QQ, Integer(5), 'matrix')
>>> A = so5.affine()
>>> A.classical() == so5
True
```

d()

Return the canonical derivation d of self.

If self is the affine Lie algebra, then this returns 0.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['A', 3, 1])
sage: g.d()
d
sage: D = g.derived_subalgebra()
```

(continues on next page)

(continued from previous page)

```
sage: D.d()
0
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['A', Integer(3), Integer(1)])
>>> g.d()
d
>>> D = g.derived_subalgebra()
>>> D.d()
0
```

derived_series()

Return the derived series of self.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['B', 3, 1])
sage: g.derived_series()
[Affine Kac-Moody algebra of ['B', 3] in the Chevalley basis,
 Affine Lie algebra of ['B', 3] in the Chevalley basis]
sage: g.lower_central_series()
[Affine Kac-Moody algebra of ['B', 3] in the Chevalley basis,
 Affine Lie algebra of ['B', 3] in the Chevalley basis]

sage: D = g.derived_subalgebra()
sage: D.derived_series()
[Affine Lie algebra of ['B', 3] in the Chevalley basis]
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['B', Integer(3), Integer(1)])
>>> g.derived_series()
[Affine Kac-Moody algebra of ['B', 3] in the Chevalley basis,
 Affine Lie algebra of ['B', 3] in the Chevalley basis]
>>> g.lower_central_series()
[Affine Kac-Moody algebra of ['B', 3] in the Chevalley basis,
 Affine Lie algebra of ['B', 3] in the Chevalley basis]

>>> D = g.derived_subalgebra()
>>> D.derived_series()
[Affine Lie algebra of ['B', 3] in the Chevalley basis]
```

e (i=None)

Return the generators e of self.

INPUT:

- i – (optional) if specified, return just the generator e_i

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['B', 3, 1])
sage: list(g.e())
[(E[-alpha[1] - 2*alpha[2] - 2*alpha[3]])#t^1,
```

(continues on next page)

(continued from previous page)

```
(E[alpha[1]])#t^0, (E[alpha[2]])#t^0, (E[alpha[3]])#t^0]
sage: g.e(2)
(E[alpha[2]])#t^0
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['B', Integer(3), Integer(1)])
>>> list(g.e())
[(E[-alpha[1] - 2*alpha[2] - 2*alpha[3]])#t^1,
 (E[alpha[1]])#t^0, (E[alpha[2]])#t^0, (E[alpha[3]])#t^0]
>>> g.e(Integer(2))
(E[alpha[2]])#t^0
```

f (*i=None*)Return the generators f of self.

INPUT:

- *i* – (optional) if specified, return just the generator f_i

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['A', 5, 2])
sage: list(g.f())
[(E[alpha[1] + 2*alpha[2] + alpha[3]])#t^-1,
 (E[-alpha[1]])#t^0, (E[-alpha[2]])#t^0, (E[-alpha[3]])#t^0]
sage: g.f(2)
(E[-alpha[2]])#t^0
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['A', Integer(5), Integer(2)])
>>> list(g.f())
[(E[alpha[1] + 2*alpha[2] + alpha[3]])#t^-1,
 (E[-alpha[1]])#t^0, (E[-alpha[2]])#t^0, (E[-alpha[3]])#t^0]
>>> g.f(Integer(2))
(E[-alpha[2]])#t^0
```

is_nilpotent ()

Return False as self is semisimple.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['B', 3, 1])
sage: g.is_nilpotent()
False
sage: g.is_solvable()
False
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['B', Integer(3), Integer(1)])
>>> g.is_nilpotent()
False
>>> g.is_solvable()
False
```

is_solvable()

Return False as self is semisimple.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['B',3,1])
sage: g.is_nilpotent()
False
sage: g.is_solvable()
False
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['B',Integer(3),Integer(1)])
>>> g.is_nilpotent()
False
>>> g.is_solvable()
False
```

lie_algebra_generators()

Return the Lie algebra generators of self.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['A',1,1])
sage: list(g.lie_algebra_generators())
[(E[alpha[1]])#t^0,
 (E[-alpha[1]])#t^0,
 (h1)#t^0,
 (E[-alpha[1]])#t^1,
 (E[alpha[1]])#t^-1,
 c,
 d]
```

```
sage: L = LieAlgebra(QQ, cartan_type=['A',5,2])
sage: list(L.lie_algebra_generators())
[(E[alpha[1]])#t^0,
 (E[alpha[2]])#t^0,
 (E[alpha[3]])#t^0,
 (E[-alpha[1]])#t^0,
 (E[-alpha[2]])#t^0,
 (E[-alpha[3]])#t^0,
 (h1)#t^0,
 (h2)#t^0,
 (h3)#t^0,
 (E[-alpha[1] - 2*alpha[2] - alpha[3]])#t^1,
 (E[alpha[1] + 2*alpha[2] + alpha[3]])#t^-1,
 c,
 d]
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['A',Integer(1),Integer(1)])
>>> list(g.lie_algebra_generators())
[(E[alpha[1]])#t^0,
 (E[-alpha[1]])#t^0,
```

(continues on next page)

(continued from previous page)

```

(h1)#t^0,
(E[-alpha[1]])#t^1,
(E[alpha[1]])#t^-1,
c,
d]

>>> L = LieAlgebra(QQ, cartan_type=['A',Integer(5),Integer(2)])
>>> list(L.lie_algebra_generators())
[(E[alpha[1]])#t^0,
 (E[alpha[2]])#t^0,
 (E[alpha[3]])#t^0,
 (E[-alpha[1]])#t^0,
 (E[-alpha[2]])#t^0,
 (E[-alpha[3]])#t^0,
 (h1)#t^0,
 (h2)#t^0,
 (h3)#t^0,
 (E[-alpha[1] - 2*alpha[2] - alpha[3]])#t^1,
 (E[alpha[1] + 2*alpha[2] + alpha[3]])#t^-1,
 c,
 d]

```

lower_central_series()

Return the derived series of self.

EXAMPLES:

```

sage: g = LieAlgebra(QQ, cartan_type=['B',3,1])
sage: g.derived_series()
[Affine Kac-Moody algebra of ['B', 3] in the Chevalley basis,
 Affine Lie algebra of ['B', 3] in the Chevalley basis]
sage: g.lower_central_series()
[Affine Kac-Moody algebra of ['B', 3] in the Chevalley basis,
 Affine Lie algebra of ['B', 3] in the Chevalley basis]

sage: D = g.derived_subalgebra()
sage: D.derived_series()
[Affine Lie algebra of ['B', 3] in the Chevalley basis]

```

```

>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['B',Integer(3),Integer(1)])
>>> g.derived_series()
[Affine Kac-Moody algebra of ['B', 3] in the Chevalley basis,
 Affine Lie algebra of ['B', 3] in the Chevalley basis]
>>> g.lower_central_series()
[Affine Kac-Moody algebra of ['B', 3] in the Chevalley basis,
 Affine Lie algebra of ['B', 3] in the Chevalley basis]

>>> D = g.derived_subalgebra()
>>> D.derived_series()
[Affine Lie algebra of ['B', 3] in the Chevalley basis]

```

monomial(m)

Construct the monomial indexed by `m`.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['B',4,1])
sage: al = RootSystem(['B',4]).root_lattice().simple_roots()
sage: g.monomial((al[1]+al[2]+al[3],4))
(E[alpha[1] + alpha[2] + alpha[3]])#t^4
sage: g.monomial((-al[1]-al[2]-2*al[3]-2*al[4],2))
(E[-alpha[1] - alpha[2] - 2*alpha[3] - 2*alpha[4]])#t^2
sage: g.monomial((al[4],-2))
(E[alpha[4]])#t^-2
sage: g.monomial('c')
c
sage: g.monomial('d')
d
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['B',Integer(4),Integer(1)])
>>> al = RootSystem(['B',Integer(4)]).root_lattice().simple_roots()
>>> g.monomial((al[Integer(1)]+al[Integer(2)]+al[Integer(3)],Integer(4)))
(E[alpha[1] + alpha[2] + alpha[3]])#t^4
>>> g.monomial((-al[Integer(1)]-al[Integer(2)]-Integer(2)*al[Integer(3)]-
↳ Integer(2)*al[Integer(4)],Integer(2)))
(E[-alpha[1] - alpha[2] - 2*alpha[3] - 2*alpha[4]])#t^2
>>> g.monomial((al[Integer(4)],-Integer(2)))
(E[alpha[4]])#t^-2
>>> g.monomial('c')
c
>>> g.monomial('d')
d
```

zero()

Return the element 0.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['F',4,1])
sage: g.zero()
0
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['F',Integer(4),Integer(1)])
>>> g.zero()
0
```

class `sage.algebras.lie_algebras.affine_lie_algebra.TwistedAffineIndices` (*cartan_type*)

Bases: `UniqueRepresentation`, `Set_generic`

The indices for the basis of a twisted affine Lie algebra.

INPUT:

- `cartan_type` – the Cartan type of twisted affine type Lie algebra

EXAMPLES:

```

sage: from sage.algebras.lie_algebras.affine_lie_algebra import _
      ↪ TwistedAffineIndices
sage: I = TwistedAffineIndices(['A', 3, 2])
sage: it = iter(I)
sage: [next(it) for _ in range(20)]
[(alpha[1], 0), (alpha[2], 0), (alpha[1] + alpha[2], 0),
 (2*alpha[1] + alpha[2], 0), (-alpha[1], 0), (-alpha[2], 0),
 (-alpha[1] - alpha[2], 0), (-2*alpha[1] - alpha[2], 0),
 (alphacheck[1], 0), (alphacheck[2], 0), (alpha[1], 1),
 (alpha[1] + alpha[2], 1), (-alpha[1], 1), (-alpha[1] - alpha[2], 1),
 (alphacheck[1], 1), (alpha[1], -1), (alpha[1] + alpha[2], -1),
 (-alpha[1], -1), (-alpha[1] - alpha[2], -1), (alphacheck[1], -1)]

sage: I = TwistedAffineIndices(['A', 4, 2])
sage: it = iter(I)
sage: [next(it) for _ in range(20)]
[(alpha[0], 0), (alpha[1], 0), (alpha[0] + alpha[1], 0),
 (2*alpha[0] + alpha[1], 0), (-alpha[0], 0), (-alpha[1], 0),
 (-alpha[0] - alpha[1], 0), (-2*alpha[0] - alpha[1], 0),
 (alphacheck[0], 0), (alphacheck[1], 0), (alpha[0], 1), (alpha[1], 1),
 (alpha[0] + alpha[1], 1), (2*alpha[0] + alpha[1], 1), (-alpha[0], 1),
 (-alpha[1], 1), (-alpha[0] - alpha[1], 1), (-2*alpha[0] - alpha[1], 1),
 (2*alpha[0], 1), (2*alpha[0] + 2*alpha[1], 1)]

sage: I = TwistedAffineIndices(['A', 2, 2])
sage: it = iter(I)
sage: [next(it) for _ in range(10)]
[(alpha[0], 0), (-alpha[0], 0), (alphacheck[0], 0), (alpha[0], 1),
 (-alpha[0], 1), (2*alpha[0], 1), (-2*alpha[0], 1),
 (alphacheck[0], 1), (alpha[0], -1), (-alpha[0], -1)]

```

```

>>> from sage.all import *
>>> from sage.algebras.lie_algebras.affine_lie_algebra import TwistedAffineIndices
>>> I = TwistedAffineIndices(['A', Integer(3), Integer(2)])
>>> it = iter(I)
>>> [next(it) for _ in range(Integer(20))]
[(alpha[1], 0), (alpha[2], 0), (alpha[1] + alpha[2], 0),
 (2*alpha[1] + alpha[2], 0), (-alpha[1], 0), (-alpha[2], 0),
 (-alpha[1] - alpha[2], 0), (-2*alpha[1] - alpha[2], 0),
 (alphacheck[1], 0), (alphacheck[2], 0), (alpha[1], 1),
 (alpha[1] + alpha[2], 1), (-alpha[1], 1), (-alpha[1] - alpha[2], 1),
 (alphacheck[1], 1), (alpha[1], -1), (alpha[1] + alpha[2], -1),
 (-alpha[1], -1), (-alpha[1] - alpha[2], -1), (alphacheck[1], -1)]

>>> I = TwistedAffineIndices(['A', Integer(4), Integer(2)])
>>> it = iter(I)
>>> [next(it) for _ in range(Integer(20))]
[(alpha[0], 0), (alpha[1], 0), (alpha[0] + alpha[1], 0),
 (2*alpha[0] + alpha[1], 0), (-alpha[0], 0), (-alpha[1], 0),
 (-alpha[0] - alpha[1], 0), (-2*alpha[0] - alpha[1], 0),
 (alphacheck[0], 0), (alphacheck[1], 0), (alpha[0], 1), (alpha[1], 1),

```

(continues on next page)

(continued from previous page)

```

(alpha[0] + alpha[1], 1), (2*alpha[0] + alpha[1], 1), (-alpha[0], 1),
(-alpha[1], 1), (-alpha[0] - alpha[1], 1), (-2*alpha[0] - alpha[1], 1),
(2*alpha[0], 1), (2*alpha[0] + 2*alpha[1], 1)]

>>> I = TwistedAffineIndices(['A', Integer(2), Integer(2)])
>>> it = iter(I)
>>> [next(it) for _ in range(Integer(10))]
[(alpha[0], 0), (-alpha[0], 0), (alphacheck[0], 0), (alpha[0], 1),
(-alpha[0], 1), (2*alpha[0], 1), (-2*alpha[0], 1),
(alphacheck[0], 1), (alpha[0], -1), (-alpha[0], -1)]

```

```

class sage.algebras.lie_algebras.affine_lie_algebra.TwistedAffineLieAlgebra (R,
                                                                    cartan_type,
                                                                    kac_moody)

```

Bases: *AffineLieAlgebra*

A twisted affine Lie algebra.

A twisted affine Lie algebra is an affine Lie algebra for type $X_N^{(r)}$ with $r > 1$. We realize this inside an untwisted affine Kac–Moody Lie algebra following Chapter 8 of [Ka1990].

Let $\bar{\mathfrak{g}}$ be the classical Lie algebra by taking the index set $I \setminus \{\epsilon\}$, where $\epsilon = 0$ unless $\epsilon = n$ for $X_N^{(r)} = A_{2n}^{(2)}$, for the twisted affine Lie algebra $\tilde{\mathfrak{g}}$. Let $\mathfrak{g}$ be the basic Lie algebra of type X_N . We realize $\bar{\mathfrak{g}}$ as the fixed-point subalgebra $\mathfrak{g}^{(0)}$ of $\mathfrak{g}$ under the order r diagram automorphism μ . This naturally acts on the ζ_r (a primitive r -th root of unity) eigenspace $\mathfrak{g}^{(1)}$ of μ , which is the highest weight representation corresponding to the small adjoint (where the weight spaces are the short roots of $\bar{\mathfrak{g}}$). The *twisted affine (Kac–Moody) Lie algebra* $\hat{\mathfrak{g}}$ is constructed as the subalgebra of $X_N^{(1)}$ given by

$$\sum_{i \in \mathbb{Z}} \mathfrak{g}^{(i \bmod 2)} \otimes t^i \oplus Rc \oplus Rd,$$

where R is the base ring.

We encode our basis by using the classical Lie algebra except for type $A_{2n}^{(2)}$. For type $A_{2n}^{(2)}$, the fixed-point algebra $\mathfrak{g}^{(0)}$ is of type B_n using the index set $\{0, \dots, n-1\}$. For $\mathfrak{g}^{(1)}$, we identify the weights in this representation with the roots of type B_n and the double all of its short roots.

```
class Element
```

Bases: *UntwistedAffineLieAlgebraElement*

```
ambient()
```

Return the ambient untwisted affine Lie algebra of *self*.

EXAMPLES:

```

sage: g = LieAlgebra(QQ, cartan_type=['A', 5, 2])
sage: g.ambient()
Affine Kac–Moody algebra of ['A', 5] in the Chevalley basis

```

```

>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['A', Integer(5), Integer(2)])
>>> g.ambient()
Affine Kac–Moody algebra of ['A', 5] in the Chevalley basis

```

derived_subalgebra()

Return the derived subalgebra of self.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['A', 5, 2])
sage: g
Twisted affine Kac-Moody algebra of type ['B', 3, 1]^* over Rational Field
sage: D = g.derived_subalgebra(); D
Twisted affine Lie algebra of type ['B', 3, 1]^* over Rational Field
sage: D.derived_subalgebra() == D
True
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['A', Integer(5), Integer(2)])
>>> g
Twisted affine Kac-Moody algebra of type ['B', 3, 1]^* over Rational Field
>>> D = g.derived_subalgebra(); D
Twisted affine Lie algebra of type ['B', 3, 1]^* over Rational Field
>>> D.derived_subalgebra() == D
True
```

retract(x)

Retract the element x from the ambient untwisted affine Lie algebra into self.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['A', 5, 2])
sage: it = iter(g.basis())
sage: elts = [next(it) for _ in range(20)]
sage: elts
[c,
 d,
 (E[alpha[1]])#t^0,
 (E[alpha[2]])#t^0,
 (E[alpha[3]])#t^0,
 (E[alpha[1] + alpha[2]])#t^0,
 (E[alpha[2] + alpha[3]])#t^0,
 (E[2*alpha[2] + alpha[3]])#t^0,
 (E[alpha[1] + alpha[2] + alpha[3]])#t^0,
 (E[2*alpha[1] + 2*alpha[2] + alpha[3]])#t^0,
 (E[alpha[1] + 2*alpha[2] + alpha[3]])#t^0,
 (E[-alpha[1]])#t^0,
 (E[-alpha[2]])#t^0,
 (E[-alpha[3]])#t^0,
 (E[-alpha[1] - alpha[2]])#t^0,
 (E[-alpha[2] - alpha[3]])#t^0,
 (E[-2*alpha[2] - alpha[3]])#t^0,
 (E[-alpha[1] - alpha[2] - alpha[3]])#t^0,
 (E[-2*alpha[1] - 2*alpha[2] - alpha[3]])#t^0,
 (E[-alpha[1] - 2*alpha[2] - alpha[3]])#t^0]
sage: all(g.retract(g.to_ambient(x)) == x for x in elts)
True
```



```

>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['A', Integer(5), Integer(2)])
>>> it = iter(g.basis())
>>> elts = [next(it) for _ in range(Integer(20))]
>>> elts
[c,
 d,
 (E[alpha[1]])#t^0,
 (E[alpha[2]])#t^0,
 (E[alpha[3]])#t^0,
 (E[alpha[1] + alpha[2]])#t^0,
 (E[alpha[2] + alpha[3]])#t^0,
 (E[2*alpha[2] + alpha[3]])#t^0,
 (E[alpha[1] + alpha[2] + alpha[3]])#t^0,
 (E[2*alpha[1] + 2*alpha[2] + alpha[3]])#t^0,
 (E[alpha[1] + 2*alpha[2] + alpha[3]])#t^0,
 (E[-alpha[1]])#t^0,
 (E[-alpha[2]])#t^0,
 (E[-alpha[3]])#t^0,
 (E[-alpha[1] - alpha[2]])#t^0,
 (E[-alpha[2] - alpha[3]])#t^0,
 (E[-2*alpha[2] - alpha[3]])#t^0,
 (E[-alpha[1] - alpha[2] - alpha[3]])#t^0,
 (E[-2*alpha[1] - 2*alpha[2] - alpha[3]])#t^0,
 (E[-alpha[1] - 2*alpha[2] - alpha[3]])#t^0]
>>> all(g.retract(g.to_ambient(x)) == x for x in elts)
True

```

to_ambient()

Lift the element x from the ambient untwisted affine Lie algebra into self.

EXAMPLES:

```

sage: g = LieAlgebra(QQ, cartan_type=['A', 5, 2])
sage: g.to_ambient
Generic morphism:
  From: Twisted affine Kac-Moody algebra of type ['B', 3, 1]^* over Rational_
↪Field
  To:   Affine Kac-Moody algebra of ['A', 5] in the Chevalley basis

```

```

>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['A', Integer(5), Integer(2)])
>>> g.to_ambient
Generic morphism:
  From: Twisted affine Kac-Moody algebra of type ['B', 3, 1]^* over Rational_
↪Field
  To:   Affine Kac-Moody algebra of ['A', 5] in the Chevalley basis

```

class sage.algebras.lie_algebras.affine_lie_algebra.UntwistedAffineLieAlgebra(g , kac_moody)

Bases: *AffineLieAlgebra*

An untwisted affine Lie algebra.

Let R be a ring. Given a finite-dimensional simple Lie algebra $\mathfrak{g}$ over R , the affine Lie algebra $\widehat{\mathfrak{g}}$ associated to $\mathfrak{g}$ is defined as

$$\widehat{\mathfrak{g}} = (\mathfrak{g} \otimes R[t, t^{-1}]) \oplus Rc,$$

where c is the canonical central element and $R[t, t^{-1}]$ is the Laurent polynomial ring over R . The Lie bracket is defined as

$$[x \otimes t^m + \lambda c, y \otimes t^n + \mu c] = [x, y] \otimes t^{m+n} + m\delta_{m, -n}(x|y)c,$$

where $(x|y)$ is the Killing form on $\mathfrak{g}$.

There is a canonical derivation d on $\widehat{\mathfrak{g}}$ that is defined by

$$d(x \otimes t^m + \lambda c) = a \otimes mt^m,$$

or equivalently by $d = t \frac{d}{dt}$.

The affine Kac-Moody algebra $\widehat{\mathfrak{g}}$ is formed by adjoining the derivation d such that

$$\widehat{\mathfrak{g}} = (\mathfrak{g} \otimes R[t, t^{-1}]) \oplus Rc \oplus Rd.$$

Specifically, the bracket on $\widehat{\mathfrak{g}}$ is defined as

$$[t^m \otimes x \oplus \lambda c \oplus \mu d, t^n \otimes y \oplus \lambda_1 c \oplus \mu_1 d] = (t^{m+n}[x, y] + \mu n t^n \otimes y - \mu_1 m t^m \otimes x) \oplus m\delta_{m, -n}(x|y)c.$$

EXAMPLES:

We begin by constructing an affine Kac-Moody algebra of type $G_2^{(1)}$ from the classical Lie algebra of type G_2 :

```
sage: g = LieAlgebra(QQ, cartan_type=['G', 2])
sage: A = g.affine()
sage: A
Affine Kac-Moody algebra of ['G', 2] in the Chevalley basis
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['G', Integer(2)])
>>> A = g.affine()
>>> A
Affine Kac-Moody algebra of ['G', 2] in the Chevalley basis
```

Next, we construct the generators and perform some computations:

```
sage: A.inject_variables()
Defining e1, e2, f1, f2, h1, h2, e0, f0, c, d
sage: e1.bracket(f1)
(h1)#t^0
sage: e0.bracket(f0)
(-h1 - 2*h2)#t^0 + 8*c
sage: e0.bracket(f1)
0
sage: A[d, f0]
(-E[3*alpha[1] + 2*alpha[2]])#t^-1
sage: A([[e0, e2], [[e1, e2], [e0, [e1, e2]]], e1]])
(-6*E[-3*alpha[1] - alpha[2]])#t^2
sage: f0.bracket(f1)
```

(continues on next page)

(continued from previous page)

```

0
sage: f0.bracket(f2)
(E[3*alpha[1] + alpha[2]])#t^-1
sage: A[h1+3*h2, A[[f0, f2], f1], [f1,f2]] + f1 - f1
(2*E[alpha[1]])#t^-1

```

```

>>> from sage.all import *
>>> A.inject_variables()
Defining e1, e2, f1, f2, h1, h2, e0, f0, c, d
>>> e1.bracket(f1)
(h1)#t^0
>>> e0.bracket(f0)
(-h1 - 2*h2)#t^0 + 8*c
>>> e0.bracket(f1)
0
>>> A[d, f0]
(-E[3*alpha[1] + 2*alpha[2]])#t^-1
>>> A([e0, e2], [[e1, e2], [e0, [e1, e2]]], e1))
(-6*E[-3*alpha[1] - alpha[2]])#t^2
>>> f0.bracket(f1)
0
>>> f0.bracket(f2)
(E[3*alpha[1] + alpha[2]])#t^-1
>>> A[h1+Integer(3)*h2, A[[f0, f2], f1], [f1,f2]] + f1 - f1
(2*E[alpha[1]])#t^-1

```

We can construct its derived subalgebra, the affine Lie algebra of type $G_2^{(1)}$. In this case, there is no canonical derivation, so the generator d is 0:

```

sage: D = A.derived_subalgebra()
sage: D.d()
0

```

```

>>> from sage.all import *
>>> D = A.derived_subalgebra()
>>> D.d()
0

```

Element

alias of `UntwistedAffineLieAlgebraElement`

derived_subalgebra()

Return the derived subalgebra of `self`.

EXAMPLES:

```

sage: g = LieAlgebra(QQ, cartan_type=['B',3,1])
sage: g
Affine Kac-Moody algebra of ['B', 3] in the Chevalley basis
sage: D = g.derived_subalgebra(); D
Affine Lie algebra of ['B', 3] in the Chevalley basis
sage: D.derived_subalgebra() == D
True

```

```

>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['B', Integer(3), Integer(1)])
>>> g
Affine Kac-Moody algebra of ['B', 3] in the Chevalley basis
>>> D = g.derived_subalgebra(); D
Affine Lie algebra of ['B', 3] in the Chevalley basis
>>> D.derived_subalgebra() == D
True

```

9.2.3 The Baker-Campbell-Hausdorff formula

AUTHORS:

- Eero Hakavuori (2018-09-23): initial version

`sage.algebras.lie_algebras.bch.bch_iterator` ($X=None, Y=None$)

A generator function which returns successive terms of the Baker-Campbell-Hausdorff formula.

INPUT:

- X – (optional) an element of a Lie algebra
- Y – (optional) an element of a Lie algebra

The BCH formula is an expression for $\log(\exp(X)\exp(Y))$ as a sum of Lie brackets of X and Y with rational coefficients. In arbitrary Lie algebras, the infinite sum is only guaranteed to converge for X and Y close to zero.

If the elements X and Y are not given, then the iterator will return successive terms of the abstract BCH formula, i.e., the BCH formula for the generators of the free Lie algebra on 2 generators.

If the Lie algebra containing X and Y is not nilpotent, the iterator will output infinitely many elements. If the Lie algebra is nilpotent, the number of elements outputted is equal to the nilpotency step.

EXAMPLES:

The terms of the abstract BCH formula up to fifth order brackets:

```

sage: from sage.algebras.lie_algebras.bch import bch_iterator
sage: bch = bch_iterator()
sage: next(bch)
X + Y
sage: next(bch)
1/2*[X, Y]
sage: next(bch)
1/12*[X, [X, Y]] + 1/12*[[X, Y], Y]
sage: next(bch)
1/24*[X, [[X, Y], Y]]
sage: next(bch)
-1/720*[X, [X, [X, [X, Y]]]] + 1/180*[X, [X, [[X, Y], Y]]]
+ 1/360*[[X, [X, Y]], [X, Y]] + 1/180*[X, [[X, Y], Y], Y]]
+ 1/120*[[X, Y], [[X, Y], Y]] - 1/720*[[[X, Y], Y], Y], Y]

```

```

>>> from sage.all import *
>>> from sage.algebras.lie_algebras.bch import bch_iterator
>>> bch = bch_iterator()
>>> next(bch)
X + Y

```

(continues on next page)

(continued from previous page)

```

>>> next(bch)
1/2*[X, Y]
>>> next(bch)
1/12*[X, [X, Y]] + 1/12*[X, Y], Y]
>>> next(bch)
1/24*[X, [[X, Y], Y]]
>>> next(bch)
-1/720*[X, [X, [X, [X, Y]]]] + 1/180*[X, [X, [[X, Y], Y]]]
+ 1/360*[X, [X, Y]], [X, Y]] + 1/180*[X, [[X, Y], Y], Y]]
+ 1/120*[X, Y], [[X, Y], Y]] - 1/720*[[[X, Y], Y], Y], Y]

```

For nilpotent Lie algebras the BCH formula only has finitely many terms:

```

sage: L = LieAlgebra(QQ, 2, step=3)
sage: L.inject_variables()
Defining X_1, X_2, X_12, X_112, X_122
sage: [Z for Z in bch_iterator(X_1, X_2)]
[X_1 + X_2, 1/2*X_12, 1/12*X_112 + 1/12*X_122]
sage: [Z for Z in bch_iterator(X_1 + X_2, X_12)]
[X_1 + X_2 + X_12, 1/2*X_112 - 1/2*X_122, 0]

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(2), step=Integer(3))
>>> L.inject_variables()
Defining X_1, X_2, X_12, X_112, X_122
>>> [Z for Z in bch_iterator(X_1, X_2)]
[X_1 + X_2, 1/2*X_12, 1/12*X_112 + 1/12*X_122]
>>> [Z for Z in bch_iterator(X_1 + X_2, X_12)]
[X_1 + X_2 + X_12, 1/2*X_112 - 1/2*X_122, 0]

```

The elements x and y don't need to be elements of the same Lie algebra if there is a coercion from one to the other:

```

sage: L = LieAlgebra(QQ, 3, step=2)
sage: L.inject_variables()
Defining X_1, X_2, X_3, X_12, X_13, X_23
sage: S = L.subalgebra(X_1, X_2)
sage: bch1 = [Z for Z in bch_iterator(S(X_1), S(X_2))]; bch1
[X_1 + X_2, 1/2*X_12]
sage: bch1[0].parent() == S
True
sage: bch2 = [Z for Z in bch_iterator(S(X_1), X_3)]; bch2
[X_1 + X_3, 1/2*X_13]
sage: bch2[0].parent() == L
True

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(3), step=Integer(2))
>>> L.inject_variables()
Defining X_1, X_2, X_3, X_12, X_13, X_23
>>> S = L.subalgebra(X_1, X_2)
>>> bch1 = [Z for Z in bch_iterator(S(X_1), S(X_2))]; bch1
[X_1 + X_2, 1/2*X_12]

```

(continues on next page)

(continued from previous page)

```
>>> bch1[Integer(0)].parent() == S
True
>>> bch2 = [Z for Z in bch_iterator(S(X_1), X_3)]; bch2
[X_1 + X_3, 1/2*X_13]
>>> bch2[Integer(0)].parent() == L
True
```

The BCH formula requires a coercion from the rationals:

```
sage: L.<X,Y,Z> = LieAlgebra(ZZ, 2, step=2)
sage: bch = bch_iterator(X, Y); next(bch)
Traceback (most recent call last):
...
TypeError: the BCH formula is not well defined since Integer Ring has no coercion
↳from Rational Field
```

```
>>> from sage.all import *
>>> L = LieAlgebra(ZZ, Integer(2), step=Integer(2), names=('X', 'Y', 'Z')); (X,
↳Y, Z) = L._first_ngens(3)
>>> bch = bch_iterator(X, Y); next(bch)
Traceback (most recent call last):
...
TypeError: the BCH formula is not well defined since Integer Ring has no coercion
↳from Rational Field
```

ALGORITHM:

The BCH formula $\log(\exp(X)\exp(Y)) = \sum_k Z_k$ is computed starting from $Z_1 = X + Y$, by the recursion

$$(m+1)Z_{m+1} = \frac{1}{2}[X - Y, Z_m] + \sum_{2 \leq 2p \leq m} \frac{B_{2p}}{(2p)!} \sum_{k_1 + \dots + k_{2p} = m} [Z_{k_1}, [\dots [Z_{k_{2p}}, X + Y] \dots]],$$

where B_{2p} are the Bernoulli numbers, see Lemma 2.15.3. in [Var1984].

Warning

The time needed to compute each successive term increases exponentially. For example on one machine iterating through $Z_{11}, \dots, Z_{18}$ for a free Lie algebra, computing each successive term took 4-5 times longer, going from 0.1s for Z_{11} to 21 minutes for Z_{18} .

9.2.4 BGG Resolutions

AUTHORS:

- Travis Scrimshaw (2024-01-07): Initial version

```
class sage.algebras.lie_algebras.bgg_resolution.BGGResolution(L)
```

Bases: UniqueRepresentation, ChainComplex_class

The BGG resolution of a simple module.

We realize the BGG resolution as a chain complex, where the (-1) -th factor corresponds to the finite dimensional

simple module L_λ and the i -th factor ($i \geq 0$) corresponds to

$$M_i := \bigoplus_{\substack{w \in W \\ \ell(w)=i}} M_{w\lambda}.$$

Since the morphisms can be defined in terms of the image of the highest weight vectors, we only encode this information as a (finite) chain complex. We do not include the final natural projection map $p : M_\lambda \rightarrow L_\lambda$ since the highest weight vector of weight λ only occurs in M_λ and L_λ .

INPUT:

- L – a simple module

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['A', 2])
sage: La = g.cartan_type().root_system().weight_lattice().fundamental_weights()
sage: L = g.simple_module(La[1] + 4*La[2])
sage: res = L.bgg_resolution()
sage: ascii_art(res)

          [ 1 -1]      [1]
        [1 1]      [-1 1]      [1]
0 <-- C_0 <----- C_1 <----- C_2 <---- C_3 <-- 0

sage: g = LieAlgebra(QQ, cartan_type=['D', 4])
sage: La = g.cartan_type().root_system().weight_lattice().fundamental_weights()
sage: L = g.simple_module(La[1] + La[2] + 3*La[3])
sage: res = L.bgg_resolution()
sage: w0 = WeylGroup(g.cartan_type(), prefix='s').long_element()
sage: all(res.differential(i) * res.differential(i+1) == 0
....:     for i in range(w0.length()))
True
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['A', Integer(2)])
>>> La = g.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> L = g.simple_module(La[Integer(1)] + Integer(4)*La[Integer(2)])
>>> res = L.bgg_resolution()
>>> ascii_art(res)

          [ 1 -1]      [1]
        [1 1]      [-1 1]      [1]
0 <-- C_0 <----- C_1 <----- C_2 <---- C_3 <-- 0

>>> g = LieAlgebra(QQ, cartan_type=['D', Integer(4)])
>>> La = g.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> L = g.simple_module(La[Integer(1)] + La[Integer(2)] +
↳ Integer(3)*La[Integer(3)])
>>> res = L.bgg_resolution()
>>> w0 = WeylGroup(g.cartan_type(), prefix='s').long_element()
>>> all(res.differential(i) * res.differential(i+Integer(1)) == Integer(0)
...     for i in range(w0.length()))
True
```

module_order(i)

Return a tuple of Weyl group elements of length i determining the ordering of the direct sum defining the differential matrix.

EXAMPLES:

```

sage: g = LieAlgebra(QQ, cartan_type=['G', 2])
sage: La = g.cartan_type().root_system().weight_lattice().fundamental_
↪weights()
sage: L = g.simple_module(La[1])
sage: res = L.bgg_resolution()
sage: [res.module_order(i) for i in range(7)]
[[1],
 [s2, s1],
 [s2*s1, s1*s2],
 [s1*s2*s1, s2*s1*s2],
 [s1*s2*s1*s2, s2*s1*s2*s1],
 [s1*s2*s1*s2*s1, s2*s1*s2*s1*s2],
 [s2*s1*s2*s1*s2*s1]]

```

```

>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['G', Integer(2)])
>>> La = g.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> L = g.simple_module(La[Integer(1)])
>>> res = L.bgg_resolution()
>>> [res.module_order(i) for i in range(Integer(7))]
[[1],
 [s2, s1],
 [s2*s1, s1*s2],
 [s1*s2*s1, s2*s1*s2],
 [s1*s2*s1*s2, s2*s1*s2*s1],
 [s1*s2*s1*s2*s1, s2*s1*s2*s1*s2],
 [s2*s1*s2*s1*s2*s1]]

```

simple_module()Return the simple module L_λ defining self.

EXAMPLES:

```

sage: g = LieAlgebra(QQ, cartan_type=['C', 2])
sage: La = g.cartan_type().root_system().weight_lattice().fundamental_
↪weights()
sage: L = g.simple_module(La[1] + La[2])
sage: res = L.bgg_resolution()
sage: res.simple_module() is L
True

```

```

>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['C', Integer(2)])
>>> La = g.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> L = g.simple_module(La[Integer(1)] + La[Integer(2)])
>>> res = L.bgg_resolution()
>>> res.simple_module() is L
True

```

sage.algebras.lie_algebras.bgg_resolution.build_differentials()Construct the differentials for the BGG resolution corresponding to the Weyl group W .

ALGORITHM:

We use the fact that (parabolic) Bruhat order is built locally from squares, all values defining the differential are $+1$ or -1 , and the product over the two different paths must sum to 0. This is outlined in Ch. 6 of [Humphreys08].

This only depends on the Coxeter group W . There is no stabilizer for any dominant integral weight λ under the dot action (i.e., the stabilizer of $\lambda + \rho$ is empty).

EXAMPLES:

```
sage: from sage.algebras.lie_algebras.bgg_resolution import build_differentials
sage: W = WeylGroup(['B', 2], prefix='s')
sage: D, O = build_differentials(W)
sage: D
{0: [],
 1: [-1  1],
 2: [1  1]
    [1  1],
 3: [ 1 -1]
    [-1  1],
 4: [1]
    [1],
 5: []}
sage: O
{0: [1],
 1: [s2, s1],
 2: [s2*s1, s1*s2],
 3: [s2*s1*s2, s1*s2*s1],
 4: [s2*s1*s2*s1]}
```

```
>>> from sage.all import *
>>> from sage.algebras.lie_algebras.bgg_resolution import build_differentials
>>> W = WeylGroup(['B', Integer(2)], prefix='s')
>>> D, O = build_differentials(W)
>>> D
{0: [],
 1: [-1  1],
 2: [1  1]
    [1  1],
 3: [ 1 -1]
    [-1  1],
 4: [1]
    [1],
 5: []}
>>> O
{0: [1],
 1: [s2, s1],
 2: [s2*s1, s1*s2],
 3: [s2*s1*s2, s1*s2*s1],
 4: [s2*s1*s2*s1]}
```

9.2.5 BGG Category O Dual Modules

AUTHORS:

- Travis Scrimshaw (2024-01-07): Initial version

```
class sage.algebras.lie_algebras.bgg_dual_module.BGGDualModule(module)
```

Bases: `CombinatorialFreeModule`

The dual module $M^\vee$ in the BGG Category $\mathcal{O}$.

Let τ be the transpose map of a semisimple (finite dimensional) Lie algebra $\mathfrak{g}$ over a field R . Let $M \in \mathcal{O}$. The *BGG dual module* is the R -module $M^\vee := \bigoplus_\lambda M_\lambda^*$ which has a $U(\mathfrak{g})$ -module structure given by

$$x \cdot \phi(v) := \phi(\tau(x) \cdot v),$$

which is also a weight module with the same grading as M .

The basis we chose to work with here is the natural dual basis to the distinguished basis B of M . That is, we define the dual function to b as $\phi_b(c) = \delta_{bc}$.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['A', 1])
sage: La = g.cartan_type().root_system().weight_lattice().fundamental_weights()
sage: M = g.verma_module(2*La[1])
sage: Mc = M.dual()
sage: B = Mc.basis()
sage: it = iter(B)
sage: elts = [next(it) for _ in range(7)]; elts
[v[2*Lambda[1]]^*,
 f[-alpha[1]]*v[2*Lambda[1]]^*,
 f[-alpha[1]]^2*v[2*Lambda[1]]^*,
 f[-alpha[1]]^3*v[2*Lambda[1]]^*,
 f[-alpha[1]]^4*v[2*Lambda[1]]^*,
 f[-alpha[1]]^5*v[2*Lambda[1]]^*,
 f[-alpha[1]]^6*v[2*Lambda[1]]^*]
sage: e, h, f = g.pbw_basis().algebra_generators()
sage: [f * vec for vec in elts]
[2*f[-alpha[1]]*v[2*Lambda[1]]^*,
 2*f[-alpha[1]]^2*v[2*Lambda[1]]^*,
 0,
 -4*f[-alpha[1]]^4*v[2*Lambda[1]]^*,
 -10*f[-alpha[1]]^5*v[2*Lambda[1]]^*,
 -18*f[-alpha[1]]^6*v[2*Lambda[1]]^*,
 -28*f[-alpha[1]]^7*v[2*Lambda[1]]^*]
sage: [e * vec for vec in elts]
[0,
 v[2*Lambda[1]]^*,
 f[-alpha[1]]*v[2*Lambda[1]]^*,
 f[-alpha[1]]^2*v[2*Lambda[1]]^*,
 f[-alpha[1]]^3*v[2*Lambda[1]]^*,
 f[-alpha[1]]^4*v[2*Lambda[1]]^*,
 f[-alpha[1]]^5*v[2*Lambda[1]]^*]
sage: [h * vec for vec in elts]
[2*v[2*Lambda[1]]^*,
 0,
 -2*f[-alpha[1]]^2*v[2*Lambda[1]]^*,
 -4*f[-alpha[1]]^3*v[2*Lambda[1]]^*,
 -6*f[-alpha[1]]^4*v[2*Lambda[1]]^*,
 -8*f[-alpha[1]]^5*v[2*Lambda[1]]^*,
 -10*f[-alpha[1]]^6*v[2*Lambda[1]]^*]
```

```

>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['A', Integer(1)])
>>> La = g.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> M = g.verma_module(Integer(2)*La[Integer(1)])
>>> Mc = M.dual()
>>> B = Mc.basis()
>>> it = iter(B)
>>> elts = [next(it) for _ in range(Integer(7))]; elts
[v[2*Lambda[1]]^*,
 f[-alpha[1]]*v[2*Lambda[1]]^*,
 f[-alpha[1]]^2*v[2*Lambda[1]]^*,
 f[-alpha[1]]^3*v[2*Lambda[1]]^*,
 f[-alpha[1]]^4*v[2*Lambda[1]]^*,
 f[-alpha[1]]^5*v[2*Lambda[1]]^*,
 f[-alpha[1]]^6*v[2*Lambda[1]]^*]
>>> e, h, f = g.pbw_basis().algebra_generators()
>>> [f * vec for vec in elts]
[2*f[-alpha[1]]*v[2*Lambda[1]]^*,
 2*f[-alpha[1]]^2*v[2*Lambda[1]]^*,
 0,
 -4*f[-alpha[1]]^4*v[2*Lambda[1]]^*,
 -10*f[-alpha[1]]^5*v[2*Lambda[1]]^*,
 -18*f[-alpha[1]]^6*v[2*Lambda[1]]^*,
 -28*f[-alpha[1]]^7*v[2*Lambda[1]]^*]
>>> [e * vec for vec in elts]
[0,
 v[2*Lambda[1]]^*,
 f[-alpha[1]]*v[2*Lambda[1]]^*,
 f[-alpha[1]]^2*v[2*Lambda[1]]^*,
 f[-alpha[1]]^3*v[2*Lambda[1]]^*,
 f[-alpha[1]]^4*v[2*Lambda[1]]^*,
 f[-alpha[1]]^5*v[2*Lambda[1]]^*]
>>> [h * vec for vec in elts]
[2*v[2*Lambda[1]]^*,
 0,
 -2*f[-alpha[1]]^2*v[2*Lambda[1]]^*,
 -4*f[-alpha[1]]^3*v[2*Lambda[1]]^*,
 -6*f[-alpha[1]]^4*v[2*Lambda[1]]^*,
 -8*f[-alpha[1]]^5*v[2*Lambda[1]]^*,
 -10*f[-alpha[1]]^6*v[2*Lambda[1]]^*]

```

REFERENCES:

- [Humphreys08]

class ElementBases: `IndexedFreeModuleElement`**degree_on_basis**(*m*)Return the degree of the basis element indexed by *m*.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['D', 5])
```

(continues on next page)

(continued from previous page)

```

sage: La = g.cartan_type().root_system().weight_space().fundamental_weights()
sage: M = g.verma_module(La[1] + La[4] - 1/3*La[5])
sage: Mc = M.dual()
sage: elt = Mc.an_element(); elt
f[-alpha[5]]^2*f[-alpha[4]]^2*f[-alpha[3]]^3*v[Lambda[1] + Lambda[4] - 1/
↪ 3*Lambda[5]]^*
+ 2*f[-alpha[5]]*v[Lambda[1] + Lambda[4] - 1/3*Lambda[5]]^*
+ 3*f[-alpha[4]]*v[Lambda[1] + Lambda[4] - 1/3*Lambda[5]]^*
+ v[Lambda[1] + Lambda[4] - 1/3*Lambda[5]]^*
sage: [M.degree_on_basis(m) for m in elt.support()]
[Lambda[1] + 3*Lambda[2] - 2*Lambda[3] - 4/3*Lambda[5],
 Lambda[1] + Lambda[4] - 1/3*Lambda[5],
 Lambda[1] + Lambda[3] + Lambda[4] - 7/3*Lambda[5],
 Lambda[1] + Lambda[3] - Lambda[4] - 1/3*Lambda[5]]

```

```

>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['D', Integer(5)])
>>> La = g.cartan_type().root_system().weight_space().fundamental_weights()
>>> M = g.verma_module(La[Integer(1)] + La[Integer(4)] - Integer(1)/
↪ Integer(3)*La[Integer(5)])
>>> Mc = M.dual()
>>> elt = Mc.an_element(); elt
f[-alpha[5]]^2*f[-alpha[4]]^2*f[-alpha[3]]^3*v[Lambda[1] + Lambda[4] - 1/
↪ 3*Lambda[5]]^*
+ 2*f[-alpha[5]]*v[Lambda[1] + Lambda[4] - 1/3*Lambda[5]]^*
+ 3*f[-alpha[4]]*v[Lambda[1] + Lambda[4] - 1/3*Lambda[5]]^*
+ v[Lambda[1] + Lambda[4] - 1/3*Lambda[5]]^*
>>> [M.degree_on_basis(m) for m in elt.support()]
[Lambda[1] + 3*Lambda[2] - 2*Lambda[3] - 4/3*Lambda[5],
 Lambda[1] + Lambda[4] - 1/3*Lambda[5],
 Lambda[1] + Lambda[3] + Lambda[4] - 7/3*Lambda[5],
 Lambda[1] + Lambda[3] - Lambda[4] - 1/3*Lambda[5]]

```

dual()

Return the dual module of self.

In Category $\mathcal{O}$, we have $(M^\vee)^\vee \cong M$, so we return the defining module M of $M^\vee$.

EXAMPLES:

```

sage: g = LieAlgebra(QQ, cartan_type=['F', 4])
sage: La = g.cartan_type().root_system().weight_space().fundamental_weights()
sage: M = g.verma_module(La[1] - 5/3*La[2] + 3*La[4])
sage: Mc = M.dual()
sage: Mc.dual() is M
True

```

```

>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['F', Integer(4)])
>>> La = g.cartan_type().root_system().weight_space().fundamental_weights()
>>> M = g.verma_module(La[Integer(1)] - Integer(5)/Integer(3)*La[Integer(2)] -
↪ Integer(3)*La[Integer(4)])

```

(continues on next page)

(continued from previous page)

```
>>> Mc = M.dual()
>>> Mc.dual() is M
True
```

highest_weight()

Return the highest weight of `self`.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['E', 7])
sage: La = g.cartan_type().root_system().weight_space().fundamental_weights()
sage: M = g.verma_module(2*La[1] + 5/3*La[4] - 3*La[6])
sage: Mc = M.dual()
sage: Mc.highest_weight()
2*Lambda[1] + 5/3*Lambda[4] - 3*Lambda[6]
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['E', Integer(7)])
>>> La = g.cartan_type().root_system().weight_space().fundamental_weights()
>>> M = g.verma_module(Integer(2)*La[Integer(1)] + Integer(5)/
↳ Integer(3)*La[Integer(4)] - Integer(3)*La[Integer(6)])
>>> Mc = M.dual()
>>> Mc.highest_weight()
2*Lambda[1] + 5/3*Lambda[4] - 3*Lambda[6]
```

highest_weight_vector()

Return the highest weight vector of `self` (assuming the defining module defines such a vector).

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['A', 1])
sage: La = g.cartan_type().root_system().weight_lattice().fundamental_
↳ weights()
sage: M = g.verma_module(2*La[1])
sage: Mc = M.dual()
sage: Mc.highest_weight_vector()
v[2*Lambda[1]]^*
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['A', Integer(1)])
>>> La = g.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> M = g.verma_module(Integer(2)*La[Integer(1)])
>>> Mc = M.dual()
>>> Mc.highest_weight_vector()
v[2*Lambda[1]]^*
```

lie_algebra()

Return the underlying Lie algebra of `self`.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['B', 3])
sage: La = g.cartan_type().root_system().weight_lattice().fundamental_
```

(continues on next page)

(continued from previous page)

```

↪weights()
sage: M = g.verma_module(2*La[1] + La[3])
sage: Mc = M.dual()
sage: Mc.lie_algebra() is g
True

```

```

>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['B', Integer(3)])
>>> La = g.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> M = g.verma_module(Integer(2)*La[Integer(1)] + La[Integer(3)])
>>> Mc = M.dual()
>>> Mc.lie_algebra() is g
True

```

```

class sage.algebras.lie_algebras.bgg_dual_module.FiniteDimensionalSimpleModule(g, weight,
                                                                                   prefix='f',
                                                                                   ba-
                                                                                   sis_key=None,
                                                                                   **kws)

```

Bases: *SimpleModule*

A finite dimensional simple module.

bgg_resolution()

Return the BGG resolution of self.

EXAMPLES:

```

sage: g = LieAlgebra(QQ, cartan_type=['A', 2])
sage: La = g.cartan_type().root_system().weight_lattice().fundamental_
↪weights()
sage: L = g.simple_module(La[1] + La[2])
sage: L.bgg_resolution()
BGG resolution of Simple module with highest weight Lambda[1] + Lambda[2]
of Lie algebra of ['A', 2] in the Chevalley basis

```

```

>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['A', Integer(2)])
>>> La = g.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> L = g.simple_module(La[Integer(1)] + La[Integer(2)])
>>> L.bgg_resolution()
BGG resolution of Simple module with highest weight Lambda[1] + Lambda[2]
of Lie algebra of ['A', 2] in the Chevalley basis

```

```

class sage.algebras.lie_algebras.bgg_dual_module.SimpleModule(g, weight, prefix='f',
                                                                basis_key=None, **kws)

```

Bases: *ModulePrinting*, *CombinatorialFreeModule*

Return the simple module L_λ as the image of the natural morphism $\phi : M_\lambda \rightarrow M_\lambda^\vee$.

class Element

Bases: *IndexedFreeModuleElement*

ambient()

Return the ambient module of self.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['G', 2])
sage: La = g.cartan_type().root_system().weight_lattice().fundamental_
↪weights()
sage: L = g.simple_module(2*La[1])
sage: L.ambient()
BGG Dual of Verma module with highest weight 2*Lambda[1] of
Lie algebra of ['G', 2] in the Chevalley basis
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['G', Integer(2)])
>>> La = g.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> L = g.simple_module(Integer(2)*La[Integer(1)])
>>> L.ambient()
BGG Dual of Verma module with highest weight 2*Lambda[1] of
Lie algebra of ['G', 2] in the Chevalley basis
```

dual()

Return the dual module of self, which is self since simple modules are self-dual.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['B', 4])
sage: La = g.cartan_type().root_system().weight_lattice().fundamental_
↪weights()
sage: L = g.simple_module(2*La[1] + 3*La[4])
sage: L.dual() is L
True
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['B', Integer(4)])
>>> La = g.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> L = g.simple_module(Integer(2)*La[Integer(1)] + Integer(3)*La[Integer(4)])
>>> L.dual() is L
True
```

highest_weight()

Return the highest weight of self.

EXAMPLES:

```
sage: g = lie_algebras.so(QQ, 7)
sage: La = g.cartan_type().root_system().weight_lattice().fundamental_
↪weights()
sage: L = g.simple_module(La[1] + La[2])
sage: L.highest_weight()
Lambda[1] + Lambda[2]
```

```
>>> from sage.all import *
>>> g = lie_algebras.so(QQ, Integer(7))
```

(continues on next page)

(continued from previous page)

```
>>> La = g.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> L = g.simple_module(La[Integer(1)] + La[Integer(2)])
>>> L.highest_weight()
Lambda[1] + Lambda[2]
```

highest_weight_vector()

Return the highest weight vector of self.

EXAMPLES:

```
sage: g = lie_algebras.sp(QQ, 6)
sage: La = g.cartan_type().root_system().weight_lattice().fundamental_
↪weights()
sage: L = g.simple_module(La[1] + La[2])
sage: L.highest_weight_vector()
u[Lambda[1] + Lambda[2]]
```

```
>>> from sage.all import *
>>> g = lie_algebras.sp(QQ, Integer(6))
>>> La = g.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> L = g.simple_module(La[Integer(1)] + La[Integer(2)])
>>> L.highest_weight_vector()
u[Lambda[1] + Lambda[2]]
```

homogeneous_component_basis(mu)

Return a basis for the mu weight space of self.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['A', 2])
sage: P = g.cartan_type().root_system().weight_lattice()
sage: La = P.fundamental_weights()
sage: la = La[1] + La[2]
sage: L = g.simple_module(la)
sage: from itertools import product
sage: al = P.simple_roots()
sage: for wts in product(range(4), repeat=2):
....:     mu = la - wts[0] * al[1] - wts[1] * al[2]
....:     print(mu)
....:     print(L.homogeneous_component_basis(mu))
Lambda[1] + Lambda[2]
Family (u[Lambda[1] + Lambda[2]],)
2*Lambda[1] - Lambda[2]
Family (f[-alpha[2]]*u[Lambda[1] + Lambda[2]],)
3*Lambda[1] - 3*Lambda[2]
Family ()
4*Lambda[1] - 5*Lambda[2]
Family ()
-Lambda[1] + 2*Lambda[2]
Family (f[-alpha[1]]*u[Lambda[1] + Lambda[2]],)
0
Family (f[-alpha[1] - alpha[2]]*u[Lambda[1] + Lambda[2]], f[-alpha[2]]*f[-
↪alpha[1]]*u[Lambda[1] + Lambda[2]])
```

(continues on next page)

(continued from previous page)

```

Lambda[1] - 2*Lambda[2]
Family (f[-alpha[2]]*f[-alpha[1] - alpha[2]]*u[Lambda[1] + Lambda[2]],)
2*Lambda[1] - 4*Lambda[2]
Family ()
-3*Lambda[1] + 3*Lambda[2]
Family ()
-2*Lambda[1] + Lambda[2]
Family (f[-alpha[1]]*f[-alpha[1] - alpha[2]]*u[Lambda[1] + Lambda[2]],)
-Lambda[1] - Lambda[2]
Family (f[-alpha[1] - alpha[2]]^2*u[Lambda[1] + Lambda[2]],)
-3*Lambda[2]
Family ()
-5*Lambda[1] + 4*Lambda[2]
Family ()
-4*Lambda[1] + 2*Lambda[2]
Family ()
-3*Lambda[1]
Family ()
-2*Lambda[1] - 2*Lambda[2]
Family ()

```

```

>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['A', Integer(2)])
>>> P = g.cartan_type().root_system().weight_lattice()
>>> La = P.fundamental_weights()
>>> la = La[Integer(1)] + La[Integer(2)]
>>> L = g.simple_module(la)
>>> from itertools import product
>>> al = P.simple_roots()
>>> for wts in product(range(Integer(4)), repeat=Integer(2)):
...     mu = la - wts[Integer(0)] * al[Integer(1)] - wts[Integer(1)] * _
↪ al[Integer(2)]
...     print(mu)
...     print(L.homogeneous_component_basis(mu))
Lambda[1] + Lambda[2]
Family (u[Lambda[1] + Lambda[2]],)
2*Lambda[1] - Lambda[2]
Family (f[-alpha[2]]*u[Lambda[1] + Lambda[2]],)
3*Lambda[1] - 3*Lambda[2]
Family ()
4*Lambda[1] - 5*Lambda[2]
Family ()
-Lambda[1] + 2*Lambda[2]
Family (f[-alpha[1]]*u[Lambda[1] + Lambda[2]],)
0
Family (f[-alpha[1] - alpha[2]]*u[Lambda[1] + Lambda[2]], f[-alpha[2]]*f[-
↪ alpha[1]]*u[Lambda[1] + Lambda[2]])
Lambda[1] - 2*Lambda[2]
Family (f[-alpha[2]]*f[-alpha[1] - alpha[2]]*u[Lambda[1] + Lambda[2]],)
2*Lambda[1] - 4*Lambda[2]
Family ()
-3*Lambda[1] + 3*Lambda[2]

```

(continues on next page)

(continued from previous page)

```

Family ()
-2*Lambda[1] + Lambda[2]
Family (f[-alpha[1]]*f[-alpha[1] - alpha[2]]*u[Lambda[1] + Lambda[2]],)
-Lambda[1] - Lambda[2]
Family (f[-alpha[1] - alpha[2]]^2*u[Lambda[1] + Lambda[2]],)
-3*Lambda[2]
Family ()
-5*Lambda[1] + 4*Lambda[2]
Family ()
-4*Lambda[1] + 2*Lambda[2]
Family ()
-3*Lambda[1]
Family ()
-2*Lambda[1] - 2*Lambda[2]
Family ()

```

lie_algebra()

Return the underlying Lie algebra of self.

EXAMPLES:

```

sage: g = lie_algebras.so(QQ, 9)
sage: La = g.cartan_type().root_system().weight_space().fundamental_weights()
sage: L = g.simple_module(La[3] - 1/2*La[1])
sage: L.lie_algebra()
Lie algebra of ['B', 4] in the Chevalley basis

```

```

>>> from sage.all import *
>>> g = lie_algebras.so(QQ, Integer(9))
>>> La = g.cartan_type().root_system().weight_space().fundamental_weights()
>>> L = g.simple_module(La[Integer(3)] - Integer(1)/Integer(2)*La[Integer(1)])
>>> L.lie_algebra()
Lie algebra of ['B', 4] in the Chevalley basis

```

lift()

Return the lift map of self to the ambient dual Verma module.

EXAMPLES:

```

sage: g = LieAlgebra(QQ, cartan_type=['G', 2])
sage: La = g.cartan_type().root_system().weight_lattice().fundamental_
↪ weights()
sage: L = g.simple_module(La[1])
sage: [L.lift(b) for b in L.basis()] # long time
[v[Lambda[1]]^*,
 f[-alpha[1]]*v[Lambda[1]]^*,
 f[-alpha[2]]*f[-alpha[1]]*v[Lambda[1]]^* - f[-alpha[1] - ↪
↪ alpha[2]]*v[Lambda[1]]^*,
 f[-alpha[1]]*f[-alpha[1] - alpha[2]]*v[Lambda[1]]^*
 + f[-2*alpha[1] - alpha[2]]*v[Lambda[1]]^*,
 f[-alpha[1]]^2*f[-alpha[1] - alpha[2]]*v[Lambda[1]]^*
 + f[-alpha[1]]*f[-2*alpha[1] - alpha[2]]*v[Lambda[1]]^*
 + 1/2*f[-3*alpha[1] - alpha[2]]*v[Lambda[1]]^*,

```

(continues on next page)

(continued from previous page)

```

f[-alpha[2]]*f[-alpha[1]]^2*f[-alpha[1] - alpha[2]]*v[Lambda[1]]^*
+ f[-alpha[2]]*f[-alpha[1]]*f[-2*alpha[1] - alpha[2]]*v[Lambda[1]]^*
+ 1/2*f[-alpha[2]]*f[-3*alpha[1] - alpha[2]]*v[Lambda[1]]^*
- f[-alpha[1] - alpha[2]]*f[-2*alpha[1] - alpha[2]]*v[Lambda[1]]^*
+ 1/2*f[-3*alpha[1] - 2*alpha[2]]*v[Lambda[1]]^*,
f[-alpha[1]]*f[-alpha[1] - alpha[2]]*f[-2*alpha[1] - alpha[2]]*v[Lambda[1]]^*
- 1/2*f[-alpha[1]]*f[-3*alpha[1] - 2*alpha[2]]*v[Lambda[1]]^*
- 1/2*f[-alpha[1] - alpha[2]]*f[-3*alpha[1] - alpha[2]]*v[Lambda[1]]^*
+ f[-2*alpha[1] - alpha[2]]^2*v[Lambda[1]]^*

```

```

>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['G', Integer(2)])
>>> La = g.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> L = g.simple_module(La[Integer(1)])
>>> [L.lift(b) for b in L.basis()] # long time
[v[Lambda[1]]^*,
 f[-alpha[1]]*v[Lambda[1]]^*,
 f[-alpha[2]]*f[-alpha[1]]*v[Lambda[1]]^* - f[-alpha[1] -
↪alpha[2]]*v[Lambda[1]]^*,
 f[-alpha[1]]*f[-alpha[1] - alpha[2]]*v[Lambda[1]]^*
 + f[-2*alpha[1] - alpha[2]]*v[Lambda[1]]^*,
 f[-alpha[1]]^2*f[-alpha[1] - alpha[2]]*v[Lambda[1]]^*
 + f[-alpha[1]]*f[-2*alpha[1] - alpha[2]]*v[Lambda[1]]^*
 + 1/2*f[-3*alpha[1] - alpha[2]]*v[Lambda[1]]^*,
 f[-alpha[2]]*f[-alpha[1]]^2*f[-alpha[1] - alpha[2]]*v[Lambda[1]]^*
 + f[-alpha[2]]*f[-alpha[1]]*f[-2*alpha[1] - alpha[2]]*v[Lambda[1]]^*
 + 1/2*f[-alpha[2]]*f[-3*alpha[1] - alpha[2]]*v[Lambda[1]]^*
 - f[-alpha[1] - alpha[2]]*f[-2*alpha[1] - alpha[2]]*v[Lambda[1]]^*
 + 1/2*f[-3*alpha[1] - 2*alpha[2]]*v[Lambda[1]]^*,
 f[-alpha[1]]*f[-alpha[1] - alpha[2]]*f[-2*alpha[1] - alpha[2]]*v[Lambda[1]]^*
 - 1/2*f[-alpha[1]]*f[-3*alpha[1] - 2*alpha[2]]*v[Lambda[1]]^*
 - 1/2*f[-alpha[1] - alpha[2]]*f[-3*alpha[1] - alpha[2]]*v[Lambda[1]]^*
 + f[-2*alpha[1] - alpha[2]]^2*v[Lambda[1]]^*

```

pbw_basis()

Return the PBW basis of the underlying Lie algebra used to define self.

EXAMPLES:

```

sage: g = lie_algebras.so(QQ, 8)
sage: La = g.cartan_type().root_system().weight_lattice().fundamental_
↪weights()
sage: L = g.simple_module(La[2] - 2*La[3])
sage: L.pbw_basis()
Universal enveloping algebra of Lie algebra of ['D', 4] in the Chevalley basis
in the Poincare-Birkhoff-Witt basis

```

```

>>> from sage.all import *
>>> g = lie_algebras.so(QQ, Integer(8))
>>> La = g.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> L = g.simple_module(La[Integer(2)] - Integer(2)*La[Integer(3)])
>>> L.pbw_basis()

```

(continues on next page)

(continued from previous page)

Universal enveloping algebra of Lie algebra of ['D', 4] in the Chevalley basis
in the Poincare-Birkhoff-Witt basis

retract(x)

Return the retraction of x in self.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['A', 2])
sage: La = g.cartan_type().root_system().weight_lattice().fundamental_
↳weights()
sage: L = g.simple_module(2*La[1])
sage: L.retract(L.lift(sum(L.basis())))
f[-alpha[1]]^2*u[2*Lambda[1]] + f[-alpha[1]]*f[-alpha[1] -
↳alpha[2]]*u[2*Lambda[1]]
+ f[-alpha[1] - alpha[2]]^2*u[2*Lambda[1]] + f[-alpha[1]]*u[2*Lambda[1]]
+ f[-alpha[1] - alpha[2]]*u[2*Lambda[1]] + u[2*Lambda[1]]
sage: B = list(L.basis())
sage: L.retract(3/2*L.lift(B[0]) - L.lift(B[2]) - 10/3*L.lift(B[3]))
-10/3*f[-alpha[1]]^2*u[2*Lambda[1]]
- f[-alpha[1] - alpha[2]]*u[2*Lambda[1]]
+ 3/2*u[2*Lambda[1]]
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['A', Integer(2)])
>>> La = g.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> L = g.simple_module(Integer(2)*La[Integer(1)])
>>> L.retract(L.lift(sum(L.basis())))
f[-alpha[1]]^2*u[2*Lambda[1]] + f[-alpha[1]]*f[-alpha[1] -
↳alpha[2]]*u[2*Lambda[1]]
+ f[-alpha[1] - alpha[2]]^2*u[2*Lambda[1]] + f[-alpha[1]]*u[2*Lambda[1]]
+ f[-alpha[1] - alpha[2]]*u[2*Lambda[1]] + u[2*Lambda[1]]
>>> B = list(L.basis())
>>> L.retract(Integer(3)/Integer(2)*L.lift(B[Integer(0)]) - L.
↳lift(B[Integer(2)]) - Integer(10)/Integer(3)*L.lift(B[Integer(3)]))
-10/3*f[-alpha[1]]^2*u[2*Lambda[1]]
- f[-alpha[1] - alpha[2]]*u[2*Lambda[1]]
+ 3/2*u[2*Lambda[1]]
```

weight_space_basis(mu)

Return a basis for the mu weight space of self.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['A', 2])
sage: P = g.cartan_type().root_system().weight_lattice()
sage: La = P.fundamental_weights()
sage: la = La[1] + La[2]
sage: L = g.simple_module(la)
sage: from itertools import product
sage: al = P.simple_roots()
sage: for wts in product(range(4), repeat=2):
.....:     mu = la - wts[0] * al[1] - wts[1] * al[2]
```

(continues on next page)

(continued from previous page)

```

.....:     print(mu)
.....:     print(L.homogeneous_component_basis(mu))
Lambda[1] + Lambda[2]
Family (u[Lambda[1] + Lambda[2]],)
2*Lambda[1] - Lambda[2]
Family (f[-alpha[2]]*u[Lambda[1] + Lambda[2]],)
3*Lambda[1] - 3*Lambda[2]
Family ()
4*Lambda[1] - 5*Lambda[2]
Family ()
-Lambda[1] + 2*Lambda[2]
Family (f[-alpha[1]]*u[Lambda[1] + Lambda[2]],)
0
Family (f[-alpha[1] - alpha[2]]*u[Lambda[1] + Lambda[2]], f[-alpha[2]]*f[-
↪alpha[1]]*u[Lambda[1] + Lambda[2]])
Lambda[1] - 2*Lambda[2]
Family (f[-alpha[2]]*f[-alpha[1] - alpha[2]]*u[Lambda[1] + Lambda[2]],)
2*Lambda[1] - 4*Lambda[2]
Family ()
-3*Lambda[1] + 3*Lambda[2]
Family ()
-2*Lambda[1] + Lambda[2]
Family (f[-alpha[1]]*f[-alpha[1] - alpha[2]]*u[Lambda[1] + Lambda[2]],)
-Lambda[1] - Lambda[2]
Family (f[-alpha[1] - alpha[2]]^2*u[Lambda[1] + Lambda[2]],)
-3*Lambda[2]
Family ()
-5*Lambda[1] + 4*Lambda[2]
Family ()
-4*Lambda[1] + 2*Lambda[2]
Family ()
-3*Lambda[1]
Family ()
-2*Lambda[1] - 2*Lambda[2]
Family ()

```

```

>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['A', Integer(2)])
>>> P = g.cartan_type().root_system().weight_lattice()
>>> La = P.fundamental_weights()
>>> la = La[Integer(1)] + La[Integer(2)]
>>> L = g.simple_module(la)
>>> from itertools import product
>>> al = P.simple_roots()
>>> for wts in product(range(Integer(4)), repeat=Integer(2)):
...     mu = la - wts[Integer(0)] * al[Integer(1)] - wts[Integer(1)] * _
↪al[Integer(2)]
...     print(mu)
...     print(L.homogeneous_component_basis(mu))
Lambda[1] + Lambda[2]
Family (u[Lambda[1] + Lambda[2]],)
2*Lambda[1] - Lambda[2]

```

(continues on next page)

(continued from previous page)

```

Family (f[-alpha[2]]*u[Lambda[1] + Lambda[2]],)
3*Lambda[1] - 3*Lambda[2]
Family ()
4*Lambda[1] - 5*Lambda[2]
Family ()
-Lambda[1] + 2*Lambda[2]
Family (f[-alpha[1]]*u[Lambda[1] + Lambda[2]],)
0
Family (f[-alpha[1] - alpha[2]]*u[Lambda[1] + Lambda[2]], f[-alpha[2]]*f[-
↪alpha[1]]*u[Lambda[1] + Lambda[2]])
Lambda[1] - 2*Lambda[2]
Family (f[-alpha[2]]*f[-alpha[1] - alpha[2]]*u[Lambda[1] + Lambda[2]],)
2*Lambda[1] - 4*Lambda[2]
Family ()
-3*Lambda[1] + 3*Lambda[2]
Family ()
-2*Lambda[1] + Lambda[2]
Family (f[-alpha[1]]*f[-alpha[1] - alpha[2]]*u[Lambda[1] + Lambda[2]],)
-Lambda[1] - Lambda[2]
Family (f[-alpha[1] - alpha[2]]^2*u[Lambda[1] + Lambda[2]],)
-3*Lambda[2]
Family ()
-5*Lambda[1] + 4*Lambda[2]
Family ()
-4*Lambda[1] + 2*Lambda[2]
Family ()
-3*Lambda[1]
Family ()
-2*Lambda[1] - 2*Lambda[2]
Family ()

```

```

class sage.algebras.lie_algebras.bgg_dual_module.SimpleModuleIndices (simple, prefix,
                                                                    category=None,
                                                                    **kwds)

```

Bases: `IndexedFreeAbelianMonoid`

The indices of the basis for a simple $U(\mathfrak{g})$ -module.

Note

The current implementation assumes the Lie algebra $\mathfrak{g}$ is finite dimensional.

cardinality()

Return the cardinality of `self`.

EXAMPLES:

```

sage: g = LieAlgebra(QQ, cartan_type=['E', 6])
sage: La = g.cartan_type().root_system().weight_lattice().fundamental_
↪weights()
sage: L = g.simple_module(La[1]+La[4])

```

(continues on next page)

(continued from previous page)

```
sage: L._indices.cardinality()
51975
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['E', Integer(6)])
>>> La = g.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> L = g.simple_module(La[Integer(1)]+La[Integer(4)])
>>> L._indices.cardinality()
51975
```

weight_space_basis(mu)

Return the indices of the μ weight space basis elements.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['A', 2])
sage: P = g.cartan_type().root_system().weight_lattice()
sage: La = P.fundamental_weights()
sage: al = P.simple_roots()
sage: wt = -3*La[1] + 3*La[2]
sage: I = g.simple_module(wt).indices()
sage: I.weight_space_basis(wt)
[1]
sage: I.weight_space_basis(wt - al[1])
[f[-alpha[1]]]
sage: I.weight_space_basis(wt - al[2])
[f[-alpha[2]]]
sage: I.weight_space_basis(wt - al[1] - al[2])
[f[-alpha[1] - alpha[2]], f[-alpha[2]]*f[-alpha[1]]]
sage: I.weight_space_basis(wt - 4*al[1])
[f[-alpha[1]]^4]
sage: I.weight_space_basis(wt - 4*al[2])
[]
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['A', Integer(2)])
>>> P = g.cartan_type().root_system().weight_lattice()
>>> La = P.fundamental_weights()
>>> al = P.simple_roots()
>>> wt = -Integer(3)*La[Integer(1)] + Integer(3)*La[Integer(2)]
>>> I = g.simple_module(wt).indices()
>>> I.weight_space_basis(wt)
[1]
>>> I.weight_space_basis(wt - al[Integer(1)])
[f[-alpha[1]]]
>>> I.weight_space_basis(wt - al[Integer(2)])
[f[-alpha[2]]]
>>> I.weight_space_basis(wt - al[Integer(1)] - al[Integer(2)])
[f[-alpha[1] - alpha[2]], f[-alpha[2]]*f[-alpha[1]]]
>>> I.weight_space_basis(wt - Integer(4)*al[Integer(1)])
[f[-alpha[1]]^4]
>>> I.weight_space_basis(wt - Integer(4)*al[Integer(2)])
```

(continues on next page)

(continued from previous page)

[]

9.2.6 Center of a Universal Enveloping Algebra

AUTHORS:

- Travis Scrimshaw (2024-01-02): Initial version

class sage.algebras.lie_algebras.center_uea.**CenterIndices** (*center*, *indices=None*)

Bases: `IndexedFreeAbelianMonoid`

Set of basis indices for the center of a universal enveloping algebra.

This also constructs the lift from the center to the universal enveloping algebra as part of computing the generators and basis elements. The basic algorithm is to construct the centralizer of each filtered component in increasing order (as each is a finite dimensional vector space). For more precise details, see [Motsak2006].

degree (*m*)

Return the degree of *m* in *self*.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['E', 6])
sage: U = g.pbw_basis()
sage: Z = U.center()
sage: I = Z.indices()
sage: [I.degree(g) for g in I.monoid_generators()]
[2, 5, 6, 8, 9, 12]
sage: [(elt, I.degree(elt)) for elt in I.some_elements()]
[(1, 0), (Z[0], 2), (Z[0]^2, 4), (Z[1], 5), (Z[0]^3*Z[1], 11),
 (Z[0]^10, 20), (Z[0]*Z[1]^4, 22)]
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['E', Integer(6)])
>>> U = g.pbw_basis()
>>> Z = U.center()
>>> I = Z.indices()
>>> [I.degree(g) for g in I.monoid_generators()]
[2, 5, 6, 8, 9, 12]
>>> [(elt, I.degree(elt)) for elt in I.some_elements()]
[(1, 0), (Z[0], 2), (Z[0]^2, 4), (Z[1], 5), (Z[0]^3*Z[1], 11),
 (Z[0]^10, 20), (Z[0]*Z[1]^4, 22)]
```

lift_on_basis (*m*)

Return the image of the basis element indexed by *m* in the universal enveloping algebra.

EXAMPLES:

```
sage: g = lie_algebras.Heisenberg(QQ, 3)
sage: U = g.pbw_basis()
sage: Z = U.center()
sage: I = Z.indices()
sage: z0 = I.monoid_generators()[0]
sage: I._lift_map
```

(continues on next page)

(continued from previous page)

```

{0: {1: 1}}
sage: I.lift_on_basis(z0)
PBW['z']
sage: I._lift_map
{0: {1: 1}, 1: {PBW['z']: PBW['z']}}
sage: I.lift_on_basis(z0^3)
PBW['z']^3
sage: I._lift_map
{0: {1: 1}, 1: {PBW['z']: PBW['z']}}
sage: I._construct_next_degree()
sage: I._construct_next_degree()
sage: I._lift_map
{0: {1: 1},
 1: {PBW['z']: PBW['z']},
 2: {PBW['z']^2: PBW['z']^2},
 3: {PBW['z']^3: PBW['z']^3}}
sage: I.lift_on_basis(z0^3)
PBW['z']^3

```

```

>>> from sage.all import *
>>> g = lie_algebras.Heisenberg(QQ, Integer(3))
>>> U = g.pbw_basis()
>>> Z = U.center()
>>> I = Z.indices()
>>> z0 = I.monoid_generators()[Integer(0)]
>>> I._lift_map
{0: {1: 1}}
>>> I.lift_on_basis(z0)
PBW['z']
>>> I._lift_map
{0: {1: 1}, 1: {PBW['z']: PBW['z']}}
>>> I.lift_on_basis(z0**Integer(3))
PBW['z']^3
>>> I._lift_map
{0: {1: 1}, 1: {PBW['z']: PBW['z']}}
>>> I._construct_next_degree()
>>> I._construct_next_degree()
>>> I._lift_map
{0: {1: 1},
 1: {PBW['z']: PBW['z']},
 2: {PBW['z']^2: PBW['z']^2},
 3: {PBW['z']^3: PBW['z']^3}}
>>> I.lift_on_basis(z0**Integer(3))
PBW['z']^3

```

some_elements()

Return some elements of self.

EXAMPLES:

```

sage: g = lie_algebras.pwitt(GF(3), 3)
sage: U = g.pbw_basis()

```

(continues on next page)

(continued from previous page)

```
sage: Z = U.center()
sage: I = Z.indices()
sage: I.some_elements()
[1, Z[0], Z[1], Z[2], Z[0]*Z[1]*Z[2], Z[0]*Z[2]^4, Z[0]^4*Z[1]^3]
```

```
>>> from sage.all import *
>>> g = lie_algebras.pwitt(GF(Integer(3)), Integer(3))
>>> U = g.pbw_basis()
>>> Z = U.center()
>>> I = Z.indices()
>>> I.some_elements()
[1, Z[0], Z[1], Z[2], Z[0]*Z[1]*Z[2], Z[0]*Z[2]^4, Z[0]^4*Z[1]^3]
```

```
class sage.algebras.lie_algebras.center_uea.CenterUEA(g, UEA)
```

Bases: `CombinatorialFreeModule`

The center of a universal enveloping algebra.

Todo

Generalize this to be the centralizer of any set of the UEA.

Todo

For characteristic $p > 0$, implement the p -center of a simple Lie algebra. See, e.g.,

- Theorem 5.12 of [Motsak2006]
- <http://www.math.kobe-u.ac.jp/icms2006/icms2006-video/slides/059.pdf>

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['A', 2])
sage: U = g.pbw_basis()
sage: Z = U.center()
sage: B = Z.basis()
sage: it = iter(B)
sage: center_elts = [next(it) for _ in range(6)]; center_elts
[1, Z[0], Z[1], Z[0]^2, Z[0]*Z[1], Z[1]^2]
sage: elts = [U(v) for v in center_elts] # long time
sage: all(v * g == g * v for g in U.algebra_generators() for v in elts) # long_
↳time
True
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['A', Integer(2)])
>>> U = g.pbw_basis()
>>> Z = U.center()
>>> B = Z.basis()
>>> it = iter(B)
```

(continues on next page)

(continued from previous page)

```

>>> center_elts = [next(it) for _ in range(Integer(6))]; center_elts
[1, Z[0], Z[1], Z[0]^2, Z[0]*Z[1], Z[1]^2]
>>> elts = [U(v) for v in center_elts] # long time
>>> all(v * g == g * v for g in U.algebra_generators() for v in elts) # long time
True

```

The Heisenberg Lie algebra H_4 over a finite field; note the basis elements $b^p \in Z(U(H_4))$ for the basis elements $b \in H_4$:

```

sage: g = lie_algebras.Heisenberg(GF(3), 4)
sage: U = g.pbw_basis()
sage: Z = U.center()
sage: B = Z.basis()
sage: it = iter(B)
sage: center_elts = [next(it) for _ in range(12)]; center_elts
[1, Z[0], Z[0]^2, Z[0]^3, Z[1], Z[2], Z[3], Z[4], Z[5], Z[6], Z[7], Z[8]]
sage: elts = [U(v) for v in center_elts]; elts
[1, PBW['z'], PBW['z']^2, PBW['z']^3, PBW['p1']^3, PBW['p2']^3, PBW['p3']^3,
 PBW['p4']^3, PBW['q1']^3, PBW['q2']^3, PBW['q3']^3, PBW['q4']^3]
sage: all(v * g == g * v for g in U.algebra_generators() for v in elts)
True

```

```

>>> from sage.all import *
>>> g = lie_algebras.Heisenberg(GF(Integer(3)), Integer(4))
>>> U = g.pbw_basis()
>>> Z = U.center()
>>> B = Z.basis()
>>> it = iter(B)
>>> center_elts = [next(it) for _ in range(Integer(12))]; center_elts
[1, Z[0], Z[0]^2, Z[0]^3, Z[1], Z[2], Z[3], Z[4], Z[5], Z[6], Z[7], Z[8]]
>>> elts = [U(v) for v in center_elts]; elts
[1, PBW['z'], PBW['z']^2, PBW['z']^3, PBW['p1']^3, PBW['p2']^3, PBW['p3']^3,
 PBW['p4']^3, PBW['q1']^3, PBW['q2']^3, PBW['q3']^3, PBW['q4']^3]
>>> all(v * g == g * v for g in U.algebra_generators() for v in elts)
True

```

An example with a free 4-step nilpotent Lie algebras on 2 generators:

```

sage: L = LieAlgebra(QQ, 2, step=4); L
Free Nilpotent Lie algebra on 8 generators
(X_1, X_2, X_12, X_112, X_122, X_1112, X_1122, X_1222) over Rational Field
sage: U = L.pbw_basis()
sage: Z = U.center()
sage: it = iter(Z.basis())
sage: center_elts = [next(it) for _ in range(10)]; center_elts
[1, Z[0], Z[1], Z[2], Z[0]^2, Z[0]*Z[1], Z[0]*Z[2], Z[1]^2, Z[1]*Z[2], Z[2]^2]
sage: elts = [U(v) for v in center_elts]; elts
[1, PBW[(1, 1, 1, 2)], PBW[(1, 1, 2, 2)], PBW[(1, 2, 2, 2)], PBW[(1, 1, 1, 2)]^2,
 PBW[(1, 1, 1, 2)]*PBW[(1, 1, 2, 2)], PBW[(1, 1, 1, 2)]*PBW[(1, 2, 2, 2)],
 PBW[(1, 1, 2, 2)]^2, PBW[(1, 1, 2, 2)]*PBW[(1, 2, 2, 2)], PBW[(1, 2, 2, 2)]^2]
sage: all(v * g == g * v for g in U.algebra_generators() for v in elts)
True

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(2), step=Integer(4)); L
Free Nilpotent Lie algebra on 8 generators
(X_1, X_2, X_12, X_112, X_122, X_1112, X_1122, X_1222) over Rational Field
>>> U = L.pbw_basis()
>>> Z = U.center()
>>> it = iter(Z.basis())
>>> center_elts = [next(it) for _ in range(Integer(10))]; center_elts
[1, Z[0], Z[1], Z[2], Z[0]^2, Z[0]*Z[1], Z[0]*Z[2], Z[1]^2, Z[1]*Z[2], Z[2]^2]
>>> elts = [U(v) for v in center_elts]; elts
[1, PBW[(1, 1, 1, 2)], PBW[(1, 1, 2, 2)], PBW[(1, 2, 2, 2)], PBW[(1, 1, 1, 2)]^2,
PBW[(1, 1, 1, 2)]*PBW[(1, 1, 2, 2)], PBW[(1, 1, 1, 2)]*PBW[(1, 2, 2, 2)],
PBW[(1, 1, 2, 2)]^2, PBW[(1, 1, 2, 2)]*PBW[(1, 2, 2, 2)], PBW[(1, 2, 2, 2)]^2]
>>> all(v * g == g * v for g in U.algebra_generators() for v in elts)
True

```

Using the Engel Lie algebra:

```

sage: L.<X,Y,Z> = LieAlgebra(QQ, {'X','Y': {'Z': 1}}, nilpotent=True)
sage: U = L.pbw_basis()
sage: Z = U.center()
sage: it = iter(Z.basis())
sage: center_elts = [next(it) for _ in range(6)]; center_elts
[1, Z[0], Z[0]^2, Z[0]^3, Z[0]^4, Z[0]^5]
sage: elts = [U(v) for v in center_elts]; elts
[1, PBW['Z'], PBW['Z']^2, PBW['Z']^3, PBW['Z']^4, PBW['Z']^5]
sage: all(v * g == g * v for g in U.algebra_generators() for v in elts)
True

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, {'X','Y': {'Z': Integer(1)}}, nilpotent=True, names=('X',
↪ 'Y', 'Z',)); (X, Y, Z,) = L._first_ngens(3)
>>> U = L.pbw_basis()
>>> Z = U.center()
>>> it = iter(Z.basis())
>>> center_elts = [next(it) for _ in range(Integer(6))]; center_elts
[1, Z[0], Z[0]^2, Z[0]^3, Z[0]^4, Z[0]^5]
>>> elts = [U(v) for v in center_elts]; elts
[1, PBW['Z'], PBW['Z']^2, PBW['Z']^3, PBW['Z']^4, PBW['Z']^5]
>>> all(v * g == g * v for g in U.algebra_generators() for v in elts)
True

```

algebra_generators()

Return the algebra generators of self.

Warning

When the universal enveloping algebra is not known to have a finite generating set, the generating set will be the basis of self in a degree (weakly) increasing order indexed by $\mathbb{Z}_{\geq 0}$. In particular, the 0-th generator will be the multiplicative identity 1.

EXAMPLES:

```

sage: g = lie_algebras.Heisenberg(QQ, 3)
sage: U = g.pbw_basis()
sage: Z = U.center()
sage: Z.algebra_generators()[0]
1
sage: Z.algebra_generators()[1]
Z[0]

sage: g = LieAlgebra(QQ, cartan_type=['G', 2])
sage: U = g.pbw_basis()
sage: Z = U.center()
sage: Z.algebra_generators()
Finite family {0: Z[0], 1: Z[1]}

```

```

>>> from sage.all import *
>>> g = lie_algebras.Heisenberg(QQ, Integer(3))
>>> U = g.pbw_basis()
>>> Z = U.center()
>>> Z.algebra_generators()[Integer(0)]
1
>>> Z.algebra_generators()[Integer(1)]
Z[0]

>>> g = LieAlgebra(QQ, cartan_type=['G', Integer(2)])
>>> U = g.pbw_basis()
>>> Z = U.center()
>>> Z.algebra_generators()
Finite family {0: Z[0], 1: Z[1]}

```

ambient()

Return the ambient algebra of `self`.

EXAMPLES:

```

sage: g = LieAlgebra(GF(5), cartan_type=['A', 2])
sage: U = g.pbw_basis()
sage: Z = U.center()
sage: Z.ambient() is U
True

```

```

>>> from sage.all import *
>>> g = LieAlgebra(GF(Integer(5)), cartan_type=['A', Integer(2)])
>>> U = g.pbw_basis()
>>> Z = U.center()
>>> Z.ambient() is U
True

```

degree_on_basis(m)

Return the degree of the basis element indexed by `m` in `self`.

EXAMPLES:

```

sage: g = LieAlgebra(QQ, cartan_type=['E', 6])
sage: U = g.pbw_basis()
sage: Z = U.center()
sage: I = Z.indices()
sage: it = iter(I)
sage: supports = [next(it) for _ in range(10)]; supports
[1, Z[0], Z[0]^2, Z[1], Z[2], Z[0]^3, Z[0]*Z[1], Z[3], Z[0]*Z[2], Z[0]^4]
sage: [Z.degree_on_basis(m) for m in supports]
[0, 2, 4, 5, 6, 6, 7, 8, 8, 8]

```

```

>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['E', Integer(6)])
>>> U = g.pbw_basis()
>>> Z = U.center()
>>> I = Z.indices()
>>> it = iter(I)
>>> supports = [next(it) for _ in range(Integer(10))]; supports
[1, Z[0], Z[0]^2, Z[1], Z[2], Z[0]^3, Z[0]*Z[1], Z[3], Z[0]*Z[2], Z[0]^4]
>>> [Z.degree_on_basis(m) for m in supports]
[0, 2, 4, 5, 6, 6, 7, 8, 8, 8]

```

lift()

The lift map from self to the universal enveloping algebra.

EXAMPLES:

```

sage: g = LieAlgebra(QQ, cartan_type=['A', 1])
sage: U = g.pbw_basis()
sage: Z = U.center()
sage: gens = Z.algebra_generators()
sage: U(gens[0]^2 + gens[0])
4*PBW[alpha[1]]^2*PBW[-alpha[1]]^2
+ 2*PBW[alpha[1]]*PBW[alphacheck[1]]^2*PBW[-alpha[1]]
+ 1/4*PBW[alphacheck[1]]^4 - PBW[alphacheck[1]]^3
- 2*PBW[alpha[1]]*PBW[-alpha[1]] + 1/2*PBW[alphacheck[1]]^2
+ PBW[alphacheck[1]]
sage: U(-1/4*gens[0]) == U.casimir_element()
True

```

```

>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['A', Integer(1)])
>>> U = g.pbw_basis()
>>> Z = U.center()
>>> gens = Z.algebra_generators()
>>> U(gens[Integer(0)]**Integer(2) + gens[Integer(0)])
4*PBW[alpha[1]]^2*PBW[-alpha[1]]^2
+ 2*PBW[alpha[1]]*PBW[alphacheck[1]]^2*PBW[-alpha[1]]
+ 1/4*PBW[alphacheck[1]]^4 - PBW[alphacheck[1]]^3
- 2*PBW[alpha[1]]*PBW[-alpha[1]] + 1/2*PBW[alphacheck[1]]^2
+ PBW[alphacheck[1]]
>>> U(-Integer(1)/Integer(4)*gens[Integer(0)]) == U.casimir_element()
True

```

one_basis()

Return the basis index of 1 in self.

EXAMPLES:

```
sage: g = LieAlgebra(QQ['t'].fraction_field(), cartan_type=['B', 5])
sage: U = g.pbw_basis()
sage: Z = U.center()
sage: ob = Z.one_basis(); ob
1
sage: ob.parent() is Z.indices()
True
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ['t'].fraction_field(), cartan_type=['B', Integer(5)])
>>> U = g.pbw_basis()
>>> Z = U.center()
>>> ob = Z.one_basis(); ob
1
>>> ob.parent() is Z.indices()
True
```

product_on_basis(left, right)

Return the product of basis elements indexed by left and right.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['E', 6])
sage: U = g.pbw_basis()
sage: Z = U.center()
sage: mg = Z.indices().monoid_generators()
sage: Z.product_on_basis(mg[1]*mg[2], mg[0]*mg[1]^3*mg[2]*mg[3]^3)
Z[0]*Z[1]^4*Z[2]^2*Z[3]^3
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['E', Integer(6)])
>>> U = g.pbw_basis()
>>> Z = U.center()
>>> mg = Z.indices().monoid_generators()
>>> Z.product_on_basis(mg[Integer(1)]*mg[Integer(2)], mg[Integer(0)]*mg[Intege
↳ r(1)]**Integer(3)*mg[Integer(2)]*mg[Integer(3)]**Integer(3))
Z[0]*Z[1]^4*Z[2]^2*Z[3]^3
```

retract(elt)

The retraction map to self from the universal enveloping algebra.

Todo

Implement a version of this that checks if the leading term of `elt` is divisible by a product of all of the currently known generators in order to avoid constructing the full centralizer of larger degrees than needed.

EXAMPLES:

```

sage: g = lie_algebras.Heisenberg(QQ, 3)
sage: U = g.pbw_basis()
sage: Z = U.center()
sage: z0 = Z.algebra_generators()[1]; z0
Z[0]
sage: Z.retract(U(z0^2) - U(3*z0))
Z[0]^2 - 3*Z[0]

sage: g = LieAlgebra(QQ, cartan_type=['A', 2])
sage: U = g.pbw_basis()
sage: Z = U.center()
sage: z0, z1 = Z.algebra_generators()
sage: Z.retract(U(z0*z0) - U(z1)) # long time
Z[0]^2 - Z[1]
sage: zc = Z.retract(U.casimir_element()); zc
-1/3*Z[0]
sage: U(zc) == U.casimir_element()
True

```

```

>>> from sage.all import *
>>> g = lie_algebras.Heisenberg(QQ, Integer(3))
>>> U = g.pbw_basis()
>>> Z = U.center()
>>> z0 = Z.algebra_generators()[Integer(1)]; z0
Z[0]
>>> Z.retract(U(z0**Integer(2)) - U(Integer(3)*z0))
Z[0]^2 - 3*Z[0]

>>> g = LieAlgebra(QQ, cartan_type=['A', Integer(2)])
>>> U = g.pbw_basis()
>>> Z = U.center()
>>> z0, z1 = Z.algebra_generators()
>>> Z.retract(U(z0*z0) - U(z1)) # long time
Z[0]^2 - Z[1]
>>> zc = Z.retract(U.casimir_element()); zc
-1/3*Z[0]
>>> U(zc) == U.casimir_element()
True

```

class sage.algebras.lie_algebras.center_uea.**SimpleLieCenterIndices** (*center*)

Bases: [CenterIndices](#)

Set of basis indices for the center of a universal enveloping algebra of a simple Lie algebra.

For more information, see [CenterIndices](#).

9.2.7 Classical Lie Algebras

These are the Lie algebras corresponding to types A_n , B_n , C_n , and D_n . We also include support for the exceptional types $E_{6,7,8}$, F_4 , and G_2 in the Chevalley basis, and we give the matrix representation given in [HRT2000].

AUTHORS:

- Travis Scrimshaw (2013-05-03): Initial version
- Sebastian Oehms (2018-03-18): matrix method of the element class of ClassicalMatrixLieAlgebra added

- Travis Scrimshaw (2019-07-09): Implemented compact real form

```
class sage.algebras.lie_algebras.classical_lie_algebra.ClassicalMatrixLieAlgebra(R, ct, e,  
                                                                           f, h,  
                                                                           sparse=True)
```

Bases: *MatrixLieAlgebraFromAssociative*

A classical Lie algebra represented using matrices.

This means a classical Lie algebra given as a Lie algebra of matrices, with commutator as Lie bracket.

INPUT:

- *R* – the base ring
- *ct* – the finite Cartan type

EXAMPLES:

```
sage: lie_algebras.ClassicalMatrix(QQ, ['A', 4])  
Special linear Lie algebra of rank 5 over Rational Field  
sage: lie_algebras.ClassicalMatrix(QQ, CartanType(['B', 4]))  
Special orthogonal Lie algebra of rank 9 over Rational Field  
sage: lie_algebras.ClassicalMatrix(QQ, 'C4')  
Symplectic Lie algebra of rank 8 over Rational Field  
sage: lie_algebras.ClassicalMatrix(QQ, cartan_type=['D', 4])  
Special orthogonal Lie algebra of rank 8 over Rational Field
```

```
>>> from sage.all import *  
>>> lie_algebras.ClassicalMatrix(QQ, ['A', Integer(4)])  
Special linear Lie algebra of rank 5 over Rational Field  
>>> lie_algebras.ClassicalMatrix(QQ, CartanType(['B', Integer(4)]))  
Special orthogonal Lie algebra of rank 9 over Rational Field  
>>> lie_algebras.ClassicalMatrix(QQ, 'C4')  
Symplectic Lie algebra of rank 8 over Rational Field  
>>> lie_algebras.ClassicalMatrix(QQ, cartan_type=['D', Integer(4)])  
Special orthogonal Lie algebra of rank 8 over Rational Field
```

affine(*kac_moody=True*)

Return the affine (Kac-Moody) Lie algebra of self.

EXAMPLES:

```
sage: so5 = lie_algebras.so(QQ, 5, 'matrix')  
sage: so5  
Special orthogonal Lie algebra of rank 5 over Rational Field  
sage: so5.affine()  
Affine Special orthogonal Kac-Moody algebra of rank 5 over Rational Field  
sage: so5.affine(False)  
Affine Special orthogonal Lie algebra of rank 5 over Rational Field
```

```
>>> from sage.all import *  
>>> so5 = lie_algebras.so(QQ, Integer(5), 'matrix')  
>>> so5  
Special orthogonal Lie algebra of rank 5 over Rational Field  
>>> so5.affine()
```

(continues on next page)

(continued from previous page)

```
Affine Special orthogonal Kac-Moody algebra of rank 5 over Rational Field
>>> so5.affine(False)
Affine Special orthogonal Lie algebra of rank 5 over Rational Field
```

basis()

Return a basis of self.

EXAMPLES:

```
sage: M = LieAlgebra(ZZ, cartan_type=['A',2], representation='matrix')
sage: list(M.basis())
[
[ 1  0  0] [0 1 0] [0 0 1] [0 0 0] [ 0  0  0] [0 0 0] [0 0 0]
[ 0  0  0] [0 0 0] [0 0 0] [1 0 0] [ 0  1  0] [0 0 1] [0 0 0]
[ 0  0 -1], [0 0 0], [0 0 0], [0 0 0], [ 0  0 -1], [0 0 0], [1 0 0],

[0 0 0]
[0 0 0]
[0 1 0]
]
```

```
>>> from sage.all import *
>>> M = LieAlgebra(ZZ, cartan_type=['A',Integer(2)], representation='matrix')
>>> list(M.basis())
[
[ 1  0  0] [0 1 0] [0 0 1] [0 0 0] [ 0  0  0] [0 0 0] [0 0 0]
[ 0  0  0] [0 0 0] [0 0 0] [1 0 0] [ 0  1  0] [0 0 1] [0 0 0]
[ 0  0 -1], [0 0 0], [0 0 0], [0 0 0], [ 0  0 -1], [0 0 0], [1 0 0],
<BLANKLINE>
[0 0 0]
[0 0 0]
[0 1 0]
]
```

Sparse version:

```
sage: e6 = LieAlgebra(QQ, cartan_type=['E',6], representation='matrix')
sage: len(e6.basis()) # long time
78
```

```
>>> from sage.all import *
>>> e6 = LieAlgebra(QQ, cartan_type=['E',Integer(6)], representation='matrix')
>>> len(e6.basis()) # long time
78
```

cartan_type()

Return the Cartan type of self.

EXAMPLES:

```
sage: g = lie_algebras.sl(QQ, 3, representation='matrix')
sage: g.cartan_type()
['A', 2]
```

```
>>> from sage.all import *
>>> g = lie_algebras.sl(QQ, Integer(3), representation='matrix')
>>> g.cartan_type()
['A', 2]
```

e(*i*)Return the generator e_i .

EXAMPLES:

```
sage: g = lie_algebras.sl(QQ, 3, representation='matrix')
sage: g.e(2)
[0 0 0]
[0 0 1]
[0 0 0]
```

```
>>> from sage.all import *
>>> g = lie_algebras.sl(QQ, Integer(3), representation='matrix')
>>> g.e(Integer(2))
[0 0 0]
[0 0 1]
[0 0 0]
```

epsilon(*i*, *h*)Return the action of the functional $\varepsilon_i: \mathfrak{h} \rightarrow R$, where R is the base ring of `self`, on the element *h*.

EXAMPLES:

```
sage: g = lie_algebras.sl(QQ, 3, representation='matrix')
sage: g.epsilon(1, g.h(1))
1
sage: g.epsilon(2, g.h(1))
-1
sage: g.epsilon(3, g.h(1))
0
```

```
>>> from sage.all import *
>>> g = lie_algebras.sl(QQ, Integer(3), representation='matrix')
>>> g.epsilon(Integer(1), g.h(Integer(1)))
1
>>> g.epsilon(Integer(2), g.h(Integer(1)))
-1
>>> g.epsilon(Integer(3), g.h(Integer(1)))
0
```

f(*i*)Return the generator f_i .

EXAMPLES:

```
sage: g = lie_algebras.sl(QQ, 3, representation='matrix')
sage: g.f(2)
[0 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 0 0]
[0 1 0]
```

```
>>> from sage.all import *
>>> g = lie_algebras.sl(QQ, Integer(3), representation='matrix')
>>> g.f(Integer(2))
[0 0 0]
[0 0 0]
[0 1 0]
```

h(i)Return the generator h_i .

EXAMPLES:

```
sage: g = lie_algebras.sl(QQ, 3, representation='matrix')
sage: g.h(2)
[ 0  0  0]
[ 0  1  0]
[ 0  0 -1]
```

```
>>> from sage.all import *
>>> g = lie_algebras.sl(QQ, Integer(3), representation='matrix')
>>> g.h(Integer(2))
[ 0  0  0]
[ 0  1  0]
[ 0  0 -1]
```

highest_root_basis_elt (*pos=True*)Return the basis element corresponding to the highest root θ . If *pos* is *True*, then returns e_θ , otherwise it returns f_θ .

EXAMPLES:

```
sage: g = lie_algebras.sl(QQ, 3, representation='matrix')
sage: g.highest_root_basis_elt()
[0 0 1]
[0 0 0]
[0 0 0]
```

```
>>> from sage.all import *
>>> g = lie_algebras.sl(QQ, Integer(3), representation='matrix')
>>> g.highest_root_basis_elt()
[0 0 1]
[0 0 0]
[0 0 0]
```

index_set ()Return the *index_set* of *self*.

EXAMPLES:

```
sage: g = lie_algebras.sl(QQ, 3, representation='matrix')
sage: g.index_set()
(1, 2)
```

```
>>> from sage.all import *
>>> g = lie_algebras.sl(QQ, Integer(3), representation='matrix')
>>> g.index_set()
(1, 2)
```

simple_root (*i*, *h*)

Return the action of the simple root $\alpha_i: \mathfrak{h} \rightarrow R$, where R is the base ring of `self`, on the element *h*.

EXAMPLES:

```
sage: g = lie_algebras.sl(QQ, 3, representation='matrix')
sage: g.simple_root(1, g.h(1))
2
sage: g.simple_root(1, g.h(2))
-1
```

```
>>> from sage.all import *
>>> g = lie_algebras.sl(QQ, Integer(3), representation='matrix')
>>> g.simple_root(Integer(1), g.h(Integer(1)))
2
>>> g.simple_root(Integer(1), g.h(Integer(2)))
-1
```

```
class sage.algebras.lie_algebras.classical_lie_algebra.ExceptionalMatrixLieAlgebra (R,
                                                                                      car-
                                                                                      tan_type,
                                                                                      e, f,
                                                                                      h=None,
                                                                                      sparse=False)
```

Bases: *ClassicalMatrixLieAlgebra*

A matrix Lie algebra of exceptional type.

```
class sage.algebras.lie_algebras.classical_lie_algebra.LieAlgebraChevalleyBasis (R, car-
                                                                                      tan_type)
```

Bases: *LieAlgebraWithStructureCoefficients*

A simple finite dimensional Lie algebra in the Chevalley basis.

Let L be a simple (complex) Lie algebra with roots Φ , then the Chevalley basis is given by e_α for all $\alpha \in \Phi$ and $h_{\alpha_i} := h_i$ where α_i is a simple root subject. These generators are subject to the relations:

$$\begin{aligned} [h_i, h_j] &= 0, \\ [h_i, e_\beta] &= A_{\alpha_i, \beta} e_\beta, \\ [e_\beta, e_{-\beta}] &= \sum_i A_{\beta, \alpha_i} h_i, \\ [e_\beta, e_\gamma] &= \begin{cases} N_{\beta, \gamma} e_{\beta+\gamma} & \beta + \gamma \in \Phi, \\ 0 & \text{otherwise,} \end{cases} \end{aligned}$$

where $A_{\alpha, \beta} = \frac{2(\alpha, \beta)}{(\alpha, \alpha)}$ and $N_{\alpha, \beta}$ is the maximum such that $\alpha - N_{\alpha, \beta} \beta \in \Phi$.

For computing the signs of the coefficients, see Section 3 of [CMT2003].

See also

For simply-laced types, an alternative construction using an asymmetry function is given by `LieAlgebraChevalleyBasis_simply_laced`.

affine(*kac_moody=True*)

Return the affine Lie algebra of `self`.

EXAMPLES:

```
sage: sp6 = lie_algebras.sp(QQ, 6)
sage: sp6
Lie algebra of ['C', 3] in the Chevalley basis
sage: sp6.affine()
Affine Kac-Moody algebra of ['C', 3] in the Chevalley basis

sage: L = LieAlgebra(QQ, cartan_type=['A', 3], epsilon=[(1, 2), (3, 2)])
sage: L.affine(False)
Affine Lie algebra of ['A', 3] in the Chevalley basis
sage: L.affine(True)
Affine Kac-Moody algebra of ['A', 3] in the Chevalley basis
```

```
>>> from sage.all import *
>>> sp6 = lie_algebras.sp(QQ, Integer(6))
>>> sp6
Lie algebra of ['C', 3] in the Chevalley basis
>>> sp6.affine()
Affine Kac-Moody algebra of ['C', 3] in the Chevalley basis

>>> L = LieAlgebra(QQ, cartan_type=['A', Integer(3)], epsilon=[(Integer(1),
↳ Integer(2)), (Integer(3), Integer(2))])
>>> L.affine(False)
Affine Lie algebra of ['A', 3] in the Chevalley basis
>>> L.affine(True)
Affine Kac-Moody algebra of ['A', 3] in the Chevalley basis
```

degree_on_basis(*m*)

Return the degree of the basis element indexed by *m*.

EXAMPLES:

```
sage: L = LieAlgebra(QQ, cartan_type=['G', 2])
sage: [L.degree_on_basis(m) for m in L.basis().keys()]
[alpha[2], alpha[1], alpha[1] + alpha[2],
 2*alpha[1] + alpha[2], 3*alpha[1] + alpha[2],
 3*alpha[1] + 2*alpha[2],
 0, 0,
 -alpha[2], -alpha[1], -alpha[1] - alpha[2],
 -2*alpha[1] - alpha[2], -3*alpha[1] - alpha[2],
 -3*alpha[1] - 2*alpha[2]]
```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, cartan_type=['G', Integer(2)])
>>> [L.degree_on_basis(m) for m in L.basis().keys()]
[alpha[2], alpha[1], alpha[1] + alpha[2],
 2*alpha[1] + alpha[2], 3*alpha[1] + alpha[2],
 3*alpha[1] + 2*alpha[2],
 0, 0,
 -alpha[2], -alpha[1], -alpha[1] - alpha[2],
 -2*alpha[1] - alpha[2], -3*alpha[1] - alpha[2],
 -3*alpha[1] - 2*alpha[2]]

```

gens()

Return the generators of `self` in the order of e_i , f_i , and h_i .

EXAMPLES:

```

sage: L = LieAlgebra(QQ, cartan_type=['A', 2])
sage: L.gens()
(E[alpha[1]], E[alpha[2]], E[-alpha[1]], E[-alpha[2]], h1, h2)

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, cartan_type=['A', Integer(2)])
>>> L.gens()
(E[alpha[1]], E[alpha[2]], E[-alpha[1]], E[-alpha[2]], h1, h2)

```

highest_root_basis_elt (*pos=True*)

Return the basis element corresponding to the highest root θ .

INPUT:

- `pos` – boolean (default: `True`); if `True`, then return e_θ , otherwise return f_θ

EXAMPLES:

```

sage: L = LieAlgebra(QQ, cartan_type=['A', 2])
sage: L.highest_root_basis_elt()
E[alpha[1] + alpha[2]]
sage: L.highest_root_basis_elt(False)
E[-alpha[1] - alpha[2]]

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, cartan_type=['A', Integer(2)])
>>> L.highest_root_basis_elt()
E[alpha[1] + alpha[2]]
>>> L.highest_root_basis_elt(False)
E[-alpha[1] - alpha[2]]

```

indices_to_positive_roots_map()

Return the map from indices to positive roots.

EXAMPLES:

```

sage: L = LieAlgebra(QQ, cartan_type=['A', 2])
sage: L.indices_to_positive_roots_map()
{1: alpha[1], 2: alpha[2], 3: alpha[1] + alpha[2]}

```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, cartan_type=['A', Integer(2)])
>>> L.indices_to_positive_roots_map()
{1: alpha[1], 2: alpha[2], 3: alpha[1] + alpha[2]}
```

killling_form(x, y)

Return the Killing form on x and y , where x and y are two elements of `self`.

The Killing form is defined as

$$\langle x | y \rangle = \text{tr}(\text{ad}_x \circ \text{ad}_y).$$

EXAMPLES:

```
sage: L = LieAlgebra(QQ, cartan_type=['A', 2])
sage: L.killing_form(L.an_element(), L.an_element())
36
sage: B = L.basis()
sage: matrix([[L.killing_form(a, b) for a in B] for b in B])
[ 0  0  0  0  0  6  0  0]
[ 0  0  0  0  0  0  6  0]
[ 0  0  0  0  0  0  0  6]
[ 0  0  0 12 -6  0  0  0]
[ 0  0  0 -6 12  0  0  0]
[ 6  0  0  0  0  0  0  0]
[ 0  6  0  0  0  0  0  0]
[ 0  0  6  0  0  0  0  0]
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, cartan_type=['A', Integer(2)])
>>> L.killing_form(L.an_element(), L.an_element())
36
>>> B = L.basis()
>>> matrix([[L.killing_form(a, b) for a in B] for b in B])
[ 0  0  0  0  0  6  0  0]
[ 0  0  0  0  0  0  6  0]
[ 0  0  0  0  0  0  0  6]
[ 0  0  0 12 -6  0  0  0]
[ 0  0  0 -6 12  0  0  0]
[ 6  0  0  0  0  0  0  0]
[ 0  6  0  0  0  0  0  0]
[ 0  0  6  0  0  0  0  0]
```

killling_form_matrix()

Return the matrix of the Killing form of `self`.

The rows and the columns of this matrix are indexed by the elements of the basis of `self` (in the order provided by `basis()`).

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['A', 2])
sage: g.killing_form_matrix()
[ 0  0  0  0  0  6  0  0]
```

(continues on next page)

(continued from previous page)

```
[ 0  0  0  0  0  0  6  0]
[ 0  0  0  0  0  0  0  6]
[ 0  0  0 12 -6  0  0  0]
[ 0  0  0 -6 12  0  0  0]
[ 6  0  0  0  0  0  0  0]
[ 0  6  0  0  0  0  0  0]
[ 0  0  6  0  0  0  0  0]
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['A', Integer(2)])
>>> g.killing_form_matrix()
[ 0  0  0  0  0  6  0  0]
[ 0  0  0  0  0  0  6  0]
[ 0  0  0  0  0  0  0  6]
[ 0  0  0 12 -6  0  0  0]
[ 0  0  0 -6 12  0  0  0]
[ 6  0  0  0  0  0  0  0]
[ 0  6  0  0  0  0  0  0]
[ 0  0  6  0  0  0  0  0]
```

lie_algebra_generators (*str_keys=False*)

Return the Chevalley Lie algebra generators of *self*.

INPUT:

- *str_keys* – boolean (default: *False*); set to *True* to have the indices indexed by strings instead of simple (co)roots

EXAMPLES:

```
sage: L = LieAlgebra(QQ, cartan_type=['A', 1])
sage: L.lie_algebra_generators()
Finite family {alpha[1]: E[alpha[1]], -alpha[1]: E[-alpha[1]], alphacheck[1]: ↪
↪ h1}
sage: L.lie_algebra_generators(True)
Finite family {'e1': E[alpha[1]], 'f1': E[-alpha[1]], 'h1': h1}
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, cartan_type=['A', Integer(1)])
>>> L.lie_algebra_generators()
Finite family {alpha[1]: E[alpha[1]], -alpha[1]: E[-alpha[1]], alphacheck[1]: ↪
↪ h1}
>>> L.lie_algebra_generators(True)
Finite family {'e1': E[alpha[1]], 'f1': E[-alpha[1]], 'h1': h1}
```

class sage.algebras.lie_algebras.classical_lie_algebra.LieAlgebraChevalleyBasis_simply_laced(*R*,
car-
tan_type,
ep-
silon)

Bases: *LieAlgebraChevalleyBasis*

A finite dimensional simply-laced Lie algebra in the Chevalley basis with structure coefficients given by an orientation of the Dynkin diagram.

We follow Chapter 7.7 of [Ka1990], where the structure coefficients are given by an *asymmetry function* defined by $\varepsilon(\alpha_i, \alpha_j) = -1$ if there is an arrow $i \rightarrow j$ in the Dynkin quiver (an orientation of the Dynkin diagram). However we twist E_α by $\text{sign}(\alpha)$ so that $F_i = E_{-\alpha_i}$ rather than its negative.

EXAMPLES:

```
sage: L = LieAlgebra(QQ, cartan_type=['A', 2], epsilon=[(2, 1)])
sage: L.e(1).bracket(L.e(2))
E[alpha[1] + alpha[2]]

sage: L = LieAlgebra(QQ, cartan_type=['A', 2], epsilon=[(1, 2)])
sage: L.e(1).bracket(L.e(2))
-E[alpha[1] + alpha[2]]
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, cartan_type=['A', Integer(2)], epsilon=[(Integer(2), Integer(1))])
>>> L.e(Integer(1)).bracket(L.e(Integer(2)))
E[alpha[1] + alpha[2]]

>>> L = LieAlgebra(QQ, cartan_type=['A', Integer(2)], epsilon=[(Integer(1), Integer(2))])
>>> L.e(Integer(1)).bracket(L.e(Integer(2)))
-E[alpha[1] + alpha[2]]
```

asymmetry_function()

Return the asymmetry function of `self`.

An *asymmetry function* is a function $\varepsilon : Q \times Q \rightarrow \{1, -1\}$ that satisfies the following properties:

1. $\varepsilon(\alpha, \alpha) = (-1)^{(\alpha|\alpha)/2}$
2. bimultiplicativity $\varepsilon(\alpha + \alpha', \beta) = \varepsilon(\alpha, \beta)\varepsilon(\alpha', \beta)$ and $\varepsilon(\alpha, \beta + \beta') = \varepsilon(\alpha, \beta)\varepsilon(\alpha, \beta')$,

where $(\alpha|\beta)$ is the symmetric bilinear form on Q given by the Cartan matrix. Some consequences of these properties are that $\varepsilon(\alpha, 0) = \varepsilon(0, \beta) = 1$ and $\text{varepsilon}(\alpha, \beta)\varepsilon(\beta, \alpha) = (-1)^{(\alpha|\beta)}$.

OUTPUT:

The asymmetry function as a dict consisting of pairs of all of the roots of Q and 0.

EXAMPLES:

```
sage: L = LieAlgebra(QQ, cartan_type=['A', 2], epsilon=[(2, 1)])
sage: ep = L.asymmetry_function()
sage: al = L.cartan_type().root_system().root_lattice().simple_roots()
sage: ep[al[1], al[2]]
1
sage: ep[al[2], al[1]]
-1

sage: L = LieAlgebra(QQ, cartan_type=['A', 2], epsilon=[(1, 2)])
sage: ep = L.asymmetry_function()
sage: al = L.cartan_type().root_system().root_lattice().simple_roots()
sage: ep[al[1], al[2]]
-1
```

(continues on next page)

(continued from previous page)

```
sage: ep[al[2],al[1]]
1
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, cartan_type=['A',Integer(2)], epsilon=[(Integer(2),
↪Integer(1))])
>>> ep = L.asymmetry_function()
>>> al = L.cartan_type().root_system().root_lattice().simple_roots()
>>> ep[al[Integer(1)], al[Integer(2)]]
1
>>> ep[al[Integer(2)], al[Integer(1)]]
-1

>>> L = LieAlgebra(QQ, cartan_type=['A',Integer(2)], epsilon=[(Integer(1),
↪Integer(2))])
>>> ep = L.asymmetry_function()
>>> al = L.cartan_type().root_system().root_lattice().simple_roots()
>>> ep[al[Integer(1)], al[Integer(2)]]
-1
>>> ep[al[Integer(2)], al[Integer(1)]]
1
```

```
class sage.algebras.lie_algebras.classical_lie_algebra.MatrixCompactRealForm(R,
                                                                    cartan_type)
```

Bases: *FinitelyGeneratedLieAlgebra*

The compact real form of a matrix Lie algebra.

Let L be a classical (i.e., type $ABCD$) Lie algebra over $\mathbf{R}$ given as matrices that is invariant under matrix transpose (i.e., $X^T \in L$ for all $X \in L$). Then we can perform the *Cartan decomposition* of L by $L = K \oplus S$, where K (resp. S) is the set of skew-symmetric (resp. symmetric) matrices in L . Then the Lie algebra $U = K \oplus iS$ is an $\mathbf{R}$ -subspace of the complexification of L that is closed under commutators and has skew-hermitian matrices. Hence, the Killing form is negative definitive (i.e., U is a compact Lie algebra), and thus U is the complex real form of the complexification of L .

EXAMPLES:

```
sage: U = LieAlgebra(QQ, cartan_type=['A',1], representation="compact real")
sage: list(U.basis())
[
[ 0  1] [ i  0] [ 0 i]
[-1  0], [ 0 -i], [i 0]
]
sage: U.killing_form_matrix()
[-8  0  0]
[ 0 -8  0]
[ 0  0 -8]
```

```
>>> from sage.all import *
>>> U = LieAlgebra(QQ, cartan_type=['A',Integer(1)], representation="compact real
↪")
>>> list(U.basis())
[
```

(continues on next page)

(continued from previous page)

```

[ 0  1] [ i  0] [0 i]
[-1  0], [ 0 -i], [i 0]
]
>>> U.killing_form_matrix()
[-8  0  0]
[ 0 -8  0]
[ 0  0 -8]

```

Computations are only (currently) possible if this is defined over a field:

```

sage: U = LieAlgebra(ZZ, cartan_type=['A',1], representation="compact real")
sage: list(U.basis())
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer

```

```

>>> from sage.all import *
>>> U = LieAlgebra(ZZ, cartan_type=['A',Integer(1)], representation="compact real
↪")
>>> list(U.basis())
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer

```

class `Element` (*parent, real, imag*)

Bases: `Element`

An element of a matrix Lie algebra in its compact real form.

monomial_coefficients (*copy=False*)

Return the monomial coefficients of `self`.

EXAMPLES:

```

sage: L = LieAlgebra(QQ, cartan_type=['C',3], representation="compact real
↪")
sage: B = L.basis()
sage: x = L.sum(i*B[i] for i in range(len(B)))
sage: x.monomial_coefficients() == {i: i for i in range(1,len(B))}
True

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, cartan_type=['C',Integer(3)], representation=
↪"compact real")
>>> B = L.basis()
>>> x = L.sum(i*B[i] for i in range(len(B)))
>>> x.monomial_coefficients() == {i: i for i in range(Integer(1),len(B))}
True

```

basis ()

Compute a basis of `self`.

EXAMPLES:

```

sage: L = LieAlgebra(QQ, cartan_type=['B',2], representation="compact real")
sage: list(L.basis())
[
[ 0  1  0  0  0] [ 0  0  0  1  0] [ 0  0  0  0  1] [ 0  0  0  0  0]
[-1  0  0  0  0] [ 0  0 -1  0  0] [ 0  0  0  0  0] [ 0  0  0  0  1]
[ 0  0  0  1  0] [ 0  1  0  0  0] [ 0  0  0  0  1] [ 0  0  0  0  0]
[ 0  0 -1  0  0] [-1  0  0  0  0] [ 0  0  0  0  0] [ 0  0  0  0  1]
[ 0  0  0  0  0], [ 0  0  0  0  0], [-1  0 -1  0  0], [ 0 -1  0 -1  0],

[ i  0  0  0  0] [ 0  i  0  0  0] [ 0  0  0  i  0] [ 0  0  0  0  i]
[ 0  0  0  0  0] [ i  0  0  0  0] [ 0  0 -i  0  0] [ 0  0  0  0  0]
[ 0  0 -i  0  0] [ 0  0  0 -i  0] [ 0 -i  0  0  0] [ 0  0  0  0 -i]
[ 0  0  0  0  0] [ 0  0 -i  0  0] [ i  0  0  0  0] [ 0  0  0  0  0]
[ 0  0  0  0  0], [ 0  0  0  0  0], [ 0  0  0  0  0], [ i  0 -i  0  0],

[ 0  0  0  0  0] [ 0  0  0  0  0]
[ 0  i  0  0  0] [ 0  0  0  0  i]
[ 0  0  0  0  0] [ 0  0  0  0  0]
[ 0  0  0 -i  0] [ 0  0  0  0 -i]
[ 0  0  0  0  0], [ 0  i  0 -i  0]
]

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, cartan_type=['B',Integer(2)], representation="compact_
↳real")
>>> list(L.basis())
[
[ 0  1  0  0  0] [ 0  0  0  1  0] [ 0  0  0  0  1] [ 0  0  0  0  0]
[-1  0  0  0  0] [ 0  0 -1  0  0] [ 0  0  0  0  0] [ 0  0  0  0  1]
[ 0  0  0  1  0] [ 0  1  0  0  0] [ 0  0  0  0  1] [ 0  0  0  0  0]
[ 0  0 -1  0  0] [-1  0  0  0  0] [ 0  0  0  0  0] [ 0  0  0  0  1]
[ 0  0  0  0  0], [ 0  0  0  0  0], [-1  0 -1  0  0], [ 0 -1  0 -1  0],

<BLANKLINE>
[ i  0  0  0  0] [ 0  i  0  0  0] [ 0  0  0  i  0] [ 0  0  0  0  i]
[ 0  0  0  0  0] [ i  0  0  0  0] [ 0  0 -i  0  0] [ 0  0  0  0  0]
[ 0  0 -i  0  0] [ 0  0  0 -i  0] [ 0 -i  0  0  0] [ 0  0  0  0 -i]
[ 0  0  0  0  0] [ 0  0 -i  0  0] [ i  0  0  0  0] [ 0  0  0  0  0]
[ 0  0  0  0  0], [ 0  0  0  0  0], [ 0  0  0  0  0], [ i  0 -i  0  0],

<BLANKLINE>
[ 0  0  0  0  0] [ 0  0  0  0  0]
[ 0  i  0  0  0] [ 0  0  0  0  i]
[ 0  0  0  0  0] [ 0  0  0  0  0]
[ 0  0  0 -i  0] [ 0  0  0  0 -i]
[ 0  0  0  0  0], [ 0  i  0 -i  0]
]

```

monomial (*i*)

Return the monomial indexed by *i*.

EXAMPLES:

```

sage: L = LieAlgebra(QQ, cartan_type=['A',3], representation="compact real")
sage: L.monomial(0)

```

(continues on next page)

(continued from previous page)

```
[ 0  1  0  0]
[-1  0  0  0]
[ 0  0  0  0]
[ 0  0  0  0]
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, cartan_type=['A',Integer(3)], representation="compact_
↳real")
>>> L.monomial(Integer(0))
[ 0  1  0  0]
[-1  0  0  0]
[ 0  0  0  0]
[ 0  0  0  0]
```

term(*i*, *c=None*)Return the term indexed by *i* with coefficient *c*.

EXAMPLES:

```
sage: L = LieAlgebra(QQ, cartan_type=['C',3], representation="compact real")
sage: L.term(4, 7/2)
[ 0  0  0  0  0  7/2]
[ 0  0  0  0  0  0]
[ 0  0  0  7/2  0  0]
[ 0  0 -7/2  0  0  0]
[ 0  0  0  0  0  0]
[-7/2  0  0  0  0  0]
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, cartan_type=['C',Integer(3)], representation="compact_
↳real")
>>> L.term(Integer(4), Integer(7)/Integer(2))
[ 0  0  0  0  0  7/2]
[ 0  0  0  0  0  0]
[ 0  0  0  7/2  0  0]
[ 0  0 -7/2  0  0  0]
[ 0  0  0  0  0  0]
[-7/2  0  0  0  0  0]
```

zero()

Return the element 0.

EXAMPLES:

```
sage: L = LieAlgebra(QQ, cartan_type=['D',4], representation="compact real")
sage: L.zero()
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, cartan_type=['D', Integer(4)], representation="compact_
↳real")
>>> L.zero()
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
```

class sage.algebras.lie_algebras.classical_lie_algebra.**e6**(R)

Bases: *ExceptionalMatrixLieAlgebra*

The matrix Lie algebra $\mathfrak{e}_6$.

The simple Lie algebra $\mathfrak{e}_6$ of type E_6 . The matrix representation is given following [HRT2000].

class sage.algebras.lie_algebras.classical_lie_algebra.**e7**(R)

Bases: *ExceptionalMatrixLieAlgebra*

The matrix Lie algebra $\mathfrak{e}_7$.

The simple Lie algebra $\mathfrak{e}_7$ of type E_7 . The matrix representation is given following [HRT2000].

class sage.algebras.lie_algebras.classical_lie_algebra.**e8**(R)

Bases: *ExceptionalMatrixLieAlgebra*

The matrix Lie algebra $\mathfrak{e}_8$.

The simple Lie algebra $\mathfrak{e}_8$ of type E_8 built from the adjoint representation in the Chevalley basis.

basis()

Return a basis of self.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['E', 8], representation='matrix') #↳
↳long time
sage: len(g.basis()) # long time
248
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['E', Integer(8)], representation='matrix
↳') # long time
>>> len(g.basis()) # long time
248
```

class sage.algebras.lie_algebras.classical_lie_algebra.**f4**(R)

Bases: *ExceptionalMatrixLieAlgebra*

The matrix Lie algebra $\mathfrak{f}_4$.

The simple Lie algebra $\mathfrak{f}_4$ of type F_4 . The matrix representation is given following [HRT2000] but indexed in the reversed order (i.e., interchange 1 with 4 and 2 with 3).

```
class sage.algebras.lie_algebras.classical_lie_algebra.g2(R)
```

Bases: *ExceptionalMatrixLieAlgebra*

The matrix Lie algebra $\mathfrak{g}_2$.

The simple Lie algebra $\mathfrak{g}_2$ of type G_2 . The matrix representation is given following [HRT2000].

```
class sage.algebras.lie_algebras.classical_lie_algebra.g1(R, n)
```

Bases: *MatrixLieAlgebraFromAssociative*

The matrix Lie algebra $\mathfrak{gl}_n$.

The Lie algebra $\mathfrak{gl}_n$ which consists of all $n \times n$ matrices.

INPUT:

- R – the base ring
- n – the size of the matrix

```
class Element
```

Bases: *Element*

```
monomial_coefficients(copy=True)
```

Return the monomial coefficients of `self`.

EXAMPLES:

```
sage: gl4 = lie_algebras.gl(QQ, 4)
sage: x = gl4.monomial('E_2_1') + 3*gl4.monomial('E_0_3')
sage: x.monomial_coefficients()
{'E_0_3': 3, 'E_2_1': 1}
```

```
>>> from sage.all import *
>>> gl4 = lie_algebras.gl(QQ, Integer(4))
>>> x = gl4.monomial('E_2_1') + Integer(3)*gl4.monomial('E_0_3')
>>> x.monomial_coefficients()
{'E_0_3': 3, 'E_2_1': 1}
```

```
basis()
```

Return the basis of `self`.

EXAMPLES:

```
sage: g = lie_algebras.gl(QQ, 2)
sage: tuple(g.basis())
(
 [1 0]  [0 1]  [0 0]  [0 0]
 [0 0], [0 0], [1 0], [0 1]
)
```

```
>>> from sage.all import *
>>> g = lie_algebras.gl(QQ, Integer(2))
>>> tuple(g.basis())
(
```

(continues on next page)

(continued from previous page)

```
[1 0] [0 1] [0 0] [0 0]
[0 0], [0 0], [1 0], [0 1]
)
```

killling_form(*x*, *y*)Return the Killing form on *x* and *y*.The Killing form on $\mathfrak{gl}_n$ is:

$$\langle x | y \rangle = 2n\text{tr}(xy) - 2\text{tr}(x)\text{tr}(y).$$

EXAMPLES:

```
sage: g = lie_algebras.gl(QQ, 4)
sage: x = g.an_element()
sage: y = g.gens()[1]
sage: g.killing_form(x, y)
8
```

```
>>> from sage.all import *
>>> g = lie_algebras.gl(QQ, Integer(4))
>>> x = g.an_element()
>>> y = g.gens()[Integer(1)]
>>> g.killing_form(x, y)
8
```

monomial(*i*)Return the basis element indexed by *i*.

INPUT:

- *i* – an element of the index set

EXAMPLES:

```
sage: gl4 = lie_algebras.gl(QQ, 4)
sage: gl4.monomial('E_2_1')
[0 0 0 0]
[0 0 0 0]
[0 1 0 0]
[0 0 0 0]
sage: gl4.monomial((2,1))
[0 0 0 0]
[0 0 0 0]
[0 1 0 0]
[0 0 0 0]
```

```
>>> from sage.all import *
>>> gl4 = lie_algebras.gl(QQ, Integer(4))
>>> gl4.monomial('E_2_1')
[0 0 0 0]
[0 0 0 0]
[0 1 0 0]
[0 0 0 0]
```

(continues on next page)

(continued from previous page)

```
>>> gl4.monomial((Integer(2), Integer(1)))
[0 0 0 0]
[0 0 0 0]
[0 1 0 0]
[0 0 0 0]
```

class sage.algebras.lie_algebras.classical_lie_algebra.**sl**(R, n)

Bases: *ClassicalMatrixLieAlgebra*

The matrix Lie algebra $\mathfrak{sl}_n$.

The Lie algebra $\mathfrak{sl}_n$, which consists of all $n \times n$ matrices with trace 0. This is the Lie algebra of type A_{n-1} .

killling_form(x, y)

Return the Killing form on x and y .

The Killing form on $\mathfrak{sl}_n$ is:

$$\langle x | y \rangle = 2n \operatorname{tr}(xy).$$

EXAMPLES:

```
sage: g = lie_algebras.sl(QQ, 5, representation='matrix')
sage: x = g.an_element()
sage: y = g.lie_algebra_generators()['e1']
sage: g.killing_form(x, y)
10
```

```
>>> from sage.all import *
>>> g = lie_algebras.sl(QQ, Integer(5), representation='matrix')
>>> x = g.an_element()
>>> y = g.lie_algebra_generators()['e1']
>>> g.killing_form(x, y)
10
```

simple_root(i, h)

Return the action of the simple root $\alpha_i: \mathfrak{h} \rightarrow R$, where R is the base ring of `self`, on the element j .

EXAMPLES:

```
sage: g = lie_algebras.sl(QQ, 5, representation='matrix')
sage: matrix([[g.simple_root(i, g.h(j)) for i in g.index_set()] for j in g.
↪index_set()])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -1]
[ 0  0 -1  2]
```

```
>>> from sage.all import *
>>> g = lie_algebras.sl(QQ, Integer(5), representation='matrix')
>>> matrix([[g.simple_root(i, g.h(j)) for i in g.index_set()] for j in g.
↪index_set()])
[ 2 -1  0  0]
[-1  2 -1  0]
```

(continues on next page)

(continued from previous page)

```
[ 0 -1  2 -1]
[ 0  0 -1  2]
```

class sage.algebras.lie_algebras.classical_lie_algebra.**so**(R, n)

Bases: *ClassicalMatrixLieAlgebra*

The matrix Lie algebra $\mathfrak{so}_n$.

The Lie algebra $\mathfrak{so}_n$, which is isomorphic to the Lie algebra of all anti-symmetric $n \times n$ matrices. The implementation here uses a different bilinear form and follows the description in Chapter 8 of [HK2002]. More precisely, this is the set of matrices:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

such that $A^t = -D$, $B^t = -B$, $C^t = -C$ for n even and

$$\begin{pmatrix} A & B & a \\ C & D & b \\ c & d & 0 \end{pmatrix}$$

such that $A^t = -D$, $B^t = -B$, $C^t = -C$, $a^t = -d$, and $b^t = -c$ for n odd.

This is the Lie algebra of type $B_{(n-1)/2}$ or $D_{n/2}$ if n is odd or even respectively.

killling_form(x, y)

Return the Killing form on x and y .

The Killing form on $\mathfrak{so}_n$ is:

$$\langle x | y \rangle = (n - 2)\mathrm{tr}(xy).$$

EXAMPLES:

```
sage: g = lie_algebras.so(QQ, 8, representation='matrix')
sage: x = g.an_element()
sage: y = g.lie_algebra_generators()['e1']
sage: g.killing_form(x, y)
12
sage: g = lie_algebras.so(QQ, 9, representation='matrix')
sage: x = g.an_element()
sage: y = g.lie_algebra_generators()['e1']
sage: g.killing_form(x, y)
14
```

```
>>> from sage.all import *
>>> g = lie_algebras.so(QQ, Integer(8), representation='matrix')
>>> x = g.an_element()
>>> y = g.lie_algebra_generators()['e1']
>>> g.killing_form(x, y)
12
>>> g = lie_algebras.so(QQ, Integer(9), representation='matrix')
>>> x = g.an_element()
>>> y = g.lie_algebra_generators()['e1']
>>> g.killing_form(x, y)
14
```

simple_root (*i, h*)

Return the action of the simple root $\alpha_i: \mathfrak{h} \rightarrow R$, where R is the base ring of `self`, on the element *j*.

EXAMPLES:

The even or type D case:

```
sage: g = lie_algebras.so(QQ, 8, representation='matrix')
sage: matrix([[g.simple_root(i, g.h(j)) for i in g.index_set()] for j in g.
↪index_set()]])
[ 2 -1  0  0]
[-1  2 -1 -1]
[ 0 -1  2  0]
[ 0 -1  0  2]
```

```
>>> from sage.all import *
>>> g = lie_algebras.so(QQ, Integer(8), representation='matrix')
>>> matrix([[g.simple_root(i, g.h(j)) for i in g.index_set()] for j in g.
↪index_set()]])
[ 2 -1  0  0]
[-1  2 -1 -1]
[ 0 -1  2  0]
[ 0 -1  0  2]
```

The odd or type B case:

```
sage: g = lie_algebras.so(QQ, 9, representation='matrix')
sage: matrix([[g.simple_root(i, g.h(j)) for i in g.index_set()] for j in g.
↪index_set()]])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -1]
[ 0  0 -2  2]
```

```
>>> from sage.all import *
>>> g = lie_algebras.so(QQ, Integer(9), representation='matrix')
>>> matrix([[g.simple_root(i, g.h(j)) for i in g.index_set()] for j in g.
↪index_set()]])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -1]
[ 0  0 -2  2]
```

class sage.algebras.lie_algebras.classical_lie_algebra.**sp**(*R, n*)

Bases: *ClassicalMatrixLieAlgebra*

The matrix Lie algebra $\mathfrak{sp}_n$.

The Lie algebra $\mathfrak{sp}_{2k}$, which consists of all $2k \times 2k$ matrices X that satisfy the equation:

$$X^T M - M X = 0$$

where

$$M = \begin{pmatrix} 0 & I_k \\ -I_k & 0 \end{pmatrix}.$$

This is the Lie algebra of type C_k .

killling_form(x, y)

Return the Killing form on x and y .

The Killing form on $\mathfrak{sp}_n$ is:

$$\langle x | y \rangle = (2n + 2)\mathrm{tr}(xy).$$

EXAMPLES:

```
sage: g = lie_algebras.sp(QQ, 8, representation='matrix')
sage: x = g.an_element()
sage: y = g.lie_algebra_generators()['e1']
sage: g.killing_form(x, y)
36
```

```
>>> from sage.all import *
>>> g = lie_algebras.sp(QQ, Integer(8), representation='matrix')
>>> x = g.an_element()
>>> y = g.lie_algebra_generators()['e1']
>>> g.killing_form(x, y)
36
```

simple_root(i, h)

Return the action of the simple root $\alpha_i: \mathfrak{h} \rightarrow R$, where R is the base ring of `self`, on the element j .

EXAMPLES:

```
sage: g = lie_algebras.sp(QQ, 8, representation='matrix')
sage: matrix([[g.simple_root(i, g.h(j)) for i in g.index_set()] for j in g.
↪index_set()])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -2]
[ 0  0 -1  2]
```

```
>>> from sage.all import *
>>> g = lie_algebras.sp(QQ, Integer(8), representation='matrix')
>>> matrix([[g.simple_root(i, g.h(j)) for i in g.index_set()] for j in g.
↪index_set()])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -2]
[ 0  0 -1  2]
```

9.2.8 Examples of Lie Algebras

There are the following examples of Lie algebras:

- A rather comprehensive family of 3-dimensional Lie algebras
- The Lie algebra of affine transformations of the line
- All abelian Lie algebras on free modules
- The Lie algebra of upper triangular matrices
- The Lie algebra of strictly upper triangular matrices

- The symplectic derivation Lie algebra
- The rank two Heisenberg Virasoro algebra

See also `sage.algebras.lie_algebras.virasoro.LieAlgebraRegularVectorFields` and `sage.algebras.lie_algebras.virasoro.VirasoroAlgebra` for other examples.

AUTHORS:

- Travis Scrimshaw (07-15-2013): Initial implementation

`sage.algebras.lie_algebras.examples.Heisenberg(R, n, representation='structure')`

Return the rank n Heisenberg algebra in the given representation.

INPUT:

- R – the base ring
- n – the rank (a nonnegative integer or infinity)
- `representation` – (default: 'structure') can be one of the following:
 - 'structure' – using structure coefficients
 - 'matrix' – using matrices

EXAMPLES:

```
sage: lie_algebras.Heisenberg(QQ, 3)
Heisenberg algebra of rank 3 over Rational Field
```

```
>>> from sage.all import *
>>> lie_algebras.Heisenberg(QQ, Integer(3))
Heisenberg algebra of rank 3 over Rational Field
```

`sage.algebras.lie_algebras.examples.abelian(R, names=None, index_set=None)`

Return the abelian Lie algebra generated by names.

EXAMPLES:

```
sage: lie_algebras.abelian(QQ, 'x, y, z')
Abelian Lie algebra on 3 generators (x, y, z) over Rational Field
```

```
>>> from sage.all import *
>>> lie_algebras.abelian(QQ, 'x, y, z')
Abelian Lie algebra on 3 generators (x, y, z) over Rational Field
```

`sage.algebras.lie_algebras.examples.affine_transformations_line(R, names=['X', 'Y'], representation='bracket')`

The Lie algebra of affine transformations of the line.

EXAMPLES:

```
sage: L = lie_algebras.affine_transformations_line(QQ)
sage: L.structure_coefficients()
Finite family {'X', 'Y'}: Y
sage: X, Y = L.lie_algebra_generators()
sage: L[X, Y] == Y
True
```

(continues on next page)

(continued from previous page)

```
sage: TestSuite(L).run()
sage: L = lie_algebras.affine_transformations_line(QQ, representation='matrix')
sage: X, Y = L.lie_algebra_generators()
sage: L[X, Y] == Y
True
sage: TestSuite(L).run()
```

```
>>> from sage.all import *
>>> L = lie_algebras.affine_transformations_line(QQ)
>>> L.structure_coefficients()
Finite family {('X', 'Y'): Y}
>>> X, Y = L.lie_algebra_generators()
>>> L[X, Y] == Y
True
>>> TestSuite(L).run()
>>> L = lie_algebras.affine_transformations_line(QQ, representation='matrix')
>>> X, Y = L.lie_algebra_generators()
>>> L[X, Y] == Y
True
>>> TestSuite(L).run()
```

sage.algebras.lie_algebras.examples.**cross_product**(R , names=['X', 'Y', 'Z'])

The Lie algebra of $\mathbf{R}^3$ defined by the usual cross product $\times$.

EXAMPLES:

```
sage: L = lie_algebras.cross_product(QQ)
sage: L.structure_coefficients()
Finite family {('X', 'Y'): Z, ('X', 'Z'): -Y, ('Y', 'Z'): X}
sage: TestSuite(L).run()
```

```
>>> from sage.all import *
>>> L = lie_algebras.cross_product(QQ)
>>> L.structure_coefficients()
Finite family {('X', 'Y'): Z, ('X', 'Z'): -Y, ('Y', 'Z'): X}
>>> TestSuite(L).run()
```

sage.algebras.lie_algebras.examples.**pwitt**(R, p)

Return the p -Witt Lie algebra over R .

INPUT:

- R – the base ring
- p – positive integer that is 0 in R

EXAMPLES:

```
sage: lie_algebras.pwitt(GF(5), 5)
The 5-Witt Lie algebra over Finite Field of size 5
```

```
>>> from sage.all import *
>>> lie_algebras.pwitt(GF(Integer(5)), Integer(5))
The 5-Witt Lie algebra over Finite Field of size 5
```

`sage.algebras.lie_algebras.examples.regular_vector_fields(R)`

Return the Lie algebra of regular vector fields on $\mathbf{C}^\times$.

This is also known as the Witt (Lie) algebra.

See also

[*LieAlgebraRegularVectorFields*](#)

EXAMPLES:

```
sage: lie_algebras.regular_vector_fields(QQ)
The Lie algebra of regular vector fields over Rational Field
```

```
>>> from sage.all import *
>>> lie_algebras.regular_vector_fields(QQ)
The Lie algebra of regular vector fields over Rational Field
```

`sage.algebras.lie_algebras.examples.sl(R, n, representation='bracket')`

The Lie algebra $\mathfrak{sl}_n$.

The Lie algebra $\mathfrak{sl}_n$ is the type A_{n-1} Lie algebra and is finite dimensional. As a matrix Lie algebra, it is given by the set of all $n \times n$ matrices with trace 0.

INPUT:

- `R` – the base ring
- `n` – the size of the matrix
- `representation` – (default: 'bracket') can be one of the following:
 - 'bracket' – use brackets and the Chevalley basis
 - 'matrix' – use matrices

EXAMPLES:

We first construct $\mathfrak{sl}_2$ using the Chevalley basis:

```
sage: sl2 = lie_algebras.sl(QQ, 2); sl2
Lie algebra of ['A', 1] in the Chevalley basis
sage: E, F, H = sl2.gens()
sage: E.bracket(F) == H
True
sage: H.bracket(E) == 2*E
True
sage: H.bracket(F) == -2*F
True
```

```
>>> from sage.all import *
>>> sl2 = lie_algebras.sl(QQ, Integer(2)); sl2
Lie algebra of ['A', 1] in the Chevalley basis
>>> E, F, H = sl2.gens()
>>> E.bracket(F) == H
True
>>> H.bracket(E) == Integer(2)*E
```

(continues on next page)

(continued from previous page)

```
True
>>> H.bracket(F) == -Integer(2)*F
True
```

We now construct $\mathfrak{sl}_2$ as a matrix Lie algebra:

```
sage: sl2 = lie_algebras.sl(QQ, 2, representation='matrix')
sage: E,F,H = sl2.gens()
sage: E.bracket(F) == H
True
sage: H.bracket(E) == 2*E
True
sage: H.bracket(F) == -2*F
True
```

```
>>> from sage.all import *
>>> sl2 = lie_algebras.sl(QQ, Integer(2), representation='matrix')
>>> E,F,H = sl2.gens()
>>> E.bracket(F) == H
True
>>> H.bracket(E) == Integer(2)*E
True
>>> H.bracket(F) == -Integer(2)*F
True
```

```
sage.algebras.lie_algebras.examples.so(R, n, representation='bracket')
```

The Lie algebra $\mathfrak{so}_n$.

The Lie algebra $\mathfrak{so}_n$ is the type B_k Lie algebra if $n = 2k - 1$ or the type D_k Lie algebra if $n = 2k$, and in either case is finite dimensional.

A classical description of this as a matrix Lie algebra is the set of all anti-symmetric $n \times n$ matrices. However, the implementation here uses a different bilinear form for the Lie group and follows the description in Chapter 8 of [HK2002]. See [sage.algebras.lie_algebras.classical_lie_algebra.so](#) for a precise description.

INPUT:

- R – the base ring
- n – the size of the matrix
- `representation` – (default: 'bracket') can be one of the following:
 - 'bracket' – use brackets and the Chevalley basis
 - 'matrix' – use matrices

EXAMPLES:

We first construct $\mathfrak{so}_5$ using the Chevalley basis:

```
sage: so5 = lie_algebras.so(QQ, 5); so5
Lie algebra of ['B', 2] in the Chevalley basis
sage: E1,E2, F1,F2, H1,H2 = so5.gens()
sage: so5([E1, [E1, E2]])
0
sage: X = so5([E2, [E2, E1]]); X
```

(continues on next page)

(continued from previous page)

```

-2*E[alpha[1] + 2*alpha[2]]
sage: H1.bracket(X)
0
sage: H2.bracket(X)
-4*E[alpha[1] + 2*alpha[2]]
sage: so5([H1, [E1, E2]])
-E[alpha[1] + alpha[2]]
sage: so5([H2, [E1, E2]])
0

```

```

>>> from sage.all import *
>>> so5 = lie_algebras.so(QQ, Integer(5)); so5
Lie algebra of ['B', 2] in the Chevalley basis
>>> E1,E2, F1,F2, H1,H2 = so5.gens()
>>> so5([E1, [E1, E2]])
0
>>> X = so5([E2, [E2, E1]]); X
-2*E[alpha[1] + 2*alpha[2]]
>>> H1.bracket(X)
0
>>> H2.bracket(X)
-4*E[alpha[1] + 2*alpha[2]]
>>> so5([H1, [E1, E2]])
-E[alpha[1] + alpha[2]]
>>> so5([H2, [E1, E2]])
0

```

We do the same construction of $\mathfrak{so}_4$ using the Chevalley basis:

```

sage: so4 = lie_algebras.so(QQ, 4); so4
Lie algebra of ['D', 2] in the Chevalley basis
sage: E1,E2, F1,F2, H1,H2 = so4.gens()
sage: H1.bracket(E1)
2*E[alpha[1]]
sage: H2.bracket(E1) == so4.zero()
True
sage: E1.bracket(E2) == so4.zero()
True

```

```

>>> from sage.all import *
>>> so4 = lie_algebras.so(QQ, Integer(4)); so4
Lie algebra of ['D', 2] in the Chevalley basis
>>> E1,E2, F1,F2, H1,H2 = so4.gens()
>>> H1.bracket(E1)
2*E[alpha[1]]
>>> H2.bracket(E1) == so4.zero()
True
>>> E1.bracket(E2) == so4.zero()
True

```

We now construct $\mathfrak{so}_4$ as a matrix Lie algebra:

```

sage: sl2 = lie_algebras.sl(QQ, 2, representation='matrix')
sage: E1,E2, F1,F2, H1,H2 = so4.gens()
sage: H2.bracket(E1) == so4.zero()
True
sage: E1.bracket(E2) == so4.zero()
True

```

```

>>> from sage.all import *
>>> sl2 = lie_algebras.sl(QQ, Integer(2), representation='matrix')
>>> E1,E2, F1,F2, H1,H2 = so4.gens()
>>> H2.bracket(E1) == so4.zero()
True
>>> E1.bracket(E2) == so4.zero()
True

```

`sage.algebras.lie_algebras.examples.sp(R, n, representation='bracket')`

The Lie algebra $\mathfrak{sp}_n$.

The Lie algebra $\mathfrak{sp}_n$ where $n = 2k$ is the type C_k Lie algebra and is finite dimensional. As a matrix Lie algebra, it is given by the set of all matrices X that satisfy the equation:

$$X^T M - M X = 0$$

where

$$M = \begin{pmatrix} 0 & I_k \\ -I_k & 0 \end{pmatrix}.$$

This is the Lie algebra of type C_k .

INPUT:

- R – the base ring
- n – the size of the matrix
- `representation` – (default: 'bracket') can be one of the following:
 - 'bracket' – use brackets and the Chevalley basis
 - 'matrix' – use matrices

EXAMPLES:

We first construct $\mathfrak{sp}_4$ using the Chevalley basis:

```

sage: sp4 = lie_algebras.sp(QQ, 4); sp4
Lie algebra of ['C', 2] in the Chevalley basis
sage: E1,E2, F1,F2, H1,H2 = sp4.gens()
sage: sp4([E2, [E2, E1]])
0
sage: X = sp4([E1, [E1, E2]]); X
2*E[2*alpha[1] + alpha[2]]
sage: H1.bracket(X)
4*E[2*alpha[1] + alpha[2]]
sage: H2.bracket(X)
0
sage: sp4([H1, [E1, E2]])

```

(continues on next page)

(continued from previous page)

```
0
sage: sp4([H2, [E1, E2]])
-E[alpha[1] + alpha[2]]
```

```
>>> from sage.all import *
>>> sp4 = lie_algebras.sp(QQ, Integer(4)); sp4
Lie algebra of ['C', 2] in the Chevalley basis
>>> E1,E2, F1,F2, H1,H2 = sp4.gens()
>>> sp4([E2, [E2, E1]])
0
>>> X = sp4([E1, [E1, E2]]); X
2*E[2*alpha[1] + alpha[2]]
>>> H1.bracket(X)
4*E[2*alpha[1] + alpha[2]]
>>> H2.bracket(X)
0
>>> sp4([H1, [E1, E2]])
0
>>> sp4([H2, [E1, E2]])
-E[alpha[1] + alpha[2]]
```

We now construct $\mathfrak{sp}_4$ as a matrix Lie algebra:

```
sage: sp4 = lie_algebras.sp(QQ, 4, representation='matrix'); sp4
Symplectic Lie algebra of rank 4 over Rational Field
sage: E1,E2, F1,F2, H1,H2 = sp4.gens()
sage: H1.bracket(E1)
[ 0  2  0  0]
[ 0  0  0  0]
[ 0  0  0  0]
[ 0  0 -2  0]
sage: sp4([E1, [E1, E2]])
[0 0 2 0]
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
```

```
>>> from sage.all import *
>>> sp4 = lie_algebras.sp(QQ, Integer(4), representation='matrix'); sp4
Symplectic Lie algebra of rank 4 over Rational Field
>>> E1,E2, F1,F2, H1,H2 = sp4.gens()
>>> H1.bracket(E1)
[ 0  2  0  0]
[ 0  0  0  0]
[ 0  0  0  0]
[ 0  0 -2  0]
>>> sp4([E1, [E1, E2]])
[0 0 2 0]
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
```

`sage.algebras.lie_algebras.examples.strictly_upper_triangular_matrices( $R, n$ )`

Return the Lie algebra $\mathfrak{n}_k$ of strictly $k \times k$ upper triangular matrices.

Todo

This implementation does not know it is finite-dimensional and does not know its basis.

EXAMPLES:

```
sage: L = lie_algebras.strictly_upper_triangular_matrices(QQ, 4); L
Lie algebra of 4-dimensional strictly upper triangular matrices over Rational_
↪Field
sage: TestSuite(L).run()
sage: n0, n1, n2 = L.lie_algebra_generators()
sage: L[n2, n1]
[ 0  0  0  0]
[ 0  0  0 -1]
[ 0  0  0  0]
[ 0  0  0  0]
```

```
>>> from sage.all import *
>>> L = lie_algebras.strictly_upper_triangular_matrices(QQ, Integer(4)); L
Lie algebra of 4-dimensional strictly upper triangular matrices over Rational_
↪Field
>>> TestSuite(L).run()
>>> n0, n1, n2 = L.lie_algebra_generators()
>>> L[n2, n1]
[ 0  0  0  0]
[ 0  0  0 -1]
[ 0  0  0  0]
[ 0  0  0  0]
```

```
sage.algebras.lie_algebras.examples.su(R, n, representation='matrix')
```

The Lie algebra $\mathfrak{su}_n$.

The Lie algebra $\mathfrak{su}_n$ is the compact real form of the type A_{n-1} Lie algebra and is finite-dimensional. As a matrix Lie algebra, it is given by the set of all $n \times n$ skew-Hermitian matrices with trace 0.

INPUT:

- R – the base ring
- n – the size of the matrix
- `representation` – (default: 'matrix') can be one of the following:
 - 'bracket' – use brackets and the Chevalley basis
 - 'matrix' – use matrices

EXAMPLES:

We construct $\mathfrak{su}_2$, where the default is as a matrix Lie algebra:

```
sage: su2 = lie_algebras.su(QQ, 2)
sage: E, H, F = su2.basis()
sage: E.bracket(F) == 2*H
```

(continues on next page)

(continued from previous page)

```
True
sage: H.bracket(E) == 2*F
True
sage: H.bracket(F) == -2*E
True
```

```
>>> from sage.all import *
>>> su2 = lie_algebras.su(QQ, Integer(2))
>>> E,H,F = su2.basis()
>>> E.bracket(F) == Integer(2)*H
True
>>> H.bracket(E) == Integer(2)*F
True
>>> H.bracket(F) == -Integer(2)*E
True
```

Since $\mathfrak{su}_n$ is the same as the type A_{n-1} Lie algebra, the bracket is the same as `sl()`:

```
sage: su2 = lie_algebras.su(QQ, 2, representation='bracket')
sage: su2 is lie_algebras.sl(QQ, 2, representation='bracket')
True
```

```
>>> from sage.all import *
>>> su2 = lie_algebras.su(QQ, Integer(2), representation='bracket')
>>> su2 is lie_algebras.sl(QQ, Integer(2), representation='bracket')
True
```

`sage.algebras.lie_algebras.examples.three_dimensional(R, a, b, c, d, names=['X', 'Y', 'Z'])`

The 3-dimensional Lie algebra over a given commutative ring R with basis $\{X, Y, Z\}$ subject to the relations:

$$[X, Y] = aZ + dY, \quad [Y, Z] = bX, \quad [Z, X] = cY + dZ$$

where $a, b, c, d \in R$.

This is always a well-defined 3-dimensional Lie algebra, as can be easily proven by computation.

EXAMPLES:

```
sage: L = lie_algebras.three_dimensional(QQ, 4, 1, -1, 2)
sage: L.structure_coefficients()
Finite family {('X', 'Y'): 2*Y + 4*Z, ('X', 'Z'): Y - 2*Z, ('Y', 'Z'): X}
sage: TestSuite(L).run()
sage: L = lie_algebras.three_dimensional(QQ, 1, 0, 0, 0)
sage: L.structure_coefficients()
Finite family {('X', 'Y'): Z}
sage: L = lie_algebras.three_dimensional(QQ, 0, 0, -1, -1)
sage: L.structure_coefficients()
Finite family {('X', 'Y'): -Y, ('X', 'Z'): Y + Z}
sage: L = lie_algebras.three_dimensional(QQ, 0, 1, 0, 0)
sage: L.structure_coefficients()
Finite family {('Y', 'Z'): X}
sage: lie_algebras.three_dimensional(QQ, 0, 0, 0, 0)
Abelian Lie algebra on 3 generators (X, Y, Z) over Rational Field
```

(continues on next page)

(continued from previous page)

```

sage: Q.<a,b,c,d> = PolynomialRing(QQ)
sage: L = lie_algebras.three_dimensional(Q, a, b, c, d)
sage: L.structure_coefficients()
Finite family {('X', 'Y'): d*Y + a*Z, ('X', 'Z'): -c*Y - d*Z, ('Y', 'Z'): b*X}
sage: TestSuite(L).run()

```

```

>>> from sage.all import *
>>> L = lie_algebras.three_dimensional(QQ, Integer(4), Integer(1), -Integer(1),
↳ Integer(2))
>>> L.structure_coefficients()
Finite family {('X', 'Y'): 2*Y + 4*Z, ('X', 'Z'): Y - 2*Z, ('Y', 'Z'): X}
>>> TestSuite(L).run()
>>> L = lie_algebras.three_dimensional(QQ, Integer(1), Integer(0), Integer(0),
↳ Integer(0))
>>> L.structure_coefficients()
Finite family {('X', 'Y'): Z}
>>> L = lie_algebras.three_dimensional(QQ, Integer(0), Integer(0), -Integer(1), -
↳ Integer(1))
>>> L.structure_coefficients()
Finite family {('X', 'Y'): -Y, ('X', 'Z'): Y + Z}
>>> L = lie_algebras.three_dimensional(QQ, Integer(0), Integer(1), Integer(0),
↳ Integer(0))
>>> L.structure_coefficients()
Finite family {('Y', 'Z'): X}
>>> lie_algebras.three_dimensional(QQ, Integer(0), Integer(0), Integer(0),
↳ Integer(0))
Abelian Lie algebra on 3 generators (X, Y, Z) over Rational Field
>>> Q = PolynomialRing(QQ, names=('a', 'b', 'c', 'd',)); (a, b, c, d,) = Q._first_
↳ ngens(4)
>>> L = lie_algebras.three_dimensional(Q, a, b, c, d)
>>> L.structure_coefficients()
Finite family {('X', 'Y'): d*Y + a*Z, ('X', 'Z'): -c*Y - d*Z, ('Y', 'Z'): b*X}
>>> TestSuite(L).run()

```

```

sage.algebras.lie_algebras.examples.three_dimensional_by_rank(R, n, a=None, names=['X', 'Y',
                                                                                       'Z'])

```

Return a 3-dimensional Lie algebra of rank n , where $0 \leq n \leq 3$.

Here, the *rank* of a Lie algebra L is defined as the dimension of its derived subalgebra $[L, L]$. (We are assuming that R is a field of characteristic 0; otherwise the Lie algebras constructed by this function are still well-defined but no longer might have the correct ranks.) This is not to be confused with the other standard definition of a rank (namely, as the dimension of a Cartan subalgebra, when L is semisimple).

INPUT:

- R – the base ring
- n – the rank
- a – the deformation parameter (used for $n = 2$); this should be a nonzero element of R in order for the resulting Lie algebra to actually have the right rank(?)
- $names$ – (optional) the generator names

EXAMPLES:

```

sage: lie_algebras.three_dimensional_by_rank(QQ, 0)
Abelian Lie algebra on 3 generators (X, Y, Z) over Rational Field
sage: L = lie_algebras.three_dimensional_by_rank(QQ, 1)
sage: L.structure_coefficients()
Finite family {'Y', 'Z'): X}
sage: L = lie_algebras.three_dimensional_by_rank(QQ, 2, 4)
sage: L.structure_coefficients()
Finite family {'X', 'Y'): Y, ('X', 'Z'): Y + Z}
sage: L = lie_algebras.three_dimensional_by_rank(QQ, 2, 0)
sage: L.structure_coefficients()
Finite family {'X', 'Y'): Y}
sage: lie_algebras.three_dimensional_by_rank(QQ, 3)
sl2 over Rational Field

```

```

>>> from sage.all import *
>>> lie_algebras.three_dimensional_by_rank(QQ, Integer(0))
Abelian Lie algebra on 3 generators (X, Y, Z) over Rational Field
>>> L = lie_algebras.three_dimensional_by_rank(QQ, Integer(1))
>>> L.structure_coefficients()
Finite family {'Y', 'Z'): X}
>>> L = lie_algebras.three_dimensional_by_rank(QQ, Integer(2), Integer(4))
>>> L.structure_coefficients()
Finite family {'X', 'Y'): Y, ('X', 'Z'): Y + Z}
>>> L = lie_algebras.three_dimensional_by_rank(QQ, Integer(2), Integer(0))
>>> L.structure_coefficients()
Finite family {'X', 'Y'): Y}
>>> lie_algebras.three_dimensional_by_rank(QQ, Integer(3))
sl2 over Rational Field

```

`sage.algebras.lie_algebras.examples.upper_triangular_matrices( $R, n$ )`

Return the Lie algebra $\mathfrak{b}_k$ of $k \times k$ upper triangular matrices.

Todo

This implementation does not know it is finite-dimensional and does not know its basis.

EXAMPLES:

```

sage: L = lie_algebras.upper_triangular_matrices(QQ, 4); L
Lie algebra of 4-dimensional upper triangular matrices over Rational Field
sage: TestSuite(L).run()
sage: n0, n1, n2, t0, t1, t2, t3 = L.lie_algebra_generators()
sage: L[n2, t2] == -n2
True

```

```

>>> from sage.all import *
>>> L = lie_algebras.upper_triangular_matrices(QQ, Integer(4)); L
Lie algebra of 4-dimensional upper triangular matrices over Rational Field
>>> TestSuite(L).run()
>>> n0, n1, n2, t0, t1, t2, t3 = L.lie_algebra_generators()
>>> L[n2, t2] == -n2

```

(continues on next page)

(continued from previous page)

True

sage.algebras.lie_algebras.examples.witt(R)

Return the Lie algebra of regular vector fields on $\mathbf{C}^\times$.

This is also known as the Witt (Lie) algebra.

See also

[*LieAlgebraRegularVectorFields*](#)

EXAMPLES:

```
sage: lie_algebras.regular_vector_fields(QQ)
The Lie algebra of regular vector fields over Rational Field
```

```
>>> from sage.all import *
>>> lie_algebras.regular_vector_fields(QQ)
The Lie algebra of regular vector fields over Rational Field
```

9.2.9 Free Lie Algebras

AUTHORS:

- Travis Scrimshaw (2013-05-03): Initial version

REFERENCES:

- [Bou1989]
- [Reu2003]

class sage.algebras.lie_algebras.free_lie_algebra.**FreeLieAlgebra** (R , $names$, $index\_set$)

Bases: [Parent](#), [UniqueRepresentation](#)

The free Lie algebra of a set X .

The free Lie algebra g_X of a set X is the Lie algebra with generators $\{g_x\}_{x \in X}$ where there are no other relations beyond the defining relations. This can be constructed as the free magmatic algebra M_X quotiented by the ideal generated by $(xx, xy + yx, x(yz) + y(zx) + z(xy))$.

EXAMPLES:

We first construct the free Lie algebra in the Hall basis:

```
sage: L = LieAlgebra(QQ, 'x,y,z')
sage: H = L.Hall()
sage: x,y,z = H.gens()
sage: h_elt = H([x, [y, z]]) + H([x - H([y, x]), H([x, z]])]; h_elt
[x, [x, z]] + [y, [x, z]] - [z, [x, y]] + [[x, y], [x, z]]
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, 'x,y,z')
>>> H = L.Hall()
>>> x,y,z = H.gens()
```

(continues on next page)

(continued from previous page)

```
>>> h_elt = H([x, [y, z]]) + H([x - H([y, x]), H([x, z])]); h_elt
[x, [x, z]] + [y, [x, z]] - [z, [x, y]] + [[x, y], [x, z]]
```

We can also use the Lyndon basis and go between the two:

```
sage: Lyn = L.Lyndon()
sage: l_elt = Lyn([x, [y, z]]) + Lyn([x - Lyn([y, x]), Lyn([x, z])]); l_elt
[x, [x, z]] + [[x, y], [x, z]] + [x, [y, z]]
sage: Lyn(h_elt) == l_elt
True
sage: H(l_elt) == h_elt
True
```

```
>>> from sage.all import *
>>> Lyn = L.Lyndon()
>>> l_elt = Lyn([x, [y, z]]) + Lyn([x - Lyn([y, x]), Lyn([x, z])]); l_elt
[x, [x, z]] + [[x, y], [x, z]] + [x, [y, z]]
>>> Lyn(h_elt) == l_elt
True
>>> H(l_elt) == h_elt
True
```

class `Hall` (*lie*)

Bases: `FreeLieBasis_abstract`

The free Lie algebra in the Hall basis.

The basis keys are objects of class `LieObject`, each of which is either a `LieGenerator` (in degree 1) or a `GradedLieBracket` (in degree > 1).

graded_basis (*k*)

Return the basis for the *k*-th graded piece of `self`.

EXAMPLES:

```
sage: L = LieAlgebra(QQ, 'x,y,z')
sage: H = L.Hall()
sage: H.graded_basis(2)
([x, y], [x, z], [y, z])
sage: H.graded_basis(4)
([x, [x, [x, y]]], [x, [x, [x, z]]],
 [y, [x, [x, y]]], [y, [x, [x, z]]],
 [y, [y, [x, y]]], [y, [y, [x, z]]],
 [y, [y, [y, z]]], [z, [x, [x, y]]],
 [z, [x, [x, z]]], [z, [y, [x, y]]],
 [z, [y, [x, z]]], [z, [y, [y, z]]],
 [z, [z, [x, y]]], [z, [z, [x, z]]],
 [z, [z, [y, z]]], [[x, y], [x, z]],
 [[x, y], [y, z]], [[x, z], [y, z]])
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, 'x,y,z')
>>> H = L.Hall()
```

(continues on next page)

(continued from previous page)

```

>>> H.graded_basis(Integer(2))
([x, y], [x, z], [y, z])
>>> H.graded_basis(Integer(4))
([x, [x, [x, y]], [x, [x, [x, z]]],
 [y, [x, [x, y]], [y, [x, [x, z]]],
 [y, [y, [x, y]], [y, [y, [x, z]]],
 [y, [y, [y, z]], [z, [x, [x, y]],
 [z, [x, [x, z]], [z, [y, [x, y]],
 [z, [y, [x, z]], [z, [y, [y, z]],
 [z, [z, [x, y]], [z, [z, [x, z]],
 [z, [z, [y, z]], [[x, y], [x, z]],
 [[x, y], [y, z]], [[x, z], [y, z]])

```

class `Lyndon` (*lie*)

Bases: `FreeLieBasis_abstract`

The free Lie algebra in the Lyndon basis.

The basis keys are objects of class `LieObject`, each of which is either a `LieGenerator` (in degree 1) or a `LyndonBracket` (in degree > 1).

graded_basis (*k*)

Return the basis for the *k*-th graded piece of `self`.

EXAMPLES:

```

sage: L = LieAlgebra(QQ, 'x', 3)
sage: Lyn = L.Lyndon()
sage: Lyn.graded_basis(1)
(x0, x1, x2)
sage: Lyn.graded_basis(2)
([x0, x1], [x0, x2], [x1, x2])
sage: Lyn.graded_basis(4)
([x0, [x0, [x0, x1]],
 [x0, [x0, [x0, x2]],
 [x0, [[x0, x1], x1]],
 [x0, [x0, [x1, x2]],
 [x0, [[x0, x2], x1]],
 [x0, [[x0, x2], x2]],
 [[x0, x1], [x0, x2]],
 [[[x0, x1], x1], x1],
 [x0, [x1, [x1, x2]],
 [[x0, [x1, x2]], x1],
 [x0, [[x1, x2], x2]],
 [[[x0, x2], x1], x1],
 [[x0, x2], [x1, x2]],
 [[[x0, x2], x2], x1],
 [[[x0, x2], x2], x2],
 [x1, [x1, [x1, x2]],
 [x1, [[x1, x2], x2]],
 [[[x1, x2], x2], x2])

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, 'x', Integer(3))

```

(continues on next page)

(continued from previous page)

```

>>> Lyn = L.Lyndon()
>>> Lyn.graded_basis(Integer(1))
(x0, x1, x2)
>>> Lyn.graded_basis(Integer(2))
([x0, x1], [x0, x2], [x1, x2])
>>> Lyn.graded_basis(Integer(4))
([x0, [x0, [x0, x1]]],
 [x0, [x0, [x0, x2]]],
 [x0, [[x0, x1], x1]],
 [x0, [x0, [x1, x2]]],
 [x0, [[x0, x2], x1]],
 [x0, [[x0, x2], x2]],
 [[x0, x1], [x0, x2]],
 [[x0, x1], x1], x1],
 [x0, [x1, [x1, x2]]],
 [[x0, [x1, x2]], x1],
 [x0, [[x1, x2], x2]],
 [[x0, x2], x1], x1],
 [[x0, x2], [x1, x2]],
 [[x0, x2], x2], x1],
 [[x0, x2], x2], x2],
 [x1, [x1, [x1, x2]]],
 [x1, [[x1, x2], x2]],
 [[x1, x2], x2], x2])

```

pbw_basis(***kws*)

Return the Poincare-Birkhoff-Witt basis corresponding to self.

EXAMPLES:

```

sage: L = LieAlgebra(QQ, 'x,y,z', 3)
sage: Lyn = L.Lyndon()
sage: Lyn.pbw_basis()
The Poincare-Birkhoff-Witt basis of Free Algebra on 3 generators (x, y, ↵
↵z) over Rational Field

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, 'x,y,z', Integer(3))
>>> Lyn = L.Lyndon()
>>> Lyn.pbw_basis()
The Poincare-Birkhoff-Witt basis of Free Algebra on 3 generators (x, y, ↵
↵z) over Rational Field

```

poincare_birkhoff_witt_basis(***kws*)

Return the Poincare-Birkhoff-Witt basis corresponding to self.

EXAMPLES:

```

sage: L = LieAlgebra(QQ, 'x,y,z', 3)
sage: Lyn = L.Lyndon()
sage: Lyn.pbw_basis()
The Poincare-Birkhoff-Witt basis of Free Algebra on 3 generators (x, y, ↵
↵z) over Rational Field

```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, 'x,y,z', Integer(3))
>>> Lyn = L.Lyndon()
>>> Lyn.pbw_basis()
The Poincare-Birkhoff-Witt basis of Free Algebra on 3 generators (x, y, z) over Rational Field
```

a_realization()

Return a particular realization of `self` (the Lyndon basis).

EXAMPLES:

```
sage: L.<x, y> = LieAlgebra(QQ)
sage: L.a_realization()
Free Lie algebra generated by (x, y) over Rational Field in the Lyndon basis
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, names=('x', 'y',)); (x, y,) = L._first_ngens(2)
>>> L.a_realization()
Free Lie algebra generated by (x, y) over Rational Field in the Lyndon basis
```

gen(i)

Return the i -th generator of `self` in the Lyndon basis.

EXAMPLES:

```
sage: L.<x, y> = LieAlgebra(QQ)
sage: L.gen(0)
x
sage: L.gen(1)
y
sage: L.gen(0).parent()
Free Lie algebra generated by (x, y) over Rational Field in the Lyndon basis
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, names=('x', 'y',)); (x, y,) = L._first_ngens(2)
>>> L.gen(Integer(0))
x
>>> L.gen(Integer(1))
y
>>> L.gen(Integer(0)).parent()
Free Lie algebra generated by (x, y) over Rational Field in the Lyndon basis
```

gens()

Return the generators of `self` in the Lyndon basis.

EXAMPLES:

```
sage: L.<x, y> = LieAlgebra(QQ)
sage: L.gens()
(x, y)
sage: L.gens()[0].parent()
Free Lie algebra generated by (x, y) over Rational Field in the Lyndon basis
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, names=('x', 'y')); (x, y) = L._first_ngens(2)
>>> L.gens()
(x, y)
>>> L.gens()[Integer(0)].parent()
Free Lie algebra generated by (x, y) over Rational Field in the Lyndon basis
```

lie_algebra_generators()

Return the Lie algebra generators of self in the Lyndon basis.

EXAMPLES:

```
sage: L.<x, y> = LieAlgebra(QQ)
sage: L.lie_algebra_generators()
Finite family {'x': x, 'y': y}
sage: L.lie_algebra_generators()['x'].parent()
Free Lie algebra generated by (x, y) over Rational Field in the Lyndon basis
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, names=('x', 'y')); (x, y) = L._first_ngens(2)
>>> L.lie_algebra_generators()
Finite family {'x': x, 'y': y}
>>> L.lie_algebra_generators()['x'].parent()
Free Lie algebra generated by (x, y) over Rational Field in the Lyndon basis
```

class sage.algebras.lie_algebras.free_lie_algebra.**FreeLieAlgebraBases**(base)

Bases: `Category_realization_of_parent`

The category of bases of a free Lie algebra.

super_categories()

The super categories of self.

EXAMPLES:

```
sage: from sage.algebras.lie_algebras.free_lie_algebra import _
↪FreeLieAlgebraBases
sage: L.<x, y> = LieAlgebra(QQ)
sage: bases = FreeLieAlgebraBases(L)
sage: bases.super_categories()
[Category of Lie algebras with basis over Rational Field,
Category of realizations of Free Lie algebra generated by (x, y) over_
↪Rational Field]
```

```
>>> from sage.all import *
>>> from sage.algebras.lie_algebras.free_lie_algebra import _
↪FreeLieAlgebraBases
>>> L = LieAlgebra(QQ, names=('x', 'y')); (x, y) = L._first_ngens(2)
>>> bases = FreeLieAlgebraBases(L)
>>> bases.super_categories()
[Category of Lie algebras with basis over Rational Field,
Category of realizations of Free Lie algebra generated by (x, y) over_
↪Rational Field]
```

class sage.algebras.lie_algebras.free_lie_algebra.**FreeLieBasis_abstract** (*lie, basis_name*)

Bases: *FinitelyGeneratedLieAlgebra*, *IndexedGenerators*, *BindableClass*

Abstract base class for all bases of a free Lie algebra.

Element

alias of *FreeLieAlgebraElement*

basis()

Return the basis of self.

EXAMPLES:

```
sage: L = LieAlgebra(QQ, 3, 'x')
sage: L.Hall().basis()
Disjoint union of Lazy family (graded basis(i))_{i in Positive integers}
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(3), 'x')
>>> L.Hall().basis()
Disjoint union of Lazy family (graded basis(i))_{i in Positive integers}
```

graded_basis(k)

Return the basis for the k-th graded piece of self.

EXAMPLES:

```
sage: H = LieAlgebra(QQ, 3, 'x').Hall()
sage: H.graded_basis(2)
([x0, x1], [x0, x2], [x1, x2])
```

```
>>> from sage.all import *
>>> H = LieAlgebra(QQ, Integer(3), 'x').Hall()
>>> H.graded_basis(Integer(2))
([x0, x1], [x0, x2], [x1, x2])
```

graded_dimension(k)

Return the dimension of the k-th graded piece of self.

The k-th graded part of a free Lie algebra on n generators has dimension

$$\frac{1}{k} \sum_{d|k} \mu(d) n^{k/d},$$

where μ is the Mobius function.

REFERENCES:

[MKO1998]

EXAMPLES:

```
sage: L = LieAlgebra(QQ, 'x', 3)
sage: H = L.Hall()
sage: [H.graded_dimension(i) for i in range(1, 11)]
[3, 3, 8, 18, 48, 116, 312, 810, 2184, 5880]
sage: H.graded_dimension(0)
0
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, 'x', Integer(3))
>>> H = L.Hall()
>>> [H.graded_dimension(i) for i in range(Integer(1), Integer(11))]
[3, 3, 8, 18, 48, 116, 312, 810, 2184, 5880]
>>> H.graded_dimension(Integer(0))
0
```

is_abelian()

Return True if this is an abelian Lie algebra.

EXAMPLES:

```
sage: L = LieAlgebra(QQ, 3, 'x')
sage: L.is_abelian()
False
sage: L = LieAlgebra(QQ, 1, 'x')
sage: L.is_abelian()
True
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(3), 'x')
>>> L.is_abelian()
False
>>> L = LieAlgebra(QQ, Integer(1), 'x')
>>> L.is_abelian()
True
```

monomial(x)

Return the monomial indexed by x.

EXAMPLES:

```
sage: Lyn = LieAlgebra(QQ, 'x,y').Lyndon()
sage: x = Lyn.monomial('x'); x
x
sage: x.parent() is Lyn
True
```

```
>>> from sage.all import *
>>> Lyn = LieAlgebra(QQ, 'x,y').Lyndon()
>>> x = Lyn.monomial('x'); x
x
>>> x.parent() is Lyn
True
```

`sage.algebras.lie_algebras.free_lie_algebra.is_lyndon(w)`

Modified form of `Word(w).is_lyndon()` which uses the default order (this will either be the natural integer order or lex order) and assumes the input `w` behaves like a nonempty list.

This function here is designed for speed.

EXAMPLES:


```

sage: from sage.algebras.lie_algebras.free_lie_algebra import is_lyndon
sage: is_lyndon([1])
True
sage: is_lyndon([1,3,1])
False
sage: is_lyndon((2,2,3))
True
sage: all(is_lyndon(x) for x in LyndonWords(3, 5))
True
sage: all(is_lyndon(x) for x in LyndonWords(6, 4))
True

```

```

>>> from sage.all import *
>>> from sage.algebras.lie_algebras.free_lie_algebra import is_lyndon
>>> is_lyndon([Integer(1)])
True
>>> is_lyndon([Integer(1), Integer(3), Integer(1)])
False
>>> is_lyndon((Integer(2), Integer(2), Integer(3)))
True
>>> all(is_lyndon(x) for x in LyndonWords(Integer(3), Integer(5)))
True
>>> all(is_lyndon(x) for x in LyndonWords(Integer(6), Integer(4)))
True

```

9.2.10 Heisenberg Algebras

AUTHORS:

- Travis Scrimshaw (2013-08-13): Initial version

class sage.algebras.lie_algebras.heisenberg.**HeisenbergAlgebra**(R, n)

Bases: *HeisenbergAlgebra_fd*, *HeisenbergAlgebra_abstract*, *LieAlgebraWithGenerators*

A Heisenberg algebra defined using structure coefficients.

The n -th Heisenberg algebra (where n is a nonnegative integer or infinity) is the Lie algebra with basis $\{p_i\}_{1 \leq i \leq n} \cup \{q_i\}_{1 \leq i \leq n} \cup \{z\}$ with the following relations:

$$[p_i, q_j] = \delta_{ij}z, \quad [p_i, z] = [q_i, z] = [p_i, p_j] = [q_i, q_j] = 0.$$

This Lie algebra is also known as the Heisenberg algebra of rank n .

Note

The relations $[p_i, q_j] = \delta_{ij}z$, $[p_i, z] = 0$, and $[q_i, z] = 0$ are known as canonical commutation relations. See Wikipedia article [Canonical commutation relations](#).

Warning

The n in the above definition is called the “rank” of the Heisenberg algebra; it is not, however, a rank in any of the usual meanings that this word has in the theory of Lie algebras.

INPUT:

- R – the base ring
- n – the rank of the Heisenberg algebra

REFERENCES:

- [Wikipedia article Heisenberg algebra](#)

EXAMPLES:

```
sage: L = lie_algebras.Heisenberg(QQ, 2)
```

```
>>> from sage.all import *
>>> L = lie_algebras.Heisenberg(QQ, Integer(2))
```

```
class sage.algebras.lie_algebras.heisenberg.HeisenbergAlgebra_abstract(I)
```

Bases: *IndexedGenerators*

The common methods for the (non-matrix) Heisenberg algebras.

```
class Element
```

Bases: *LieAlgebraElement*

```
bracket_on_basis(x, y)
```

Return the bracket of basis elements indexed by x and y where $x < y$.

The basis of a Heisenberg algebra is ordered in such a way that the p_i come first, the q_i come next, and the z comes last.

EXAMPLES:

```
sage: H = lie_algebras.Heisenberg(QQ, 3)
sage: p1 = ('p', 1)
sage: q1 = ('q', 1)
sage: H.bracket_on_basis(p1, q1)
z
```

```
>>> from sage.all import *
>>> H = lie_algebras.Heisenberg(QQ, Integer(3))
>>> p1 = ('p', Integer(1))
>>> q1 = ('q', Integer(1))
>>> H.bracket_on_basis(p1, q1)
z
```

$p(i)$

The generator p_i of the Heisenberg algebra.

EXAMPLES:

```
sage: L = lie_algebras.Heisenberg(QQ, oo)
sage: L.p(2)
p2
```

```
>>> from sage.all import *
>>> L = lie_algebras.Heisenberg(QQ, oo)
```

(continues on next page)

(continued from previous page)

```
>>> L.p(Integer(2))
p2
```

q(i)

The generator q_i of the Heisenberg algebra.

EXAMPLES:

```
sage: L = lie_algebras.Heisenberg(QQ, oo)
sage: L.q(2)
q2
```

```
>>> from sage.all import *
>>> L = lie_algebras.Heisenberg(QQ, oo)
>>> L.q(Integer(2))
q2
```

step()

Return the nilpotency step of self.

EXAMPLES:

```
sage: h = lie_algebras.Heisenberg(ZZ, 10)
sage: h.step()
2

sage: h = lie_algebras.Heisenberg(ZZ, oo)
sage: h.step()
2
```

```
>>> from sage.all import *
>>> h = lie_algebras.Heisenberg(ZZ, Integer(10))
>>> h.step()
2

>>> h = lie_algebras.Heisenberg(ZZ, oo)
>>> h.step()
2
```

z()

Return the basis element z of the Heisenberg algebra.

The element z spans the center of the Heisenberg algebra.

EXAMPLES:

```
sage: L = lie_algebras.Heisenberg(QQ, oo)
sage: L.z()
z
```

```
>>> from sage.all import *
>>> L = lie_algebras.Heisenberg(QQ, oo)
>>> L.z()
z
```

```
class sage.algebras.lie_algebras.heisenberg.HeisenbergAlgebra_fd(n)
```

Bases: object

Common methods for finite-dimensional Heisenberg algebras.

basis()

Return the basis of `self`.

EXAMPLES:

```
sage: H = lie_algebras.Heisenberg(QQ, 1)
sage: H.basis()
Finite family {'p1': p1, 'q1': q1, 'z': z}
```

```
>>> from sage.all import *
>>> H = lie_algebras.Heisenberg(QQ, Integer(1))
>>> H.basis()
Finite family {'p1': p1, 'q1': q1, 'z': z}
```

gen(i)

Return the `i`-th generator of `self`.

EXAMPLES:

```
sage: H = lie_algebras.Heisenberg(QQ, 2)
sage: H.gen(0)
p1
sage: H.gen(3)
q2
```

```
>>> from sage.all import *
>>> H = lie_algebras.Heisenberg(QQ, Integer(2))
>>> H.gen(Integer(0))
p1
>>> H.gen(Integer(3))
q2
```

gens()

Return the Lie algebra generators of `self`.

EXAMPLES:

```
sage: H = lie_algebras.Heisenberg(QQ, 2)
sage: H.gens()
(p1, p2, q1, q2)
sage: H = lie_algebras.Heisenberg(QQ, 0)
sage: H.gens()
(z,)
```

```
>>> from sage.all import *
>>> H = lie_algebras.Heisenberg(QQ, Integer(2))
>>> H.gens()
(p1, p2, q1, q2)
>>> H = lie_algebras.Heisenberg(QQ, Integer(0))
```

(continues on next page)

(continued from previous page)

```
>>> H.gens()
(z,)
```

lie_algebra_generators()

Return the Lie algebra generators of *self*.

EXAMPLES:

```
sage: H = lie_algebras.Heisenberg(QQ, 1)
sage: H.lie_algebra_generators()
Finite family {'p1': p1, 'q1': q1}
sage: H = lie_algebras.Heisenberg(QQ, 0)
sage: H.lie_algebra_generators()
Finite family {'z': z}
```

```
>>> from sage.all import *
>>> H = lie_algebras.Heisenberg(QQ, Integer(1))
>>> H.lie_algebra_generators()
Finite family {'p1': p1, 'q1': q1}
>>> H = lie_algebras.Heisenberg(QQ, Integer(0))
>>> H.lie_algebra_generators()
Finite family {'z': z}
```

n()

Return the rank of the Heisenberg algebra *self*.

This is the *n* such that *self* is the *n*-th Heisenberg algebra. The dimension of this Heisenberg algebra is then $2n + 1$.

EXAMPLES:

```
sage: H = lie_algebras.Heisenberg(QQ, 3)
sage: H.n()
3
sage: H = lie_algebras.Heisenberg(QQ, 3, representation='matrix')
sage: H.n()
3
```

```
>>> from sage.all import *
>>> H = lie_algebras.Heisenberg(QQ, Integer(3))
>>> H.n()
3
>>> H = lie_algebras.Heisenberg(QQ, Integer(3), representation='matrix')
>>> H.n()
3
```

class sage.algebras.lie_algebras.heisenberg.**HeisenbergAlgebra_matrix**(*R*, *n*)

Bases: *HeisenbergAlgebra_fd*, *LieAlgebraFromAssociative*

A Heisenberg algebra represented using matrices.

The *n*-th Heisenberg algebra over *R* is a Lie algebra which is defined as the Lie algebra of the $(n + 2) \times (n +$

2)-matrices:

$$\begin{bmatrix} 0 & p^T & k \\ 0 & 0_n & q \\ 0 & 0 & 0 \end{bmatrix}$$

where $p, q \in R^n$ and 0_n in the $n \times n$ zero matrix. It has a basis consisting of

$$\begin{aligned} p_i &= \begin{bmatrix} 0 & e_i^T & 0 \\ 0 & 0_n & 0 \\ 0 & 0 & 0 \end{bmatrix} & \text{for } 1 \leq i \leq n, \\ q_i &= \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0_n & e_i \\ 0 & 0 & 0 \end{bmatrix} & \text{for } 1 \leq i \leq n, \\ z &= \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0_n & 0 \\ 0 & 0 & 0 \end{bmatrix}, \end{aligned}$$

where $\{e_i\}$ is the standard basis of R^n . In other words, it has the basis $(p_1, p_2, \dots, p_n, q_1, q_2, \dots, q_n, z)$, where $p_i = E_{1,i+1}$, $q_i = E_{i+1,n+2}$ and $z = E_{1,n+2}$ are elementary matrices.

This Lie algebra is isomorphic to the n -th Heisenberg algebra constructed in [HeisenbergAlgebra](#); the bases correspond to each other.

INPUT:

- R – the base ring
- n – the nonnegative integer n

EXAMPLES:

```
sage: L = lie_algebras.Heisenberg(QQ, 1, representation='matrix')
sage: p = L.p(1)
sage: q = L.q(1)
sage: z = L.bracket(p, q); z
[0 0 1]
[0 0 0]
[0 0 0]
sage: z == L.z()
True
sage: L.dimension()
3

sage: L = lie_algebras.Heisenberg(QQ, 2, representation='matrix')
sage: sorted(dict(L.basis()).items())
[ (
    [0 1 0 0]
    [0 0 0 0]
    [0 0 0 0]
    'p1', [0 0 0 0]
),
  (
    [0 0 1 0]
    [0 0 0 0]
    [0 0 0 0]
```

(continues on next page)

(continued from previous page)

```

'p2', [0 0 0 0]
),
(
    [0 0 0 0]
    [0 0 0 1]
    [0 0 0 0]
'q1', [0 0 0 0]
),
(
    [0 0 0 0]
    [0 0 0 0]
    [0 0 0 1]
'q2', [0 0 0 0]
),
(
    [0 0 0 1]
    [0 0 0 0]
    [0 0 0 0]
'z', [0 0 0 0]
)]

sage: L = lie_algebras.Heisenberg(QQ, 0, representation='matrix')
sage: sorted(dict(L.basis()).items())
[(
    [0 1]
'z', [0 0]
)]
sage: L.gens()
(
[0 1]
[0 0]
)
sage: L.lie_algebra_generators()
Finite family {'z': [0 1]
[0 0]}

```

```

>>> from sage.all import *
>>> L = lie_algebras.Heisenberg(QQ, Integer(1), representation='matrix')
>>> p = L.p(Integer(1))
>>> q = L.q(Integer(1))
>>> z = L.bracket(p, q); z
[0 0 1]
[0 0 0]
[0 0 0]
>>> z == L.z()
True
>>> L.dimension()
3

>>> L = lie_algebras.Heisenberg(QQ, Integer(2), representation='matrix')
>>> sorted(dict(L.basis()).items())
[(

```

(continues on next page)

(continued from previous page)

```

        [0 1 0 0]
        [0 0 0 0]
        [0 0 0 0]
    'p1', [0 0 0 0]
),
(
    [0 0 1 0]
    [0 0 0 0]
    [0 0 0 0]
    'p2', [0 0 0 0]
),
(
    [0 0 0 0]
    [0 0 0 1]
    [0 0 0 0]
    'q1', [0 0 0 0]
),
(
    [0 0 0 0]
    [0 0 0 0]
    [0 0 0 1]
    'q2', [0 0 0 0]
),
(
    [0 0 0 1]
    [0 0 0 0]
    [0 0 0 0]
    'z', [0 0 0 0]
)]

>>> L = lie_algebras.Heisenberg(QQ, Integer(0), representation='matrix')
>>> sorted(dict(L.basis()).items())
[(
    [0 1]
    'z', [0 0]
)]
>>> L.gens()
(
    [0 1]
    [0 0]
)
>>> L.lie_algebra_generators()
Finite family {'z': [0 1]
[0 0]}

```

class ElementBases: *LieAlgebraMatrixWrapper, Element***monomial_coefficients** (*copy=True*)

Return a dictionary whose keys are indices of basis elements in the support of `self` and whose values are the corresponding coefficients.

INPUT:

- copy – ignored

EXAMPLES:

```
sage: L = lie_algebras.Heisenberg(QQ, 3, representation='matrix')
sage: elt = L(Matrix(QQ, [[0, 1, 3, 0, 3], [0, 0, 0, 0, 0], [0, 0, 0, 0, -
↪3],
.....:                [0, 0, 0, 0, 7], [0, 0, 0, 0, 0]]))
sage: elt
[ 0  1  3  0  3]
[ 0  0  0  0  0]
[ 0  0  0  0 -3]
[ 0  0  0  0  7]
[ 0  0  0  0  0]
sage: sorted(elt.monomial_coefficients().items())
[('p1', 1), ('p2', 3), ('q2', -3), ('q3', 7), ('z', 3)]
```

```
>>> from sage.all import *
>>> L = lie_algebras.Heisenberg(QQ, Integer(3), representation='matrix')
>>> elt = L(Matrix(QQ, [[Integer(0), Integer(1), Integer(3), Integer(0), ↪
↪Integer(3)], [Integer(0), Integer(0), Integer(0), Integer(0), ↪
↪Integer(0)], [Integer(0), Integer(0), Integer(0), Integer(0), -
↪Integer(3)],
...                [Integer(0), Integer(0), Integer(0), Integer(0), ↪
↪Integer(7)], [Integer(0), Integer(0), Integer(0), Integer(0), ↪
↪Integer(0)]]))
>>> elt
[ 0  1  3  0  3]
[ 0  0  0  0  0]
[ 0  0  0  0 -3]
[ 0  0  0  0  7]
[ 0  0  0  0  0]
>>> sorted(elt.monomial_coefficients().items())
[('p1', 1), ('p2', 3), ('q2', -3), ('q3', 7), ('z', 3)]
```

$p(i)$

Return the generator p_i of the Heisenberg algebra.

EXAMPLES:

```
sage: L = lie_algebras.Heisenberg(QQ, 1, representation='matrix')
sage: L.p(1)
[0 1 0]
[0 0 0]
[0 0 0]
```

```
>>> from sage.all import *
>>> L = lie_algebras.Heisenberg(QQ, Integer(1), representation='matrix')
>>> L.p(Integer(1))
[0 1 0]
[0 0 0]
[0 0 0]
```

$q(i)$

Return the generator q_i of the Heisenberg algebra.

EXAMPLES:

```
sage: L = lie_algebras.Heisenberg(QQ, 1, representation='matrix')
sage: L.q(1)
[0 0 0]
[0 0 1]
[0 0 0]
```

```
>>> from sage.all import *
>>> L = lie_algebras.Heisenberg(QQ, Integer(1), representation='matrix')
>>> L.q(Integer(1))
[0 0 0]
[0 0 1]
[0 0 0]
```

step()

Return the nilpotency step of self.

EXAMPLES:

```
sage: h = lie_algebras.Heisenberg(ZZ, 2, representation='matrix')
sage: h.step()
2
```

```
>>> from sage.all import *
>>> h = lie_algebras.Heisenberg(ZZ, Integer(2), representation='matrix')
>>> h.step()
2
```

z()

Return the basis element z of the Heisenberg algebra.

The element z spans the center of the Heisenberg algebra.

EXAMPLES:

```
sage: L = lie_algebras.Heisenberg(QQ, 1, representation='matrix')
sage: L.z()
[0 0 1]
[0 0 0]
[0 0 0]
```

```
>>> from sage.all import *
>>> L = lie_algebras.Heisenberg(QQ, Integer(1), representation='matrix')
>>> L.z()
[0 0 1]
[0 0 0]
[0 0 0]
```

class sage.algebras.lie_algebras.heisenberg.**InfiniteHeisenbergAlgebra**(R)

Bases: *HeisenbergAlgebra_abstract*, *LieAlgebraWithGenerators*

The infinite Heisenberg algebra.

This is the Heisenberg algebra on an infinite number of generators. In other words, this is the Heisenberg algebra of rank ∞ . See *HeisenbergAlgebra* for more information.

basis()

Return the basis of self.

EXAMPLES:

```
sage: L = lie_algebras.Heisenberg(QQ, oo)
sage: L.basis()
Lazy family (basis map(i))_{i in Disjoint union of Family ({'z'},
The Cartesian product of (Positive integers, {'p', 'q'}))}
sage: L.basis()['z']
z
sage: L.basis()[(12, 'p')]
p12
```

```
>>> from sage.all import *
>>> L = lie_algebras.Heisenberg(QQ, oo)
>>> L.basis()
Lazy family (basis map(i))_{i in Disjoint union of Family ({'z'},
The Cartesian product of (Positive integers, {'p', 'q'}))}
>>> L.basis()['z']
z
>>> L.basis()[(Integer(12), 'p')]
p12
```

lie_algebra_generators()

Return the generators of self as a Lie algebra.

EXAMPLES:

```
sage: L = lie_algebras.Heisenberg(QQ, oo)
sage: L.lie_algebra_generators()
Lazy family (generator map(i))_{i in The Cartesian product of
(Positive integers, {'p', 'q'})}
```

```
>>> from sage.all import *
>>> L = lie_algebras.Heisenberg(QQ, oo)
>>> L.lie_algebra_generators()
Lazy family (generator map(i))_{i in The Cartesian product of
(Positive integers, {'p', 'q'})}
```

9.2.11 Lie Algebras

AUTHORS:

- Travis Scrimshaw (2013-05-03): Initial version

```
class sage.algebras.lie_algebras.lie_algebra.FinitelyGeneratedLieAlgebra(R, names=None,
                                                                    index_set=None,
                                                                    category=None)
```

Bases: *LieAlgebraWithGenerators*

A finitely generated Lie algebra.

```
class sage.algebras.lie_algebras.lie_algebra.InfinatelyGeneratedLieAlgebra(R, names=None,
                                                                    index_set=None,
                                                                    category=None,
                                                                    prefix='L',
                                                                    **kws)
```

Bases: *LieAlgebraWithGenerators*

An infinitely generated Lie algebra.

```
class sage.algebras.lie_algebras.lie_algebra.LieAlgebra(R, names=None, category=None)
```

Bases: *Parent, UniqueRepresentation*

A Lie algebra L over a base ring R .

A Lie algebra is an R -module L with a bilinear operation called Lie bracket $[\cdot, \cdot] : L \times L \rightarrow L$ such that $[x, x] = 0$ and the following relation holds:

$$[x, [y, z]] + [y, [z, x]] + [z, [x, y]] = 0.$$

This relation is known as the *Jacobi identity* (or sometimes the Jacobi relation). We note that from $[x, x] = 0$, we have $[x + y, x + y] = 0$. Next from bilinearity, we see that

$$0 = [x + y, x + y] = [x, x] + [x, y] + [y, x] + [y, y] = [x, y] + [y, x],$$

thus $[x, y] = -[y, x]$ and the Lie bracket is antisymmetric.

Lie algebras are closely related to Lie groups. Let G be a Lie group and fix some $g \in G$. We can construct the Lie algebra L of G by considering the tangent space at g . We can also (partially) recover G from L by using what is known as the exponential map.

Given any associative algebra A , we can construct a Lie algebra L on the R -module A by defining the Lie bracket to be the commutator $[a, b] = ab - ba$. We call an associative algebra A which contains L in this fashion an *enveloping algebra* of L . The embedding $L \rightarrow A$ which sends the Lie bracket to the commutator will be called a Lie embedding. Now if we are given a Lie algebra L , we can construct an enveloping algebra U_L with Lie embedding $h : L \rightarrow U_L$ which has the following universal property: for any enveloping algebra A with Lie embedding $f : L \rightarrow A$, there exists a unique unital algebra homomorphism $g : U_L \rightarrow A$ such that $f = g \circ h$. The algebra U_L is known as the *universal enveloping algebra* of L .

INPUT:

See examples below for various input options.

EXAMPLES:

1. The simplest examples of Lie algebras are *abelian Lie algebras*. These are Lie algebras whose Lie bracket is (identically) zero. We can create them using the `abelian` keyword:

```
sage: L.<x,y,z> = LieAlgebra(QQ, abelian=True); L
Abelian Lie algebra on 3 generators (x, y, z) over Rational Field
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, abelian=True, names=('x', 'y', 'z')); (x, y, z) = L._
↳first_ngens(3); L
Abelian Lie algebra on 3 generators (x, y, z) over Rational Field
```

2. A Lie algebra can be built from any associative algebra by defining the Lie bracket to be the commutator. For example, we can start with the descent algebra:

```

sage: D = DescentAlgebra(QQ, 4).D()
sage: L = LieAlgebra(associative=D); L
Lie algebra of Descent algebra of 4 over Rational Field
in the standard basis
sage: L(D[2]).bracket(L(D[3]))
D{1, 2} - D{1, 3} + D{2} - D{3}

```

```

>>> from sage.all import *
>>> D = DescentAlgebra(QQ, Integer(4)).D()
>>> L = LieAlgebra(associative=D); L
Lie algebra of Descent algebra of 4 over Rational Field
in the standard basis
>>> L(D[Integer(2)]).bracket(L(D[Integer(3)]))
D{1, 2} - D{1, 3} + D{2} - D{3}

```

Next we use a free algebra and do some simple computations:

```

sage: R.<a,b,c> = FreeAlgebra(QQ, 3)
sage: L.<x,y,z> = LieAlgebra(associative=R.gens())
sage: x-y+z
a - b + c
sage: L.bracket(x-y, x-z)
a*b - a*c - b*a + b*c + c*a - c*b
sage: L.bracket(x-y, L.bracket(x,y))
a^2*b - 2*a*b*a + a*b^2 + b*a^2 - 2*b*a*b + b^2*a

```

```

>>> from sage.all import *
>>> R = FreeAlgebra(QQ, Integer(3), names=('a', 'b', 'c')); (a, b, c,) = R._
↳first_ngens(3)
>>> L = LieAlgebra(associative=R.gens(), names=('x', 'y', 'z')); (x, y, z,) = L._
↳first_ngens(3)
>>> x-y+z
a - b + c
>>> L.bracket(x-y, x-z)
a*b - a*c - b*a + b*c + c*a - c*b
>>> L.bracket(x-y, L.bracket(x,y))
a^2*b - 2*a*b*a + a*b^2 + b*a^2 - 2*b*a*b + b^2*a

```

We can also use a subset of the elements as a generating set of the Lie algebra:

```

sage: R.<a,b,c> = FreeAlgebra(QQ, 3)
sage: L.<x,y> = LieAlgebra(associative=[a,b+c])
sage: L.bracket(x, y)
a*b + a*c - b*a - c*a

```

```

>>> from sage.all import *
>>> R = FreeAlgebra(QQ, Integer(3), names=('a', 'b', 'c')); (a, b, c,) = R._
↳first_ngens(3)
>>> L = LieAlgebra(associative=[a,b+c], names=('x', 'y')); (x, y,) = L._first_
↳ngens(2)
>>> L.bracket(x, y)
a*b + a*c - b*a - c*a

```

Now for a more complicated example using the group ring of S_3 as our base algebra:

```
sage: G = SymmetricGroup(3)
sage: S = GroupAlgebra(G, QQ)
sage: L.<x,y> = LieAlgebra(associative=S.gens())
sage: L.bracket(x, y)
(2, 3) - (1, 3)
sage: L.bracket(x, y-x)
(2, 3) - (1, 3)
sage: L.bracket(L.bracket(x, y), y)
2*(1, 2, 3) - 2*(1, 3, 2)
sage: L.bracket(x, L.bracket(x, y))
(2, 3) - 2*(1, 2) + (1, 3)
sage: L.bracket(x, L.bracket(L.bracket(x, y), y))
0
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(3))
>>> S = GroupAlgebra(G, QQ)
>>> L = LieAlgebra(associative=S.gens(), names=('x', 'y')); (x, y) = L._first_
↳ngens(2)
>>> L.bracket(x, y)
(2, 3) - (1, 3)
>>> L.bracket(x, y-x)
(2, 3) - (1, 3)
>>> L.bracket(L.bracket(x, y), y)
2*(1, 2, 3) - 2*(1, 3, 2)
>>> L.bracket(x, L.bracket(x, y))
(2, 3) - 2*(1, 2) + (1, 3)
>>> L.bracket(x, L.bracket(L.bracket(x, y), y))
0
```

Here is an example using matrices:

```
sage: MS = MatrixSpace(QQ, 2)
sage: m1 = MS([[0, -1], [1, 0]])
sage: m2 = MS([[-1, 4], [3, 2]])
sage: L.<x,y> = LieAlgebra(associative=[m1, m2])
sage: x
[ 0 -1]
[ 1  0]
sage: y
[-1  4]
[ 3  2]
sage: L.bracket(x, y)
[-7 -3]
[-3  7]
sage: L.bracket(y, y)
[0 0]
[0 0]
sage: L.bracket(y, x)
[ 7  3]
[ 3 -7]
```

(continues on next page)

(continued from previous page)

```
sage: L.bracket(x, L.bracket(y, x))
[-6 14]
[14  6]
```

```
>>> from sage.all import *
>>> MS = MatrixSpace(QQ, Integer(2))
>>> m1 = MS([[Integer(0), -Integer(1)], [Integer(1), Integer(0)]])
>>> m2 = MS([[-Integer(1), Integer(4)], [Integer(3), Integer(2)]])
>>> L = LieAlgebra(associative=[m1, m2], names=('x', 'y')); (x, y,) = L._first_
↳ngens(2)
>>> x
[ 0 -1]
[ 1  0]
>>> y
[-1  4]
[ 3  2]
>>> L.bracket(x, y)
[-7 -3]
[-3  7]
>>> L.bracket(y, y)
[0 0]
[0 0]
>>> L.bracket(y, x)
[ 7  3]
[ 3 -7]
>>> L.bracket(x, L.bracket(y, x))
[-6 14]
[14  6]
```

(See [LieAlgebraFromAssociative](#) for other examples.)

3. We can also create a Lie algebra by inputting a set of structure coefficients. For example, we can create the Lie algebra of $\mathbf{Q}^3$ under the Lie bracket $\times$ (cross-product):

```
sage: d = {('x', 'y'): {'z': 1}, ('y', 'z'): {'x': 1}, ('z', 'x'): {'y': 1}}
sage: L.<x, y, z> = LieAlgebra(QQ, d)
sage: L
Lie algebra on 3 generators (x, y, z) over Rational Field
```

```
>>> from sage.all import *
>>> d = {('x', 'y'): {'z': Integer(1)}, ('y', 'z'): {'x': Integer(1)}, ('z', 'x'): {'y': Integer(1)}}
>>> L = LieAlgebra(QQ, d, names=('x', 'y', 'z')); (x, y, z,) = L._first_ngens(3)
>>> L
Lie algebra on 3 generators (x, y, z) over Rational Field
```

To compute the Lie bracket of two elements, you cannot use the `*` operator. Indeed, this automatically lifts up to the universal enveloping algebra and takes the (associative) product there. To get elements in the Lie algebra, you must use `bracket()`:

```
sage: L = LieAlgebra(QQ, {('e', 'h'): {'e': -2}, ('f', 'h'): {'f': 2},
....:                    ('e', 'f'): {'h': 1}}, names='e, f, h')
```

(continues on next page)

(continued from previous page)

```

sage: e,f,h = L.lie_algebra_generators()
sage: L.bracket(h, e)
2*e
sage: elt = h*e; elt
e*h + 2*e
sage: P = elt.parent(); P
Noncommutative Multivariate Polynomial Ring in e, f, h over Rational Field,
nc-relations: {...}
sage: R = P.relations()
sage: for rhs in sorted(R, key=str): print("{} = {}".format(rhs, R[rhs]))
f*e = e*f - h
h*e = e*h + 2*e
h*f = f*h - 2*f

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, {'e','h': {'e':-Integer(2)}, ('f','h'): {'f':Integer(2)},
...                    ('e','f'): {'h':Integer(1)}}), names='e,f,h')
>>> e,f,h = L.lie_algebra_generators()
>>> L.bracket(h, e)
2*e
>>> elt = h*e; elt
e*h + 2*e
>>> P = elt.parent(); P
Noncommutative Multivariate Polynomial Ring in e, f, h over Rational Field,
nc-relations: {...}
>>> R = P.relations()
>>> for rhs in sorted(R, key=str): print("{} = {}".format(rhs, R[rhs]))
f*e = e*f - h
h*e = e*h + 2*e
h*f = f*h - 2*f

```

For convenience, there are two shorthand notations for computing Lie brackets:

```

sage: L([h,e])
2*e
sage: L([h,[e,f]])
0
sage: L([[h,e],[e,f]])
-4*e
sage: L[h, e]
2*e
sage: L[h, L[e, f]]
0

```

```

>>> from sage.all import *
>>> L([h,e])
2*e
>>> L([h,[e,f]])
0
>>> L([[h,e],[e,f]])
-4*e
>>> L[h, e]

```

(continues on next page)

(continued from previous page)

```

2*e
>>> L[h, L[e, f]]
0

```

Warning

Because this is a modified (abused) version of python syntax, it does **NOT** work with addition. For example `L([e + [h, f], h])` and `L[e + [h, f], h]` will both raise errors. Instead you must use `L[e + L[h, f], h]`.

4. We can construct a Lie algebra from a Cartan type by using the `cartan_type` option:

```

sage: L = LieAlgebra(ZZ, cartan_type=['C', 3])
sage: L.inject_variables()
Defining e1, e2, e3, f1, f2, f3, h1, h2, h3
sage: e1.bracket(e2)
-E[alpha[1] + alpha[2]]
sage: L([[e1, e2], e2])
0
sage: L([[e2, e3], e3])
0
sage: L([e2, [e2, e3]])
2*E[2*alpha[2] + alpha[3]]

sage: L = LieAlgebra(ZZ, cartan_type=['E', 6])
sage: L
Lie algebra of ['E', 6] in the Chevalley basis

```

```

>>> from sage.all import *
>>> L = LieAlgebra(ZZ, cartan_type=['C', Integer(3)])
>>> L.inject_variables()
Defining e1, e2, e3, f1, f2, f3, h1, h2, h3
>>> e1.bracket(e2)
-E[alpha[1] + alpha[2]]
>>> L([[e1, e2], e2])
0
>>> L([[e2, e3], e3])
0
>>> L([e2, [e2, e3]])
2*E[2*alpha[2] + alpha[3]]

>>> L = LieAlgebra(ZZ, cartan_type=['E', Integer(6)])
>>> L
Lie algebra of ['E', 6] in the Chevalley basis

```

When the Cartan type is finite type and simply-laced, we can also specify an asymmetry function from [Ka1990] using a Dynkin diagram orientation with the `epsilon` option:

```

sage: L = LieAlgebra(QQ, cartan_type=['A', 2], epsilon=[(1, 2)])
sage: e1, e2 = L.e()
sage: L[e1, e2]

```

(continues on next page)

(continued from previous page)

```
-E[alpha[1] + alpha[2]]
```

```
sage: L = LieAlgebra(QQ, cartan_type=['A', 2], epsilon=[(2, 1)])
```

```
sage: e1, e2 = L.e()
```

```
sage: L[e1, e2]
```

```
E[alpha[1] + alpha[2]]
```

```
>>> from sage.all import *
```

```
>>> L = LieAlgebra(QQ, cartan_type=['A', Integer(2)], epsilon=[(Integer(1), Integer(2))])
```

```
>>> e1, e2 = L.e()
```

```
>>> L[e1, e2]
```

```
-E[alpha[1] + alpha[2]]
```

```
>>> L = LieAlgebra(QQ, cartan_type=['A', Integer(2)], epsilon=[(Integer(2), Integer(1))])
```

```
>>> e1, e2 = L.e()
```

```
>>> L[e1, e2]
```

```
E[alpha[1] + alpha[2]]
```

We also have matrix versions of the classical Lie algebras:

```
sage: L = LieAlgebra(ZZ, cartan_type=['A', 2], representation='matrix')
```

```
sage: L.gens()
```

```
(
[0 1 0] [0 0 0] [0 0 0] [0 0 0] [ 1 0 0] [ 0 0 0]
[0 0 0] [0 0 1] [1 0 0] [0 0 0] [ 0 -1 0] [ 0 1 0]
[0 0 0], [0 0 0], [0 0 0], [0 1 0], [ 0 0 0], [ 0 0 -1]
)
```

```
>>> from sage.all import *
```

```
>>> L = LieAlgebra(ZZ, cartan_type=['A', Integer(2)], representation='matrix')
```

```
>>> L.gens()
```

```
(
[0 1 0] [0 0 0] [0 0 0] [0 0 0] [ 1 0 0] [ 0 0 0]
[0 0 0] [0 0 1] [1 0 0] [0 0 0] [ 0 -1 0] [ 0 1 0]
[0 0 0], [0 0 0], [0 0 0], [0 1 0], [ 0 0 0], [ 0 0 -1]
)
```

There is also the compact real form of matrix Lie algebras implemented (the base ring must currently be a field):

```
sage: L = LieAlgebra(QQ, cartan_type=['A', 2], representation="compact real")
```

```
sage: list(L.basis())
```

```
[
[ 0 1 0] [ 0 0 1] [ 0 0 0] [ i 0 0] [0 i 0] [0 0 i]
[-1 0 0] [ 0 0 0] [ 0 0 1] [ 0 0 0] [i 0 0] [0 0 0]
[ 0 0 0], [-1 0 0], [ 0 -1 0], [ 0 0 -i], [0 0 0], [i 0 0],

[ 0 0 0] [0 0 0]
[ 0 i 0] [0 0 i]
[ 0 0 -i], [0 i 0]
]
```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, cartan_type=['A', Integer(2)], representation="compact real
↳")
>>> list(L.basis())
[
[ 0  1  0] [ 0  0  1] [ 0  0  0] [ i  0  0] [0 i 0] [0 0 i]
[-1  0  0] [ 0  0  0] [ 0  0  1] [ 0  0  0] [i 0 0] [0 0 0]
[ 0  0  0], [-1  0  0], [ 0 -1  0], [ 0  0 -i], [0 0 0], [i 0 0],
<BLANKLINE>
[ 0  0  0] [0 0 0]
[ 0  i  0] [0 0 i]
[ 0  0 -i], [0 i 0]
]

```

5. We construct a free Lie algebra in a few different ways. There are two primary representations, as brackets and as polynomials:

```

sage: L = LieAlgebra(QQ, 'x,y,z'); L
Free Lie algebra generated by (x, y, z) over Rational Field
sage: P.<a,b,c> = LieAlgebra(QQ, representation='polynomial'); P
Lie algebra generated by (a, b, c) in
Free Algebra on 3 generators (a, b, c) over Rational Field

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, 'x,y,z'); L
Free Lie algebra generated by (x, y, z) over Rational Field
>>> P = LieAlgebra(QQ, representation='polynomial', names=('a', 'b', 'c',)); (a,
↳b, c,) = P._first_ngens(3); P
Lie algebra generated by (a, b, c) in
Free Algebra on 3 generators (a, b, c) over Rational Field

```

This has the basis given by Hall and the one indexed by Lyndon words. We do some computations and convert between the bases:

```

sage: H = L.Hall()
doctest:warning...:
FutureWarning: The Hall basis has not been fully proven correct, but currently no
↳bugs are known
See https://github.com/sagemath/sage/issues/16823 for details.
sage: H
Free Lie algebra generated by (x, y, z) over Rational Field in the Hall basis
sage: Lyn = L.Lyndon()
sage: Lyn
Free Lie algebra generated by (x, y, z) over Rational Field in the Lyndon basis
sage: x,y,z = Lyn.lie_algebra_generators()
sage: a = Lyn([x, [[z, [x, y]], [y, x]]]); a
-[x, [[x, y], [x, [y, z]]]] - [x, [[x, y], [[x, z], y]]]
sage: H(a)
[[x, y], [z, [x, [x, y]]]] - [[x, y], [[x, y], [x, z]]]
+ [[x, [x, y]], [z, [x, y]]]

```

```

>>> from sage.all import *
>>> H = L.Hall()

```

(continues on next page)

(continued from previous page)

```

doctest:warning...:
FutureWarning: The Hall basis has not been fully proven correct, but currently no
↳ bugs are known
See https://github.com/sagemath/sage/issues/16823 for details.
>>> H
Free Lie algebra generated by (x, y, z) over Rational Field in the Hall basis
>>> Lyn = L.Lyndon()
>>> Lyn
Free Lie algebra generated by (x, y, z) over Rational Field in the Lyndon basis
>>> x,y,z = Lyn.lie_algebra_generators()
>>> a = Lyn([x, [[z, [x, y]], [y, x]]]); a
-[x, [[x, y], [x, [y, z]]]] - [x, [[x, y], [[x, z], y]]]
>>> H(a)
[[x, y], [z, [x, [x, y]]]] - [[x, y], [[x, y], [x, z]]]
+ [[x, [x, y]], [z, [x, y]]]

```

We also have the free Lie algebra given in the polynomial representation, which is the canonical embedding of the free Lie algebra into the free algebra (i.e., the ring of noncommutative polynomials). So the generators of the free Lie algebra are the generators of the free algebra and the Lie bracket is the commutator:

```

sage: P.<a,b,c> = LieAlgebra(QQ, representation='polynomial'); P
Lie algebra generated by (a, b, c) in
Free Algebra on 3 generators (a, b, c) over Rational Field
sage: P.bracket(a, b) + P.bracket(a - c, b + 3*c)
2*a*b + 3*a*c - 2*b*a + b*c - 3*c*a - c*b

```

```

>>> from sage.all import *
>>> P = LieAlgebra(QQ, representation='polynomial', names=('a', 'b', 'c')); (a,
↳ b, c) = P._first_ngens(3); P
Lie algebra generated by (a, b, c) in
Free Algebra on 3 generators (a, b, c) over Rational Field
>>> P.bracket(a, b) + P.bracket(a - c, b + Integer(3)*c)
2*a*b + 3*a*c - 2*b*a + b*c - 3*c*a - c*b

```

6. Nilpotent Lie algebras are Lie algebras such that there exists an integer s such that all iterated brackets of length longer than s are zero. They can be constructed from structural coefficients using the nilpotent keyword:

```

sage: L.<X,Y,Z> = LieAlgebra(QQ, {'X','Y': {'Z': 1}}, nilpotent=True)
sage: L
Nilpotent Lie algebra on 3 generators (X, Y, Z) over Rational Field
sage: L.category()
Category of finite dimensional nilpotent Lie algebras with basis over Rational
↳ Field

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, {'X','Y': {'Z': Integer(1)}}, nilpotent=True, names=('X',
↳ 'Y', 'Z')); (X, Y, Z) = L._first_ngens(3)
>>> L
Nilpotent Lie algebra on 3 generators (X, Y, Z) over Rational Field
>>> L.category()
Category of finite dimensional nilpotent Lie algebras with basis over Rational
↳ Field

```

A second example defining the Engel Lie algebra:

```
sage: sc = {('X','Y'): {'Z': 1}, ('X','Z'): {'W': 1}}
sage: E.<X,Y,Z,W> = LieAlgebra(QQ, sc, nilpotent=True); E
Nilpotent Lie algebra on 4 generators (X, Y, Z, W) over Rational Field
sage: E.step()
3
sage: E[X, Y + Z]
Z + W
sage: E[X, [X, Y + Z]]
W
sage: E[X, [X, [X, Y + Z]]]
0
```

```
>>> from sage.all import *
>>> sc = {('X','Y'): {'Z': Integer(1)}, ('X','Z'): {'W': Integer(1)}}
>>> E = LieAlgebra(QQ, sc, nilpotent=True, names=('X', 'Y', 'Z', 'W')); (X, Y, Z,
↪ W,) = E._first_ngens(4); E
Nilpotent Lie algebra on 4 generators (X, Y, Z, W) over Rational Field
>>> E.step()
3
>>> E[X, Y + Z]
Z + W
>>> E[X, [X, Y + Z]]
W
>>> E[X, [X, [X, Y + Z]]]
0
```

A nilpotent Lie algebra will also be constructed if given a category of a nilpotent Lie algebra:

```
sage: C = LieAlgebras(QQ).Nilpotent().FiniteDimensional().WithBasis()
sage: L.<X,Y,Z> = LieAlgebra(QQ, {('X','Y'): {'Z': 1}}, category=C); L
Nilpotent Lie algebra on 3 generators (X, Y, Z) over Rational Field
```

```
>>> from sage.all import *
>>> C = LieAlgebras(QQ).Nilpotent().FiniteDimensional().WithBasis()
>>> L = LieAlgebra(QQ, {('X','Y'): {'Z': Integer(1)}}, category=C, names=('X', 'Y',
↪ 'Z')); (X, Y, Z,) = L._first_ngens(3); L
Nilpotent Lie algebra on 3 generators (X, Y, Z) over Rational Field
```

7. Free nilpotent Lie algebras are the truncated versions of the free Lie algebras. That is, the only relations other than anticommutativity and the Jacobi identity among the Lie brackets are that brackets of length higher than the nilpotency step vanish. They can be created by using the `step` keyword:

```
sage: L = LieAlgebra(ZZ, 2, step=3); L
Free Nilpotent Lie algebra on 5 generators (X_1, X_2, X_12, X_112, X_122) over_
↪ Integer Ring
sage: L.step()
3
```

```
>>> from sage.all import *
>>> L = LieAlgebra(ZZ, Integer(2), step=Integer(3)); L
Free Nilpotent Lie algebra on 5 generators (X_1, X_2, X_12, X_112, X_122) over_
```

(continues on next page)

(continued from previous page)

```

→ Integer Ring
>>> L.step()
3

```

REFERENCES:

- [deG2000] Willem A. de Graaf. *Lie Algebras: Theory and Algorithms*.
- [Ka1990] Victor Kac, *Infinite dimensional Lie algebras*.
- [Wikipedia article Lie_algebra](#)

get_order()

Return an ordering of the basis indices.

Todo

Remove this method and in `CombinatorialFreeModule` in favor of a method in the category of (finite dimensional) modules with basis.

EXAMPLES:

```

sage: L.<x,y> = LieAlgebra(QQ, {})
sage: L.get_order()
('x', 'y')

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, {}, names=('x', 'y',)); (x, y,) = L._first_ngens(2)
>>> L.get_order()
('x', 'y')

```

monomial(i)

Return the monomial indexed by *i*.

EXAMPLES:

```

sage: L = lie_algebras.Heisenberg(QQ, oo)
sage: L.monomial('p1')
p1

```

```

>>> from sage.all import *
>>> L = lie_algebras.Heisenberg(QQ, oo)
>>> L.monomial('p1')
p1

```

term(i, c=None)

Return the term indexed by *i* with coefficient *c*.

EXAMPLES:

```

sage: L = lie_algebras.Heisenberg(QQ, oo)
sage: L.term('p1', 4)
4*p1

```

```
>>> from sage.all import *
>>> L = lie_algebras.Heisenberg(QQ, oo)
>>> L.term('p1', Integer(4))
4*p1
```

zero()

Return the element 0.

EXAMPLES:

```
sage: L.<x,y> = LieAlgebra(QQ, representation='polynomial')
sage: L.zero()
0
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, representation='polynomial', names=('x', 'y',)); (x, y,
↪) = L._first_ngens(2)
>>> L.zero()
0
```

```
class sage.algebras.lie_algebras.lie_algebra.LieAlgebraFromAssociative(A, gens=None,
                                                                    names=None,
                                                                    index_set=None,
                                                                    category=None)
```

Bases: *LieAlgebraWithGenerators*

A Lie algebra whose elements are from an associative algebra and whose bracket is the commutator.

Todo

Split this class into 2 classes, the base class for the Lie algebra corresponding to the full associative algebra and a subclass for the Lie subalgebra (of the full algebra) generated by a generating set?

Todo

Return the subalgebra generated by the basis elements of `self` for the universal enveloping algebra.

EXAMPLES:

For the first example, we start with a commutative algebra. Note that the bracket of everything will be 0:

```
sage: R = SymmetricGroupAlgebra(QQ, 2)
sage: L = LieAlgebra(associative=R)
sage: x, y = L.basis()
sage: L.bracket(x, y)
0
```

```
>>> from sage.all import *
>>> R = SymmetricGroupAlgebra(QQ, Integer(2))
>>> L = LieAlgebra(associative=R)
>>> x, y = L.basis()
```

(continues on next page)

(continued from previous page)

```
>>> L.bracket(x, y)
0
```

Next we use a free algebra and do some simple computations:

```
sage: R.<a,b> = FreeAlgebra(QQ, 2)
sage: L = LieAlgebra(associative=R)
sage: x,y = L(a), L(b)
sage: x-y
a - b
sage: L.bracket(x-y, x)
a*b - b*a
sage: L.bracket(x-y, L.bracket(x,y))
a^2*b - 2*a*b*a + a*b^2 + b*a^2 - 2*b*a*b + b^2*a
```

```
>>> from sage.all import *
>>> R = FreeAlgebra(QQ, Integer(2), names=('a', 'b')); (a, b,) = R._first_
↳ngens(2)
>>> L = LieAlgebra(associative=R)
>>> x,y = L(a), L(b)
>>> x-y
a - b
>>> L.bracket(x-y, x)
a*b - b*a
>>> L.bracket(x-y, L.bracket(x,y))
a^2*b - 2*a*b*a + a*b^2 + b*a^2 - 2*b*a*b + b^2*a
```

We can also use a subset of the generators as a generating set of the Lie algebra:

```
sage: R.<a,b,c> = FreeAlgebra(QQ, 3)
sage: L.<x,y> = LieAlgebra(associative=[a,b])
```

```
>>> from sage.all import *
>>> R = FreeAlgebra(QQ, Integer(3), names=('a', 'b', 'c')); (a, b, c,) = R._
↳first_ngens(3)
>>> L = LieAlgebra(associative=[a,b], names=('x', 'y')); (x, y,) = L._first_
↳ngens(2)
```

Now for a more complicated example using the group ring of S_3 as our base algebra:

```
sage: G = SymmetricGroup(3)
sage: S = GroupAlgebra(G, QQ)
sage: L.<x,y> = LieAlgebra(associative=S.gens())
sage: L.bracket(x, y)
(2,3) - (1,3)
sage: L.bracket(x, y-x)
(2,3) - (1,3)
sage: L.bracket(L.bracket(x, y), y)
2*(1,2,3) - 2*(1,3,2)
sage: L.bracket(x, L.bracket(x, y))
(2,3) - 2*(1,2) + (1,3)
sage: L.bracket(x, L.bracket(L.bracket(x, y), y))
```

(continues on next page)

(continued from previous page)

0

```

>>> from sage.all import *
>>> G = SymmetricGroup(Integer(3))
>>> S = GroupAlgebra(G, QQ)
>>> L = LieAlgebra(associative=S.gens(), names=('x', 'y')); (x, y) = L._first_
↳ngens(2)
>>> L.bracket(x, y)
(2, 3) - (1, 3)
>>> L.bracket(x, y-x)
(2, 3) - (1, 3)
>>> L.bracket(L.bracket(x, y), y)
2*(1, 2, 3) - 2*(1, 3, 2)
>>> L.bracket(x, L.bracket(x, y))
(2, 3) - 2*(1, 2) + (1, 3)
>>> L.bracket(x, L.bracket(L.bracket(x, y), y))
0

```

Here is an example using matrices:

```

sage: MS = MatrixSpace(QQ, 2)
sage: m1 = MS([[0, -1], [1, 0]])
sage: m2 = MS([[-1, 4], [3, 2]])
sage: L.<x,y> = LieAlgebra(associative=[m1, m2])
sage: x
[ 0 -1]
[ 1  0]
sage: y
[-1  4]
[ 3  2]
sage: L.bracket(x, y)
[-7 -3]
[-3  7]
sage: L.bracket(y, y)
[0 0]
[0 0]
sage: L.bracket(y, x)
[ 7  3]
[ 3 -7]
sage: L.bracket(x, L.bracket(y, x))
[-6 14]
[14  6]

```

```

>>> from sage.all import *
>>> MS = MatrixSpace(QQ, Integer(2))
>>> m1 = MS([Integer(0), -Integer(1)], [Integer(1), Integer(0)])
>>> m2 = MS([-Integer(1), Integer(4)], [Integer(3), Integer(2)])
>>> L = LieAlgebra(associative=[m1, m2], names=('x', 'y')); (x, y) = L._first_
↳ngens(2)
>>> x
[ 0 -1]
[ 1  0]

```

(continues on next page)

(continued from previous page)

```

>>> y
[-1  4]
[ 3  2]
>>> L.bracket(x,y)
[-7 -3]
[-3  7]
>>> L.bracket(y,y)
[0 0]
[0 0]
>>> L.bracket(y,x)
[ 7  3]
[ 3 -7]
>>> L.bracket(x, L.bracket(y,x))
[-6 14]
[14  6]

```

class ElementBases: *LieAlgebraElementWrapper***lift_associative()**

Lift *self* to the ambient associative algebra (which might be smaller than the universal enveloping algebra).

EXAMPLES:

```

sage: R = FreeAlgebra(QQ, 3, 'x,y,z')
sage: L.<x,y,z> = LieAlgebra(associative=R.gens())
sage: x.lift_associative()
x
sage: x.lift_associative().parent()
Free Algebra on 3 generators (x, y, z) over Rational Field

```

```

>>> from sage.all import *
>>> R = FreeAlgebra(QQ, Integer(3), 'x,y,z')
>>> L = LieAlgebra(associative=R.gens(), names=('x', 'y', 'z')); (x, y, z) = L._first_ngens(3)
>>> x.lift_associative()
x
>>> x.lift_associative().parent()
Free Algebra on 3 generators (x, y, z) over Rational Field

```

monomial_coefficients(copy=True)

Return the monomial coefficients of *self* (if this notion makes sense for *self.parent()*).

EXAMPLES:

```

sage: R.<x,y,z> = FreeAlgebra(QQ)
sage: L = LieAlgebra(associative=R)
sage: elt = L(x) + 2*L(y) - L(z)
sage: sorted(elt.monomial_coefficients().items())
[(x, 1), (y, 2), (z, -1)]

sage: L = LieAlgebra(associative=[x,y])

```

(continues on next page)

(continued from previous page)

```
sage: elt = L(x) + 2*L(y)
sage: elt.monomial_coefficients()
Traceback (most recent call last):
...
NotImplementedError: the basis is not defined
```

```
>>> from sage.all import *
>>> R = FreeAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z) = R._first_
↳ngens(3)
>>> L = LieAlgebra(associative=R)
>>> elt = L(x) + Integer(2)*L(y) - L(z)
>>> sorted(elt.monomial_coefficients().items())
[(x, 1), (y, 2), (z, -1)]

>>> L = LieAlgebra(associative=[x,y])
>>> elt = L(x) + Integer(2)*L(y)
>>> elt.monomial_coefficients()
Traceback (most recent call last):
...
NotImplementedError: the basis is not defined
```

associative_algebra()

Return the associative algebra used to construct self.

EXAMPLES:

```
sage: G = SymmetricGroup(3)
sage: S = GroupAlgebra(G, QQ)
sage: L = LieAlgebra(associative=S)
sage: L.associative_algebra() is S
True
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(3))
>>> S = GroupAlgebra(G, QQ)
>>> L = LieAlgebra(associative=S)
>>> L.associative_algebra() is S
True
```

is_abelian()

Return True if self is abelian.

EXAMPLES:

```
sage: R = FreeAlgebra(QQ, 2, 'x,y')
sage: L = LieAlgebra(associative=R.gens())
sage: L.is_abelian()
False

sage: R = PolynomialRing(QQ, 'x,y')
sage: L = LieAlgebra(associative=R.gens())
sage: L.is_abelian()
True
```

```

>>> from sage.all import *
>>> R = FreeAlgebra(QQ, Integer(2), 'x,y')
>>> L = LieAlgebra(associative=R.gens())
>>> L.is_abelian()
False

>>> R = PolynomialRing(QQ, 'x,y')
>>> L = LieAlgebra(associative=R.gens())
>>> L.is_abelian()
True

```

An example with a Lie algebra from the group algebra:

```

sage: G = SymmetricGroup(3)
sage: S = GroupAlgebra(G, QQ)
sage: L = LieAlgebra(associative=S)
sage: L.is_abelian()
False

```

```

>>> from sage.all import *
>>> G = SymmetricGroup(Integer(3))
>>> S = GroupAlgebra(G, QQ)
>>> L = LieAlgebra(associative=S)
>>> L.is_abelian()
False

```

Now we construct a Lie algebra from commuting elements in the group algebra:

```

sage: G = SymmetricGroup(5)
sage: S = GroupAlgebra(G, QQ)
sage: gens = map(S, [G((1, 2)), G((3, 4))])
sage: L.<x,y> = LieAlgebra(associative=gens)
sage: L.is_abelian()
True

```

```

>>> from sage.all import *
>>> G = SymmetricGroup(Integer(5))
>>> S = GroupAlgebra(G, QQ)
>>> gens = map(S, [G((Integer(1), Integer(2))), G((Integer(3), Integer(4)))]])
>>> L = LieAlgebra(associative=gens, names=('x', 'y')); (x, y) = L._first_
↳ngens(2)
>>> L.is_abelian()
True

```

lie_algebra_generators()

Return the Lie algebra generators of self.

EXAMPLES:

```

sage: G = SymmetricGroup(3)
sage: S = GroupAlgebra(G, QQ)
sage: L = LieAlgebra(associative=S)
sage: L.lie_algebra_generators()

```

(continues on next page)

(continued from previous page)

```
Finite family {(): (), (1,3,2): (1,3,2), (1,2,3): (1,2,3),
               (2,3): (2,3), (1,3): (1,3), (1,2): (1,2)}
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(3))
>>> S = GroupAlgebra(G, QQ)
>>> L = LieAlgebra(associative=S)
>>> L.lie_algebra_generators()
Finite family {(): (), (1,3,2): (1,3,2), (1,2,3): (1,2,3),
               (2,3): (2,3), (1,3): (1,3), (1,2): (1,2)}
```

monomial(*i*)Return the monomial indexed by *i*.

EXAMPLES:

```
sage: F.<x,y> = FreeAlgebra(QQ)
sage: L = LieAlgebra(associative=F)
sage: L.monomial(x.leading_support())
x
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, names=('x', 'y',)); (x, y,) = F._first_ngens(2)
>>> L = LieAlgebra(associative=F)
>>> L.monomial(x.leading_support())
x
```

term(*i*, *c=**None*)Return the term indexed by *i* with coefficient *c*.

EXAMPLES:

```
sage: F.<x,y> = FreeAlgebra(QQ)
sage: L = LieAlgebra(associative=F)
sage: L.term(x.leading_support(), 4)
4*x
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, names=('x', 'y',)); (x, y,) = F._first_ngens(2)
>>> L = LieAlgebra(associative=F)
>>> L.term(x.leading_support(), Integer(4))
4*x
```

zero()Return the element 0 in *self*.

EXAMPLES:

```
sage: G = SymmetricGroup(3)
sage: S = GroupAlgebra(G, QQ)
sage: L = LieAlgebra(associative=S)
sage: L.zero()
0
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(3))
>>> S = GroupAlgebra(G, QQ)
>>> L = LieAlgebra(associative=S)
>>> L.zero()
0
```

```
class sage.algebras.lie_algebras.lie_algebra.LieAlgebraWithGenerators(R, names=None,
                                                                    index_set=None,
                                                                    category=None,
                                                                    prefix='L', **kwds)
```

Bases: *LieAlgebra*

A Lie algebra with distinguished generators.

gen(*i*)

Return the *i*-th generator of *self*.

EXAMPLES:

```
sage: L.<x,y> = LieAlgebra(QQ, abelian=True)
sage: L.gen(0)
x
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, abelian=True, names=('x', 'y')); (x, y) = L._first_
↳ngens(2)
>>> L.gen(Integer(0))
x
```

gens()

Return a tuple whose entries are the generators for this object, in some order.

EXAMPLES:

```
sage: L.<x,y> = LieAlgebra(QQ, abelian=True)
sage: L.gens()
(x, y)
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, abelian=True, names=('x', 'y')); (x, y) = L._first_
↳ngens(2)
>>> L.gens()
(x, y)
```

indices()

Return the indices of *self*.

EXAMPLES:

```
sage: L.<x,y> = LieAlgebra(QQ, representation='polynomial')
sage: L.indices()
{'x', 'y'}
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, representation='polynomial', names=('x', 'y')); (x, y,
↩) = L._first_ngens(2)
>>> L.indices()
{'x', 'y'}
```

lie_algebra_generators()

Return the generators of `self` as a Lie algebra.

EXAMPLES:

```
sage: L.<x,y> = LieAlgebra(QQ, representation='polynomial')
sage: L.lie_algebra_generators()
Finite family {'x': x, 'y': y}
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, representation='polynomial', names=('x', 'y')); (x, y,
↩) = L._first_ngens(2)
>>> L.lie_algebra_generators()
Finite family {'x': x, 'y': y}
```

class `sage.algebras.lie_algebras.lie_algebra.LiftMorphismToAssociative` (*domain*, *codomain*)

Bases: `LiftMorphism`

The natural lifting morphism from a Lie algebra constructed from an associative algebra A to A .

preimage (*x*)

Return the preimage of *x* under `self`.

EXAMPLES:

```
sage: R = FreeAlgebra(QQ, 3, 'a,b,c')
sage: L = LieAlgebra(associative=R)
sage: x,y,z = R.gens()
sage: f = R.coerce_map_from(L)
sage: p = f.preimage(x*y - z); p
-c + a*b
sage: p.parent() is L
True
```

```
>>> from sage.all import *
>>> R = FreeAlgebra(QQ, Integer(3), 'a,b,c')
>>> L = LieAlgebra(associative=R)
>>> x,y,z = R.gens()
>>> f = R.coerce_map_from(L)
>>> p = f.preimage(x*y - z); p
-c + a*b
>>> p.parent() is L
True
```

section ()

Return the section map of `self`.

EXAMPLES:

```

sage: R = FreeAlgebra(QQ, 3, 'x,y,z')
sage: L.<x,y,z> = LieAlgebra(associative=R.gens())
sage: f = R.coerce_map_from(L)
sage: f.section()
Generic morphism:
  From: Free Algebra on 3 generators (x, y, z) over Rational Field
  To:   Lie algebra generated by (x, y, z) in Free Algebra on 3 generators (x,
↪ y, z) over Rational Field

```

```

>>> from sage.all import *
>>> R = FreeAlgebra(QQ, Integer(3), 'x,y,z')
>>> L = LieAlgebra(associative=R.gens(), names=('x', 'y', 'z')); (x, y, z).␣
↪ L._first_ngens(3)
>>> f = R.coerce_map_from(L)
>>> f.section()
Generic morphism:
  From: Free Algebra on 3 generators (x, y, z) over Rational Field
  To:   Lie algebra generated by (x, y, z) in Free Algebra on 3 generators (x,
↪ y, z) over Rational Field

```

```

class sage.algebras.lie_algebras.lie_algebra.MatrixLieAlgebraFromAssociative(A,
                                                                              gens=None,
                                                                              names=None,
                                                                              in-
                                                                              dex_set=None,
                                                                              cate-
                                                                              gory=None)

```

Bases: *LieAlgebraFromAssociative*

A Lie algebra constructed from a matrix algebra.

This means a Lie algebra consisting of matrices, with commutator as Lie bracket.

class Element

Bases: *LieAlgebraMatrixWrapper, Element*

matrix()

Return *self* as element of the underlying matrix algebra.

OUTPUT: an instance of the element class of MatrixSpace

EXAMPLES:

```

sage: sl3m = lie_algebras.sl(ZZ, 3, representation='matrix')
sage: e1,e2, f1, f2, h1, h2 = sl3m.gens()
sage: h1m = h1.matrix(); h1m
[ 1  0  0]
[ 0 -1  0]
[ 0  0  0]
sage: h1m.parent()
Full MatrixSpace of 3 by 3 sparse matrices over Integer Ring
sage: matrix(h2)
[ 0  0  0]
[ 0  1  0]
[ 0  0 -1]

```

(continues on next page)

(continued from previous page)

```

sage: L = lie_algebras.so(QQ['z'], 5, representation='matrix')
sage: matrix(L.an_element())
[ 1  1  0  0  0]
[ 1  1  0  0  2]
[ 0  0 -1 -1  0]
[ 0  0 -1 -1 -1]
[ 0  1  0 -2  0]

sage: gl2 = lie_algebras.gl(QQ, 2)
sage: matrix(gl2.an_element())
[1 1]
[1 1]

```

```

>>> from sage.all import *
>>> sl3m = lie_algebras.sl(ZZ, Integer(3), representation='matrix')
>>> e1,e2, f1, f2, h1, h2 = sl3m.gens()
>>> h1m = h1.matrix(); h1m
[ 1  0  0]
[ 0 -1  0]
[ 0  0  0]
>>> h1m.parent()
Full MatrixSpace of 3 by 3 sparse matrices over Integer Ring
>>> matrix(h2)
[ 0  0  0]
[ 0  1  0]
[ 0  0 -1]
>>> L = lie_algebras.so(QQ['z'], Integer(5), representation='matrix')
>>> matrix(L.an_element())
[ 1  1  0  0  0]
[ 1  1  0  0  2]
[ 0  0 -1 -1  0]
[ 0  0 -1 -1 -1]
[ 0  1  0 -2  0]

>>> gl2 = lie_algebras.gl(QQ, Integer(2))
>>> matrix(gl2.an_element())
[1 1]
[1 1]

```

9.2.12 Lie Algebra Elements

AUTHORS:

- Travis Scrimshaw (2013-05-04): Initial implementation

class sage.algebras.lie_algebras.lie_algebra_element.**FreeLieAlgebraElement**

Bases: *LieAlgebraElement*

An element of a free Lie algebra.

lift()

Lift *self* to the universal enveloping algebra.

EXAMPLES:

```

sage: L = LieAlgebra(QQ, 'x,y,z')
sage: Lyn = L.Lyndon()
sage: x,y,z = Lyn.gens()
sage: a = Lyn([z, [[x, y], x]]); a
[x, [x, [y, z]]] + [x, [[x, z], y]] - [[x, y], [x, z]]
sage: a.lift()
x^2*y*z - 2*x*y*x*z + y*x^2*z - z*x^2*y + 2*z*x*y*x - z*y*x^2

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, 'x,y,z')
>>> Lyn = L.Lyndon()
>>> x,y,z = Lyn.gens()
>>> a = Lyn([z, [[x, y], x]]); a
[x, [x, [y, z]]] + [x, [[x, z], y]] - [[x, y], [x, z]]
>>> a.lift()
x^2*y*z - 2*x*y*x*z + y*x^2*z - z*x^2*y + 2*z*x*y*x - z*y*x^2

```

list()

Return *self* as a list of pairs (*m*, *c*) where *m* is a basis key (i.e., a key of one of the basis elements) and *c* is its coefficient.

This list is sorted from highest to lowest degree.

EXAMPLES:

```

sage: L.<x, y> = LieAlgebra(QQ)
sage: elt = x + L.bracket(y, x)
sage: elt.list()
[[x, y], -1), (x, 1)]

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, names=('x', 'y',)); (x, y) = L._first_ngens(2)
>>> elt = x + L.bracket(y, x)
>>> elt.list()
[[x, y], -1), (x, 1)]

```

class sage.algebras.lie_algebras.lie_algebra_element.**GradedLieBracket**

Bases: *LieBracket*

A Lie bracket (*LieBracket*) for a graded Lie algebra.

Unlike the vanilla Lie bracket class, this also stores a degree, and uses it as a first criterion when comparing graded Lie brackets. (Graded Lie brackets still compare greater than Lie generators.)

class sage.algebras.lie_algebras.lie_algebra_element.**LieAlgebraElement**

Bases: *IndexedFreeModuleElement*

A Lie algebra element.

lift()

Lift *self* to the universal enveloping algebra.

EXAMPLES:

```
sage: L.<x,y,z> = LieAlgebra(QQ, {'x','y':{'z':1}})
sage: x.lift().parent() == L.universal_enveloping_algebra()
True
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, {'x','y':{'z':Integer(1)}}, names=('x', 'y', 'z'));
↳ (x, y, z,) = L._first_ngens(3)
>>> x.lift().parent() == L.universal_enveloping_algebra()
True
```

class sage.algebras.lie_algebras.lie_algebra_element.**LieAlgebraElementWrapper**

Bases: [ElementWrapper](#)

Wrap an element as a Lie algebra element.

class sage.algebras.lie_algebras.lie_algebra_element.**LieAlgebraMatrixWrapper**

Bases: [LieAlgebraElementWrapper](#)

Lie algebra element wrapper around a matrix.

class sage.algebras.lie_algebras.lie_algebra_element.**LieBracket**

Bases: [LieObject](#)

An abstract Lie bracket (formally, just a binary tree).

lift (UEA_gens_dict)

Lift self to the universal enveloping algebra.

UEA_gens_dict should be the dictionary for the generators of the universal enveloping algebra.

EXAMPLES:

```
sage: L = LieAlgebra(QQ, 'x,y,z')
sage: Lyn = L.Lyndon()
sage: x,y,z = Lyn.gens()
sage: a = Lyn([z, [[x, y], x]]); a
[x, [x, [y, z]]] + [x, [[x, z], y]] - [[x, y], [x, z]]
sage: a.lift() # indirect doctest
x^2*y*z - 2*x*y*x*z + y*x^2*z - z*x^2*y + 2*z*x*y*x - z*y*x^2
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, 'x,y,z')
>>> Lyn = L.Lyndon()
>>> x,y,z = Lyn.gens()
>>> a = Lyn([z, [[x, y], x]]); a
[x, [x, [y, z]]] + [x, [[x, z], y]] - [[x, y], [x, z]]
>>> a.lift() # indirect doctest
x^2*y*z - 2*x*y*x*z + y*x^2*z - z*x^2*y + 2*z*x*y*x - z*y*x^2
```

to_word()

Return the word (“flattening”) of self.

If self is a tree of Lie brackets, this word is usually obtained by “forgetting the brackets”.

EXAMPLES:

```
sage: from sage.algebras.lie_algebras.lie_algebra_element import LieGenerator,
↳ LieBracket
sage: x = LieGenerator('x', 0)
sage: y = LieGenerator('y', 1)
sage: b = LieBracket(x, y)
sage: c = LieBracket(b, x)
sage: c.to_word()
('x', 'y', 'x')
```

```
>>> from sage.all import *
>>> from sage.algebras.lie_algebras.lie_algebra_element import LieGenerator,
↳ LieBracket
>>> x = LieGenerator('x', Integer(0))
>>> y = LieGenerator('y', Integer(1))
>>> b = LieBracket(x, y)
>>> c = LieBracket(b, x)
>>> c.to_word()
('x', 'y', 'x')
```

class sage.algebras.lie_algebras.lie_algebra_element.**LieGenerator**

Bases: *LieObject*

A wrapper around an object so it can ducktype with and do comparison operations with *LieBracket*.

lift(*UEA_gens_dict*)

Lift self to the universal enveloping algebra.

UEA_gens_dict should be the dictionary for the generators of the universal enveloping algebra.

EXAMPLES:

```
sage: L = LieAlgebra(QQ, 'x,y,z')
sage: Lyn = L.Lyndon()
sage: x,y,z = Lyn.gens()
sage: x.lift()
x
sage: x.lift().parent()
Free Algebra on 3 generators (x, y, z) over Rational Field
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, 'x,y,z')
>>> Lyn = L.Lyndon()
>>> x,y,z = Lyn.gens()
>>> x.lift()
x
>>> x.lift().parent()
Free Algebra on 3 generators (x, y, z) over Rational Field
```

to_word()

Return the word (“flattening”) of self.

If self is a tree of Lie brackets, this word is usually obtained by “forgetting the brackets”.

EXAMPLES:

```
sage: from sage.algebras.lie_algebras.lie_algebra_element import LieGenerator
sage: x = LieGenerator('x', 0)
sage: x.to_word()
('x',)
```

```
>>> from sage.all import *
>>> from sage.algebras.lie_algebras.lie_algebra_element import LieGenerator
>>> x = LieGenerator('x', Integer(0))
>>> x.to_word()
('x',)
```

class sage.algebras.lie_algebras.lie_algebra_element.**LieObject**

Bases: SageObject

Abstract base class for *LieGenerator* and *LieBracket*.

to_word()

Return the word (“flattening”) of self.

If self is a tree of Lie brackets, this word is usually obtained by “forgetting the brackets”.

class sage.algebras.lie_algebras.lie_algebra_element.**LieSubalgebraElementWrapper**

Bases: *LieAlgebraElementWrapper*

Wrap an element of the ambient Lie algebra as an element.

monomial_coefficients (*copy=True*)

Return a dictionary whose keys are indices of basis elements in the support of self and whose values are the corresponding coefficients.

INPUT:

- *copy* – boolean (default: True); if self is internally represented by a dictionary d, then make a copy of d; if False, then this can cause undesired behavior by mutating d

EXAMPLES:

```
sage: L.<X,Y,Z> = LieAlgebra(ZZ, {'X','Y': {'Z': 3}})
sage: S = L.subalgebra([X, Y])
sage: S(2*Y + 9*Z).monomial_coefficients()
{1: 2, 2: 3}
sage: S2 = L.subalgebra([Y, Z])
sage: S2(2*Y + 9*Z).monomial_coefficients()
{0: 2, 1: 9}
```

```
>>> from sage.all import *
>>> L = LieAlgebra(ZZ, {'X','Y': {'Z': Integer(3)}}, names=('X', 'Y', 'Z',
↪ )); (X, Y, Z,) = L._first_ngens(3)
>>> S = L.subalgebra([X, Y])
>>> S(Integer(2)*Y + Integer(9)*Z).monomial_coefficients()
{1: 2, 2: 3}
>>> S2 = L.subalgebra([Y, Z])
>>> S2(Integer(2)*Y + Integer(9)*Z).monomial_coefficients()
{0: 2, 1: 9}
```

to_vector (*sparse=False, order=None*)

Return the vector in `g.module()` corresponding to the element `self` of `g` (where `g` is the parent of `self`).

EXAMPLES:

```
sage: L.<X,Y,Z> = LieAlgebra(ZZ, {'X','Y': {'Z': 3}})
sage: S = L.subalgebra([X, Y])
sage: S.basis()
Family (X, Y, 3*Z)
sage: S(2*Y + 9*Z).to_vector()
(0, 2, 9)
sage: S2 = L.subalgebra([Y, Z])
sage: S2.basis()
Family (Y, Z)
sage: S2(2*Y + 9*Z).to_vector()
(0, 2, 9)
```

```
>>> from sage.all import *
>>> L = LieAlgebra(ZZ, {'X','Y': {'Z': Integer(3)}}, names=('X', 'Y', 'Z',
↳ )); (X, Y, Z,) = L._first_ngens(3)
>>> S = L.subalgebra([X, Y])
>>> S.basis()
Family (X, Y, 3*Z)
>>> S(Integer(2)*Y + Integer(9)*Z).to_vector()
(0, 2, 9)
>>> S2 = L.subalgebra([Y, Z])
>>> S2.basis()
Family (Y, Z)
>>> S2(Integer(2)*Y + Integer(9)*Z).to_vector()
(0, 2, 9)
```

class `sage.algebras.lie_algebras.lie_algebra_element.LyndonBracket`

Bases: *GradedLieBracket*

A Lie bracket (*LieBracket*) tailored for the Lyndon basis.

The order on these brackets is defined by $l < r$ if $w(l) < w(r)$, where $w(l)$ is the word corresponding to l . (This is also true if one or both of l and r is a *LieGenerator*.)

class `sage.algebras.lie_algebras.lie_algebra_element.StructureCoefficientsElement`

Bases: *LieAlgebraMatrixWrapper*

An element of a Lie algebra given by structure coefficients.

bracket (*right*)

Return the Lie bracket `[self, right]`.

EXAMPLES:

```
sage: L.<x,y,z> = LieAlgebra(QQ, {'x','y': {'z':1}, ('y','z'): {'x':1}, ('z
↳ ','x'): {'y':1}})
sage: x.bracket(y)
z
sage: y.bracket(x)
-z
```

(continues on next page)

(continued from previous page)

```
sage: (x + y - z).bracket(x - y + z)
-2*y - 2*z
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, {'x','y': {'z':Integer(1)}, ('y','z'): {'x':Integer(1)}, ('z','x'): {'y':Integer(1)}}, names=('x', 'y', 'z')); (x, y, z) = L._first_ngens(3)
>>> x.bracket(y)
z
>>> y.bracket(x)
-z
>>> (x + y - z).bracket(x - y + z)
-2*y - 2*z
```

lift()Return the lift of `self` to the universal enveloping algebra.

EXAMPLES:

```
sage: L.<x,y> = LieAlgebra(QQ, {'x','y': {'x':1}})
sage: elt = x - 3/2 * y
sage: l = elt.lift(); l
x - 3/2*y
sage: l.parent()
Noncommutative Multivariate Polynomial Ring in x, y
over Rational Field, nc-relations: {y*x: x*y - x}
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, {'x','y': {'x':Integer(1)}}, names=('x', 'y')); (x, y) = L._first_ngens(2)
>>> elt = x - Integer(3)/Integer(2) * y
>>> l = elt.lift(); l
x - 3/2*y
>>> l.parent()
Noncommutative Multivariate Polynomial Ring in x, y
over Rational Field, nc-relations: {y*x: x*y - x}
```

monomial_coefficients (*copy=True*)Return the monomial coefficients of `self` as a dictionary.

EXAMPLES:

```
sage: L.<x,y,z> = LieAlgebra(QQ, {'x','y': {'z':1}})
sage: a = 2*x - 3/2*y + z
sage: a.monomial_coefficients()
{'x': 2, 'y': -3/2, 'z': 1}
sage: a = 2*x - 3/2*z
sage: a.monomial_coefficients()
{'x': 2, 'z': -3/2}
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, {'x','y': {'z':Integer(1)}}, names=('x', 'y', 'z')); (x, y, z) = L._first_ngens(3)
```

(continues on next page)

(continued from previous page)

```

↪ (x, y, z,) = L._first_ngens(3)
>>> a = Integer(2)*x - Integer(3)/Integer(2)*y + z
>>> a.monomial_coefficients()
{'x': 2, 'y': -3/2, 'z': 1}
>>> a = Integer(2)*x - Integer(3)/Integer(2)*z
>>> a.monomial_coefficients()
{'x': 2, 'z': -3/2}

```

to_vector (*sparse=False, order=None*)

Return self as a vector.

EXAMPLES:

```

sage: L.<x,y,z> = LieAlgebra(QQ, {'x','y': {'z':1}})
sage: a = x + 3*y - z/2
sage: a.to_vector()
(1, 3, -1/2)

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, {'x','y': {'z':Integer(1)}}, names=('x', 'y', 'z'));
↪ (x, y, z,) = L._first_ngens(3)
>>> a = x + Integer(3)*y - z/Integer(2)
>>> a.to_vector()
(1, 3, -1/2)

```

class sage.algebras.lie_algebras.lie_algebra_element.**UntwistedAffineLieAlgebraElement**

Bases: `Element`

An element of an untwisted affine Lie algebra.

bracket (*right*)

Return the Lie bracket [self, right].

EXAMPLES:

```

sage: L = LieAlgebra(QQ, cartan_type=['A',1,1])
sage: e1,f1,h1,e0,f0,c,d = list(L.lie_algebra_generators())
sage: e0.bracket(f0)
(-h1)#t^0 + 4*c
sage: e1.bracket(0)
0
sage: e1.bracket(1)
Traceback (most recent call last):
...
TypeError: no common canonical parent for objects with parents:
'Affine Kac-Moody algebra of ['A', 1] in the Chevalley basis'
and 'Integer Ring'

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, cartan_type=['A',Integer(1),Integer(1)])
>>> e1,f1,h1,e0,f0,c,d = list(L.lie_algebra_generators())
>>> e0.bracket(f0)
(-h1)#t^0 + 4*c

```

(continues on next page)

(continued from previous page)

```
>>> e1.bracket(Integer(0))
0
>>> e1.bracket(Integer(1))
Traceback (most recent call last):
...
TypeError: no common canonical parent for objects with parents:
  'Affine Kac-Moody algebra of ['A', 1] in the Chevalley basis'
and 'Integer Ring'
```

c_coefficient()

Return the coefficient of c of self.

EXAMPLES:

```
sage: L = lie_algebras.Affine(QQ, ['A', 1, 1])
sage: x = L.an_element() - 3 * L.c()
sage: x.c_coefficient()
-2
```

```
>>> from sage.all import *
>>> L = lie_algebras.Affine(QQ, ['A', Integer(1), Integer(1)])
>>> x = L.an_element() - Integer(3) * L.c()
>>> x.c_coefficient()
-2
```

canonical_derivation()

Return the canonical derivation d applied to self.

The canonical derivation d is defined as

$$d(a \otimes t^m + \alpha c) = a \otimes m t^{m-1}.$$

Another formulation is by $d = t \frac{d}{dt}$.

EXAMPLES:

```
sage: L = lie_algebras.Affine(QQ, ['E', 6, 1])
sage: al = RootSystem(['E', 6]).root_lattice().simple_roots()
sage: x = L.basis()[al[2]+al[3]+2*al[4]+al[5], 5] + 4*L.c() + L.d()
sage: x.canonical_derivation()
(5*E[alpha[2] + alpha[3] + 2*alpha[4] + alpha[5]])#t^5
```

```
>>> from sage.all import *
>>> L = lie_algebras.Affine(QQ, ['E', Integer(6), Integer(1)])
>>> al = RootSystem(['E', Integer(6)]).root_lattice().simple_roots()
>>> x = L.
↳ basis()[al[Integer(2)]+al[Integer(3)]+Integer(2)*al[Integer(4)]+al[Integer(5)],
↳ Integer(5)] + Integer(4)*L.c() + L.d()
>>> x.canonical_derivation()
(5*E[alpha[2] + alpha[3] + 2*alpha[4] + alpha[5]])#t^5
```

d_coefficient()

Return the coefficient of d of self.

EXAMPLES:

```
sage: L = lie_algebras.Affine(QQ, ['A',1,1])
sage: x = L.an_element() + L.d()
sage: x.d_coefficient()
2
```

```
>>> from sage.all import *
>>> L = lie_algebras.Affine(QQ, ['A',Integer(1),Integer(1)])
>>> x = L.an_element() + L.d()
>>> x.d_coefficient()
2
```

monomial_coefficients (*copy=True*)

Return the monomial coefficients of *self*.

EXAMPLES:

```
sage: L = lie_algebras.Affine(QQ, ['C',2,1])
sage: x = L.an_element()
sage: sorted(x.monomial_coefficients(), key=str)
[(-2*alpha[1] - alpha[2], 1),
 (-alpha[1], 0),
 (-alpha[2], 0),
 (2*alpha[1] + alpha[2], -1),
 (alpha[1], 0),
 (alpha[2], 0),
 (alphacheck[1], 0),
 (alphacheck[2], 0),
 'c',
 'd']
```

```
>>> from sage.all import *
>>> L = lie_algebras.Affine(QQ, ['C',Integer(2),Integer(1)])
>>> x = L.an_element()
>>> sorted(x.monomial_coefficients(), key=str)
[(-2*alpha[1] - alpha[2], 1),
 (-alpha[1], 0),
 (-alpha[2], 0),
 (2*alpha[1] + alpha[2], -1),
 (alpha[1], 0),
 (alpha[2], 0),
 (alphacheck[1], 0),
 (alphacheck[2], 0),
 'c',
 'd']
```

t_dict ()

Return the dict, whose keys are powers of *t* and values are elements of the classical Lie algebra, of *self*.

EXAMPLES:

```
sage: L = lie_algebras.Affine(QQ, ['A',1,1])
sage: x = L.an_element()
sage: x.t_dict()
```

(continues on next page)

(continued from previous page)

```
{-1: E[alpha[1]],
 0: E[alpha[1]] + h1 + E[-alpha[1]],
 1: E[-alpha[1]]}
```

```
>>> from sage.all import *
>>> L = lie_algebras.Affine(QQ, ['A', Integer(1), Integer(1)])
>>> x = L.an_element()
>>> x.t_dict()
{-1: E[alpha[1]],
 0: E[alpha[1]] + h1 + E[-alpha[1]],
 1: E[-alpha[1]]}
```

9.2.13 Homomorphisms of Lie Algebras

AUTHORS:

- Travis Scrimshaw (07-15-2013): Initial implementation
- Eero Hakavuori (08-09-2018): Morphisms defined by a generating subset

```
class sage.algebras.lie_algebras.morphism.LieAlgebraHomomorphism_im_gens (parent, im_gens,
                                                                              base_map=None,
                                                                              check=True)
```

Bases: `Morphism`

A homomorphism of Lie algebras.

Let $\mathfrak{g}$ and $\mathfrak{g}'$ be Lie algebras. A linear map $f: \mathfrak{g} \rightarrow \mathfrak{g}'$ is a homomorphism (of Lie algebras) if $f([x, y]) = [f(x), f(y)]$ for all $x, y \in \mathfrak{g}$. Thus homomorphisms are completely determined by the image of the generators of $\mathfrak{g}$.

INPUT:

- `parent` – a homset between two Lie algebras
- `im_gens` – the image of the generators of the domain
- `base_map` – a homomorphism to apply to the coefficients. It should be a map from the base ring of the domain to the base ring of the codomain. Note that if `base_map` is nontrivial then the result will not be a morphism in the category of Lie algebras over the base ring.
- `check` – whether to run checks on the validity of the defining data

EXAMPLES:

```
sage: L = LieAlgebra(QQ, 'x,y,z')
sage: Lyn = L.Lyndon()
sage: H = L.Hall()
doctest:warning...:
FutureWarning: The Hall basis has not been fully proven correct, but currently no
↳ bugs are known
See https://github.com/sagemath/sage/issues/16823 for details.
sage: phi = Lyn.coerce_map_from(H); phi
Lie algebra morphism:
  From: Free Lie algebra generated by (x, y, z) over Rational Field in the Hall
↳ basis
  To:   Free Lie algebra generated by (x, y, z) over Rational Field in the Lyndon
↳ basis
```

(continues on next page)

(continued from previous page)

```

↪basis
Defn: x |--> x
      y |--> y
      z |--> z

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, 'x,y,z')
>>> Lyn = L.Lyndon()
>>> H = L.Hall()
doctest:warning...:
FutureWarning: The Hall basis has not been fully proven correct, but currently no
↪bugs are known
See https://github.com/sagemath/sage/issues/16823 for details.
>>> phi = Lyn.coerce_map_from(H); phi
Lie algebra morphism:
  From: Free Lie algebra generated by (x, y, z) over Rational Field in the Hall
↪basis
  To:   Free Lie algebra generated by (x, y, z) over Rational Field in the Lyndon
↪basis
Defn: x |--> x
      y |--> y
      z |--> z

```

You can provide a base map, creating a semilinear map that (sometimes) preserves the Lie bracket:

```

sage: R.<x> = ZZ[]
sage: K.<i> = NumberField(x^2 + 1)
sage: cc = K.hom([-i])
sage: L.<X,Y,Z,W> = LieAlgebra(K, {'X','Y': {'Z':1}, ('X','Z'): {'W':1}})
sage: M.<A,B,C,D> = LieAlgebra(K, {'A','B': {'C':1}, ('A','C'): {'D':1}})
sage: phi = L.morphism({X:A, Y:B, Z:C, W:D}, base_map=cc)
sage: phi(X)
A
sage: phi(i*X)
-i*A
sage: all(phi(x.bracket(y)) == phi(x).bracket(phi(y)) for x,y in cartesian_
↪product_iterator([X,Y,Z,W], [X,Y,Z,W]))
True

```

```

>>> from sage.all import *
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> K = NumberField(x**Integer(2) + Integer(1), names=('i',)); (i,) = K._first_
↪ngens(1)
>>> cc = K.hom([-i])
>>> L = LieAlgebra(K, {'X','Y': {'Z':Integer(1)}, ('X','Z'): {'W':Integer(1)}},
↪names=('X', 'Y', 'Z', 'W')); (X, Y, Z, W,) = L._first_ngens(4)
>>> M = LieAlgebra(K, {'A','B': {'C':Integer(1)}, ('A','C'): {'D':Integer(1)}},
↪names=('A', 'B', 'C', 'D')); (A, B, C, D,) = M._first_ngens(4)
>>> phi = L.morphism({X:A, Y:B, Z:C, W:D}, base_map=cc)
>>> phi(X)
A
>>> phi(i*X)

```

(continues on next page)

(continued from previous page)

```

-i*A
>>> all(phi(x.bracket(y)) == phi(x).bracket(phi(y)) for x,y in cartesian_product_
↪ iterator([[X,Y,Z,W],[X,Y,Z,W]]))
True

```

Note that the Lie bracket should still be preserved, even though the map is no longer linear over the base ring:

```

sage: L.<X,Y,Z,W> = LieAlgebra(K, {'X','Y': {'Z':1}, ('X','Z'): {'W':1}})
sage: M.<A,B,C,D> = LieAlgebra(K, {'A','B': {'C':-1}, ('A','C'): {'D':1}})
sage: phi = L.morphism({X:A, Y:B, Z:C, W:D}, base_map=cc)
sage: phi(X.bracket(Y))
-i*C
sage: phi(X).bracket(phi(Y))
-i*C

```

```

>>> from sage.all import *
>>> L = LieAlgebra(K, {'X','Y': {'Z':1}, ('X','Z'): {'W':Integer(1)}}, names=('X'
↪ ', 'Y', 'Z', 'W')); (X, Y, Z, W) = L._first_ngens(4)
>>> M = LieAlgebra(K, {'A','B': {'C':-1}, ('A','C'): {'D':Integer(1)}}, names=(
↪ 'A', 'B', 'C', 'D')); (A, B, C, D) = M._first_ngens(4)
>>> phi = L.morphism({X:A, Y:B, Z:C, W:D}, base_map=cc)
>>> phi(X.bracket(Y))
-i*C
>>> phi(X).bracket(phi(Y))
-i*C

```

base_map()

Return the map on the base ring that is part of the defining data for this morphism. May return None if a coercion is used.

EXAMPLES:

```

sage: R.<x> = ZZ[]
sage: K.<i> = NumberField(x^2 + 1)
sage: cc = K.hom([-i])
sage: L.<X,Y,Z,W> = LieAlgebra(K, {'X','Y': {'Z':1}, ('X','Z'): {'W':1}})
sage: M.<A,B> = LieAlgebra(K, abelian=True)
sage: phi = L.morphism({X: A, Y: B}, base_map=cc)
sage: phi(X)
A
sage: phi(i*X)
-i*A
sage: phi.base_map()
Ring endomorphism of Number Field in i with defining polynomial x^2 + 1
Defn: i |--> -i

```

```

>>> from sage.all import *
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> K = NumberField(x**Integer(2) + Integer(1), names=('i',)); (i,) = K._
↪ first_ngens(1)
>>> cc = K.hom([-i])
>>> L = LieAlgebra(K, {'X','Y': {'Z':Integer(1)}, ('X','Z'): {'W'

```

(continues on next page)

(continued from previous page)

```

↪':Integer(1)}}

```

im_gens()

Return the images of the generators of the domain.

OUTPUT:

- list – a copy of the list of gens (it is safe to change this)

EXAMPLES:

```

sage: L = LieAlgebra(QQ, 'x,y,z')
sage: Lyn = L.Lyndon()
sage: H = L.Hall()
sage: f = Lyn.coerce_map_from(H)
sage: f.im_gens()
[x, y, z]

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, 'x,y,z')
>>> Lyn = L.Lyndon()
>>> H = L.Hall()
>>> f = Lyn.coerce_map_from(H)
>>> f.im_gens()
[x, y, z]

```

class sage.algebras.lie_algebras.morphism.**LieAlgebraHomset** (*X, Y, category=None, base=None, check=True*)

Bases: `Homset`

Homset between two Lie algebras.

Todo

This is a very minimal implementation which does not have coercions of the morphisms.

zero()

Return the zero morphism.

EXAMPLES:

```

sage: L = LieAlgebra(QQ, 'x,y,z')
sage: Lyn = L.Lyndon()
sage: H = L.Hall()
sage: HS = Hom(Lyn, H)
sage: HS.zero()
Generic morphism:
  From: Free Lie algebra generated by (x, y, z) over Rational Field in the
  ↪Lyndon basis
  To:   Free Lie algebra generated by (x, y, z) over Rational Field in the
  ↪Hall basis

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, 'x,y,z')
>>> Lyn = L.Lyndon()
>>> H = L.Hall()
>>> HS = Hom(Lyn, H)
>>> HS.zero()
Generic morphism:
  From: Free Lie algebra generated by (x, y, z) over Rational Field in the
  ↪Lyndon basis
  To:   Free Lie algebra generated by (x, y, z) over Rational Field in the
  ↪Hall basis

```

```

class sage.algebras.lie_algebras.morphism.LieAlgebraMorphism_from_generators (on_genera-
                                                                 tors,
                                                                 do-
                                                                 main=None,
                                                                 codomain=None,
                                                                 check=True,
                                                                 base_map=None,
                                                                 cate-
                                                                 gory=None)

```

Bases: *LieAlgebraHomomorphism_im_gens*

A morphism between two Lie algebras defined by images of a generating set as a Lie algebra.

This is the Lie algebra morphism $\phi: L \rightarrow K$ defined on the chosen basis of L to that of K by using the image of some generating set (as a Lie algebra) of L .

INPUT:

- `on_generators` – dictionary $\{X: Y\}$ of the images Y in `codomain` of elements X of `domain`
- `codomain` – a Lie algebra (optional); this is inferred from the values of `on_generators` if not given
- `base_map` – a homomorphism to apply to the coefficients. It should be a map from the base ring of the domain to the base ring of the codomain. Note that if `base_map` is nontrivial then the result will not be a morphism in the category of Lie algebras over the base ring.
- `check` – boolean (default: `True`); if `False` the values on the Lie brackets implied by `on_generators` will not be checked for contradictory values

EXAMPLES:

A reflection of one horizontal vector in the Heisenberg algebra:

```
sage: L.<X,Y,Z> = LieAlgebra(QQ, {'X','Y': {'Z':1}})
sage: phi = L.morphism({X:-X, Y:Y}); phi
Lie algebra endomorphism of Lie algebra on 3 generators (X, Y, Z) over Rational_
↪Field
Defn: X |--> -X
      Y |--> Y
      Z |--> -Z
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, {'X','Y': {'Z':Integer(1)}}, names=('X', 'Y', 'Z')); (X,
↪ Y, Z,) = L._first_ngens(3)
>>> phi = L.morphism({X:-X, Y:Y}); phi
Lie algebra endomorphism of Lie algebra on 3 generators (X, Y, Z) over Rational_
↪Field
Defn: X |--> -X
      Y |--> Y
      Z |--> -Z
```

There is no Lie algebra morphism that reflects one horizontal vector, but not the vertical one:

```
sage: L.morphism({X:-X, Y:Y, Z:Z})
Traceback (most recent call last):
...
ValueError: this does not define a Lie algebra morphism;
contradictory values for brackets of length 2
```

```
>>> from sage.all import *
>>> L.morphism({X:-X, Y:Y, Z:Z})
Traceback (most recent call last):
...
ValueError: this does not define a Lie algebra morphism;
contradictory values for brackets of length 2
```

Checking for mistakes can be disabled, which can produce invalid results:

```
sage: phi = L.morphism({X:-X, Y:Y, Z:Z}, check=False); phi
Lie algebra endomorphism of Lie algebra on 3 generators (X, Y, Z) over Rational_
↪Field
Defn: X |--> -X
      Y |--> Y
      Z |--> Z
sage: L[phi(X), phi(Y)] == phi(L[X,Y])
False
```

```
>>> from sage.all import *
>>> phi = L.morphism({X:-X, Y:Y, Z:Z}, check=False); phi
Lie algebra endomorphism of Lie algebra on 3 generators (X, Y, Z) over Rational_
↪Field
Defn: X |--> -X
      Y |--> Y
      Z |--> Z
>>> L[phi(X), phi(Y)] == phi(L[X,Y])
False
```


The set of keys must generate the Lie algebra:

```
sage: L.morphism({X: X})
Traceback (most recent call last):
...
ValueError: [X] is not a generating set of Lie algebra on 3 generators
(X, Y, Z) over Rational Field
```

```
>>> from sage.all import *
>>> L.morphism({X: X})
Traceback (most recent call last):
...
ValueError: [X] is not a generating set of Lie algebra on 3 generators
(X, Y, Z) over Rational Field
```

Over non-fields, generating subsets are more restricted:

```
sage: L.<X,Y,Z> = LieAlgebra(ZZ, {'X','Y': {'Z':2}})
sage: L.morphism({X: X, Y: Y})
Traceback (most recent call last):
...
ValueError: [X, Y] is not a generating set of Lie algebra on 3
generators (X, Y, Z) over Integer Ring
```

```
>>> from sage.all import *
>>> L = LieAlgebra(ZZ, {'X','Y': {'Z':Integer(2)}}, names=('X', 'Y', 'Z')); (X,
↪ Y, Z,) = L._first_ngens(3)
>>> L.morphism({X: X, Y: Y})
Traceback (most recent call last):
...
ValueError: [X, Y] is not a generating set of Lie algebra on 3
generators (X, Y, Z) over Integer Ring
```

The generators do not have to correspond to the defined generating set of the domain:

```
sage: L.<X,Y,Z,W> = LieAlgebra(QQ, {'X','Y': {'Z':1}, ('X','Z'): {'W':1}})
sage: K.<A,B,C> = LieAlgebra(QQ, {'A','B': {'C':2}})
sage: phi = L.morphism({X+2*Y: A, X-Y: B}); phi
Lie algebra morphism:
  From: Lie algebra on 4 generators (X, Y, Z, W) over Rational Field
  To:   Lie algebra on 3 generators (A, B, C) over Rational Field
  Defn: X |--> 1/3*A + 2/3*B
        Y |--> 1/3*A - 1/3*B
        Z |--> -2/3*C
        W |--> 0
sage: phi(X+2*Y)
A
sage: phi(X)
1/3*A + 2/3*B
sage: phi(W)
0
sage: phi(Z)
-2/3*C
```

(continues on next page)

(continued from previous page)

```
sage: all(K[phi(p), phi(q)] == phi(L[p,q])
....:     for p in L.basis() for q in L.basis())
True
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, {'X','Y': {'Z':Integer(1)}, ('X','Z'): {'W':Integer(1)}},
↳ names=('X', 'Y', 'Z', 'W',)); (X, Y, Z, W,) = L._first_ngens(4)
>>> K = LieAlgebra(QQ, {'A','B': {'C':Integer(2)}}, names=('A', 'B', 'C',)); (A,
↳ B, C,) = K._first_ngens(3)
>>> phi = L.morphism({X+Integer(2)*Y: A, X-Y: B}); phi
Lie algebra morphism:
  From: Lie algebra on 4 generators (X, Y, Z, W) over Rational Field
  To:   Lie algebra on 3 generators (A, B, C) over Rational Field
  Defn: X |--> 1/3*A + 2/3*B
        Y |--> 1/3*A - 1/3*B
        Z |--> -2/3*C
        W |--> 0
>>> phi(X+Integer(2)*Y)
A
>>> phi(X)
1/3*A + 2/3*B
>>> phi(W)
0
>>> phi(Z)
-2/3*C
>>> all(K[phi(p), phi(q)] == phi(L[p,q])
...     for p in L.basis() for q in L.basis())
True
```

A quotient type Lie algebra morphism:

```
sage: K.<A,B> = LieAlgebra(SR, abelian=True) #_
↳needs sage.symbolic
sage: L.morphism({X: A, Y: B}) #_
↳needs sage.symbolic
Lie algebra morphism:
  From: Lie algebra on 4 generators (X, Y, Z, W) over Rational Field
  To:   Abelian Lie algebra on 2 generators (A, B) over Symbolic Ring
  Defn: X |--> A
        Y |--> B
        Z |--> 0
        W |--> 0
```

```
>>> from sage.all import *
>>> K = LieAlgebra(SR, abelian=True, names=('A', 'B',)); (A, B,) = K._first_
↳ngens(2) # needs sage.symbolic
>>> L.morphism({X: A, Y: B}) #_
↳needs sage.symbolic
Lie algebra morphism:
  From: Lie algebra on 4 generators (X, Y, Z, W) over Rational Field
  To:   Abelian Lie algebra on 2 generators (A, B) over Symbolic Ring
  Defn: X |--> A
```

(continues on next page)

(continued from previous page)

```

Y |--> B
Z |--> 0
W |--> 0

```

9.2.14 Nilpotent Lie algebras

AUTHORS:

- Eero Hakavuori (2018-08-16): initial version

```

class sage.algebras.lie_algebras.nilpotent_lie_algebra.FreeNilpotentLieAlgebra(R, r, s,
                                                                              names,
                                                                              naming,
                                                                              category,
                                                                              **kws)

```

Bases: *NilpotentLieAlgebra_dense*

Return the free nilpotent Lie algebra of step s with r generators.

The free nilpotent Lie algebra L of step s with r generators is the quotient of the free Lie algebra on r generators by the $(s+1)$ -th term of the lower central series. That is, the only relations in the Lie algebra L are anticommutativity, the Jacobi identity, and the vanishing of all brackets of length more than s .

INPUT:

- R – the base ring
- r – integer; the number of generators
- s – integer; the nilpotency step of the algebra
- *names* – (optional) string or list of strings used to name the basis elements; if *names* is a string, then names for the basis will be autogenerated as determined by the *naming* parameter
- *naming* – (optional) a string; the naming scheme to use for the basis; valid values are:
 - 'index' – (default for $r < 10$) the basis elements are names_w , where w are Lyndon words indexing the basis
 - 'linear' – (default for $r \geq 10$) the basis is indexed $\text{names}_1, \dots, \text{names}_n$ in the ordering of the Lyndon basis

Note

The 'index' naming scheme is not supported if $r \geq 10$ since it leads to ambiguous names.

EXAMPLES:

We compute the free step 4 Lie algebra on 2 generators and verify the only non-trivial relation $[[X_1, [X_1, X_2]], X_2] = [X_1, [[X_1, X_2], X_2]]$:

```

sage: L = LieAlgebra(QQ, 2, step=4)
sage: L.basis().list()
[X_1, X_2, X_12, X_112, X_122, X_1112, X_1122, X_1222]
sage: X_1, X_2 = L.basis().list()[:2]
sage: L[[X_1, [X_1, X_2]], X_2]
X_1122

```

(continues on next page)

(continued from previous page)

```
sage: L[[X_1, [X_1, X_2]], X_2] == L[X_1, [[X_1, X_2], X_2]]
True
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(2), step=Integer(4))
>>> L.basis().list()
[X_1, X_2, X_12, X_112, X_122, X_1112, X_1122, X_1222]
>>> X_1, X_2 = L.basis().list()[Integer(2)]
>>> L[[X_1, [X_1, X_2]], X_2]
X_1122
>>> L[[X_1, [X_1, X_2]], X_2] == L[X_1, [[X_1, X_2], X_2]]
True
```

The linear naming scheme on the same Lie algebra:

```
sage: K = LieAlgebra(QQ, 2, step=4, names='Y', naming='linear')
sage: K.basis().list()
[Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, Y_7, Y_8]
sage: K.inject_variables()
Defining Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, Y_7, Y_8
sage: Y_2.bracket(Y_3)
-Y_5
sage: Y_5.bracket(Y_1)
-Y_7
sage: Y_3.bracket(Y_4)
0
```

```
>>> from sage.all import *
>>> K = LieAlgebra(QQ, Integer(2), step=Integer(4), names='Y', naming='linear')
>>> K.basis().list()
[Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, Y_7, Y_8]
>>> K.inject_variables()
Defining Y_1, Y_2, Y_3, Y_4, Y_5, Y_6, Y_7, Y_8
>>> Y_2.bracket(Y_3)
-Y_5
>>> Y_5.bracket(Y_1)
-Y_7
>>> Y_3.bracket(Y_4)
0
```

A fully custom naming scheme on the Heisenberg algebra:

```
sage: L = LieAlgebra(ZZ, 2, step=2, names=('X', 'Y', 'Z'))
sage: a, b, c = L.basis()
sage: L.basis().list()
[X, Y, Z]
sage: a.bracket(b)
Z
```

```
>>> from sage.all import *
>>> L = LieAlgebra(ZZ, Integer(2), step=Integer(2), names=('X', 'Y', 'Z'))
>>> a, b, c = L.basis()
```

(continues on next page)

(continued from previous page)

```
>>> L.basis().list()
[X, Y, Z]
>>> a.bracket(b)
Z
```

An equivalent way to define custom names for the basis elements and bind them as local variables simultaneously:

```
sage: L.<X,Y,Z> = LieAlgebra(ZZ, 2, step=2)
sage: L.basis().list()
[X, Y, Z]
sage: X.bracket(Y)
Z
```

```
>>> from sage.all import *
>>> L = LieAlgebra(ZZ, Integer(2), step=Integer(2), names=('X', 'Y', 'Z')); (X, Y, Z) = L._first_ngens(3)
>>> L.basis().list()
[X, Y, Z]
>>> X.bracket(Y)
Z
```

A free nilpotent Lie algebra is a stratified nilpotent Lie algebra:

```
sage: L = LieAlgebra(QQ, 3, step=3)
sage: L.category()
Category of finite dimensional stratified Lie algebras with basis over Rational_
↪Field
sage: L in LieAlgebras(QQ).Nilpotent()
True
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(3), step=Integer(3))
>>> L.category()
Category of finite dimensional stratified Lie algebras with basis over Rational_
↪Field
>>> L in LieAlgebras(QQ).Nilpotent()
True
```

Being graded means that each basis element has a degree:

```
sage: L in LieAlgebras(QQ).Graded()
True
sage: L.homogeneous_component_basis(1).list()
[X_1, X_2, X_3]
sage: L.homogeneous_component_basis(2).list()
[X_12, X_13, X_23]
sage: L.homogeneous_component_basis(3).list()
[X_112, X_113, X_122, X_123, X_132, X_133, X_223, X_233]
```

```
>>> from sage.all import *
>>> L in LieAlgebras(QQ).Graded()
True
```

(continues on next page)

(continued from previous page)

```
>>> L.homogeneous_component_basis(Integer(1)).list()
[X_1, X_2, X_3]
>>> L.homogeneous_component_basis(Integer(2)).list()
[X_12, X_13, X_23]
>>> L.homogeneous_component_basis(Integer(3)).list()
[X_112, X_113, X_122, X_123, X_132, X_133, X_223, X_233]
```

```
class sage.algebras.lie_algebras.nilpotent_lie_algebra.NilpotentLieAlgebra_dense(R,
                                                                              s_coeff,
                                                                              names,
                                                                              in-
                                                                              dex_set,
                                                                              step=None,
                                                                              **kwds)
```

Bases: *LieAlgebraWithStructureCoefficients*

A nilpotent Lie algebra L over a base ring.

INPUT:

- R – the base ring
- s_coeff – dictionary of structural coefficients
- $names$ – (default: `None`) list of strings to use as names of basis elements; if `None`, the names will be inferred from the structural coefficients
- $index\_set$ – (default: `None`) list of hashable and comparable elements to use for indexing
- $step$ – integer (optional); the nilpotency step of the Lie algebra if known; otherwise it will be computed when needed
- $category$ – (optional) a subcategory of finite dimensional nilpotent Lie algebras with basis

EXAMPLES:

The input to a *NilpotentLieAlgebra_dense* should be of the same form as to a *LieAlgebraWithStructureCoefficients*:

```
sage: L.<X,Y,Z,W> = LieAlgebra(QQ, {'X','Y': {'Z': 1}}, nilpotent=True)
sage: L
Nilpotent Lie algebra on 4 generators (X, Y, Z, W) over Rational Field
sage: L[X, Y]
Z
sage: L[X, W]
0
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, {'X','Y': {'Z': Integer(1)}}, nilpotent=True, names=('X',
↳ 'Y', 'Z', 'W',)); (X, Y, Z, W,) = L._first_ngens(4)
>>> L
Nilpotent Lie algebra on 4 generators (X, Y, Z, W) over Rational Field
>>> L[X, Y]
Z
>>> L[X, W]
0
```

If the parameter `names` is omitted, then the terms appearing in the structural coefficients are used as names:

```
sage: L = LieAlgebra(QQ, {'X','Y': {'Z': 1}}, nilpotent=True); L
Nilpotent Lie algebra on 3 generators (X, Y, Z) over Rational Field
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, {'X','Y': {'Z': Integer(1)}}, nilpotent=True); L
Nilpotent Lie algebra on 3 generators (X, Y, Z) over Rational Field
```

9.2.15 Onsager Algebra

AUTHORS:

- Travis Scrimshaw (2017-07): Initial version

class `sage.algebras.lie_algebras.onsager.OnsagerAlgebra` (*R*)

Bases: `LieAlgebraWithGenerators`, `IndexedGenerators`

The Onsager (Lie) algebra.

The Onsager (Lie) algebra $\mathcal{O}$ is a Lie algebra with generators A_0, A_1 that satisfy

$$[A_0, [A_0, [A_0, A_1]]] = -4[A_0, A_1], \quad [A_1, [A_1, [A_1, A_0]]] = -4[A_1, A_0].$$

Note

We are using a rescaled version of the usual defining generators.

There exist a basis $\{A_m, G_n \mid m \in \mathbf{Z}, n \in \mathbf{Z}_{>0}\}$ for $\mathcal{O}$ with structure coefficients

$$[A_m, A_{m'}] = G_{m-m'}, \quad [G_n, G_{n'}] = 0, \quad [G_n, A_m] = 2A_{m-n} - 2A_{m+n},$$

where $m > m'$.

The Onsager algebra is isomorphic to the subalgebra of the affine Lie algebra $\widehat{\mathfrak{sl}}_2 = \mathfrak{sl}_2 \otimes \mathbb{C}[t, t^{-1}] \oplus \mathbb{C}K \oplus \mathbb{C}d$ that is invariant under the Chevalley involution. In particular, we have

$$A_i \mapsto f \otimes t^i - e \otimes t^{-i}, \quad G_i \mapsto h \otimes t^{-i} - h \otimes t^i.$$

where e, f, h are the Chevalley generators of $\mathfrak{sl}_2$.

EXAMPLES:

We construct the Onsager algebra and do some basic computations:

```
sage: O = lie_algebras.OnsagerAlgebra(QQ)
sage: O.inject_variables()
Defining A0, A1
```

```
>>> from sage.all import *
>>> O = lie_algebras.OnsagerAlgebra(QQ)
>>> O.inject_variables()
Defining A0, A1
```

We verify the defining relations:

```
sage: O([A0, [A0, [A0, A1]]]) == -4 * O([A0, A1])
True
sage: O([A1, [A1, [A1, A0]]]) == -4 * O([A1, A0])
True
```

```
>>> from sage.all import *
>>> O([A0, [A0, [A0, A1]]]) == -Integer(4) * O([A0, A1])
True
>>> O([A1, [A1, [A1, A0]]]) == -Integer(4) * O([A1, A0])
True
```

We check the embedding into $\widehat{\mathfrak{sl}}_2$:

```
sage: L = LieAlgebra(QQ, cartan_type=['A',1,1])
sage: B = L.basis()
sage: al = RootSystem(['A',1]).root_lattice().simple_root(1)
sage: ac = al.associated_coroot()
sage: def emb_A(i): return B[-al,i] - B[al,-i]
sage: def emb_G(i): return B[ac,i] - B[ac,-i]
sage: a0 = emb_A(0)
sage: a1 = emb_A(1)
sage: L([a0, [a0, [a0, a1]]]) == -4 * L([a0, a1])
True
sage: L([a1, [a1, [a1, a0]]]) == -4 * L([a1, a0])
True

sage: all(emb_G(n).bracket(emb_A(m)) == 2*emb_A(m-n) - 2*emb_A(m+n)
.....:      for m in range(-10, 10) for n in range(1,10))
True
sage: all(emb_A(m).bracket(emb_A(mp)) == emb_G(m-mp)
.....:      for m in range(-10,10) for mp in range(m-10, m))
True
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, cartan_type=['A',Integer(1),Integer(1)])
>>> B = L.basis()
>>> al = RootSystem(['A',Integer(1)]).root_lattice().simple_root(Integer(1))
>>> ac = al.associated_coroot()
>>> def emb_A(i): return B[-al,i] - B[al,-i]
>>> def emb_G(i): return B[ac,i] - B[ac,-i]
>>> a0 = emb_A(Integer(0))
>>> a1 = emb_A(Integer(1))
>>> L([a0, [a0, [a0, a1]]]) == -Integer(4) * L([a0, a1])
True
>>> L([a1, [a1, [a1, a0]]]) == -Integer(4) * L([a1, a0])
True

>>> all(emb_G(n).bracket(emb_A(m)) == Integer(2)*emb_A(m-n) - Integer(2)*emb_
↪A(m+n)
...     for m in range(-Integer(10), Integer(10)) for n in range(Integer(1),
↪Integer(10)))
True
```

(continues on next page)

(continued from previous page)

```
>>> all(emb_A(m).bracket(emb_A(mp)) == emb_G(m-mp)
...      for m in range(-Integer(10), Integer(10)) for mp in range(m-Integer(10), m+
...      Integer(10)))
True
```

REFERENCES:

- [Onsager1944]
- [DG1982]

Elementalias of *LieAlgebraElement***alternating_central_extension()**Return the alternating central extension of *self*.

EXAMPLES:

```
sage: O = lie_algebras.OnsagerAlgebra(QQ)
sage: ACE = O.alternating_central_extension()
sage: ACE
Alternating central extension of the Onsager algebra over Rational Field
```

```
>>> from sage.all import *
>>> O = lie_algebras.OnsagerAlgebra(QQ)
>>> ACE = O.alternating_central_extension()
>>> ACE
Alternating central extension of the Onsager algebra over Rational Field
```

basis()Return the basis of *self*.

EXAMPLES:

```
sage: O = lie_algebras.OnsagerAlgebra(QQ)
sage: O.basis()
Lazy family (Onsager monomial(i))_{i in
Disjoint union of Family (Integer Ring, Positive integers)}
```

```
>>> from sage.all import *
>>> O = lie_algebras.OnsagerAlgebra(QQ)
>>> O.basis()
Lazy family (Onsager monomial(i))_{i in
Disjoint union of Family (Integer Ring, Positive integers)}
```

bracket_on_basis(x, y)Return the bracket of basis elements indexed by *x* and *y* where *x* < *y*.

EXAMPLES:

```
sage: O = lie_algebras.OnsagerAlgebra(QQ)
sage: O.bracket_on_basis((1,3), (1,9)) # [G, G]
0
sage: O.bracket_on_basis((0,8), (1,13)) # [A, G]
```

(continues on next page)

(continued from previous page)

```
-2*A[-5] + 2*A[21]
sage: O.bracket_on_basis((0,-9), (0, 7)) # [A, A]
-G[16]
```

```
>>> from sage.all import *
>>> O = lie_algebras.OnsagerAlgebra(QQ)
>>> O.bracket_on_basis((Integer(1),Integer(3)), (Integer(1),Integer(9))) #_
↪ [G, G]
0
>>> O.bracket_on_basis((Integer(0),Integer(8)), (Integer(1),Integer(13))) #_
↪ [A, G]
-2*A[-5] + 2*A[21]
>>> O.bracket_on_basis((Integer(0),-Integer(9)), (Integer(0), Integer(7))) #_
↪ [A, A]
-G[16]
```

lie_algebra_generators()

Return the generators of `self` as a Lie algebra.

EXAMPLES:

```
sage: O = lie_algebras.OnsagerAlgebra(QQ)
sage: O.lie_algebra_generators()
Finite family {'A0': A[0], 'A1': A[1]}
```

```
>>> from sage.all import *
>>> O = lie_algebras.OnsagerAlgebra(QQ)
>>> O.lie_algebra_generators()
Finite family {'A0': A[0], 'A1': A[1]}
```

quantum_group(*q=None, c=None*)

Return the quantum group of `self`.

The corresponding quantum group is the *QuantumOnsagerAlgebra*. The parameter *c* must be such that $c(1) = 1$

INPUT:

- *q* – (optional) the quantum parameter; the default is $q \in R(q)$, where *R* is the base ring of `self`
- *c* – (optional) the parameter *c*; the default is *q*

EXAMPLES:

```
sage: O = lie_algebras.OnsagerAlgebra(QQ)
sage: Q = O.quantum_group()
sage: Q
q-Onsager algebra with c=q over Fraction Field of
Univariate Polynomial Ring in q over Rational Field
```

```
>>> from sage.all import *
>>> O = lie_algebras.OnsagerAlgebra(QQ)
>>> Q = O.quantum_group()
>>> Q
```

(continues on next page)

(continued from previous page)

```
q-Onsager algebra with c=q over Fraction Field of
Univariate Polynomial Ring in q over Rational Field
```

some_elements()

Return some elements of `self`.

EXAMPLES:

```
sage: O = lie_algebras.OnsagerAlgebra(QQ)
sage: O.some_elements()
[A[0], A[2], A[-1], G[4], -2*A[-3] + A[2] + 3*G[2]]
```

```
>>> from sage.all import *
>>> O = lie_algebras.OnsagerAlgebra(QQ)
>>> O.some_elements()
[A[0], A[2], A[-1], G[4], -2*A[-3] + A[2] + 3*G[2]]
```

class `sage.algebras.lie_algebras.onsager.OnsagerAlgebraACE(R)`

Bases: *InfinitelyGeneratedLieAlgebra*, *IndexedGenerators*

The alternating central extension of the Onsager algebra.

The *alternating central extension* of the *Onsager algebra* is the Lie algebra with basis elements $\{\mathcal{A}_k, \mathcal{B}_k\}_{k \in \mathbb{Z}}$ that satisfy the relations

$$\begin{aligned} [\mathcal{A}_k, \mathcal{A}_m] &= \mathcal{B}_{k-m} - \mathcal{B}_{m-k}, \\ [\mathcal{A}_k, \mathcal{B}_m] &= \mathcal{A}_{k+m} - \mathcal{A}_{k-m}, \\ [\mathcal{B}_k, \mathcal{B}_m] &= 0. \end{aligned}$$

This has a natural injection from the Onsager algebra by the map ι defined by

$$\iota(\mathcal{A}_k) = \mathcal{A}_k, \quad \iota(\mathcal{B}_k) = \mathcal{B}_k - \mathcal{B}_{-k}.$$

Note that the map ι differs slightly from Lemma 4.18 in [Ter2021b] due to our choice of basis of the Onsager algebra.

Warning

We have added an extra basis vector $\mathcal{B}_0$, which would be 0 in the definition given in [Ter2021b].

EXAMPLES:

We begin by constructing the ACE and doing some sample computations:

```
sage: O = lie_algebras.OnsagerAlgebra(QQ)
sage: ACE = O.alternating_central_extension()
sage: ACE
Alternating central extension of the Onsager algebra over Rational Field

sage: B = ACE.basis()
sage: A1, A2, Am2 = B[0,1], B[0,2], B[0,-2]
sage: B1, B2, Bm2 = B[1,1], B[1,2], B[1,-2]
sage: A1.bracket(Am2)
```

(continues on next page)

(continued from previous page)

```

-B[-3] + B[3]
sage: A1.bracket(A2)
B[-1] - B[1]
sage: A1.bracket(B2)
-A[-1] + A[3]
sage: A1.bracket(Bm2)
A[-1] - A[3]
sage: B2.bracket(B1)
0
sage: Bm2.bracket(B2)
0
sage: (A2 + Am2).bracket(B1 + A2 + B2 + Bm2)
-A[-3] + A[-1] - A[1] + A[3] + B[-4] - B[4]

```

```

>>> from sage.all import *
>>> O = lie_algebras.OnsagerAlgebra(QQ)
>>> ACE = O.alternating_central_extension()
>>> ACE
Alternating central extension of the Onsager algebra over Rational Field

>>> B = ACE.basis()
>>> A1, A2, Am2 = B[Integer(0), Integer(1)], B[Integer(0), Integer(2)],
↳ B[Integer(0), -Integer(2)]
>>> B1, B2, Bm2 = B[Integer(1), Integer(1)], B[Integer(1), Integer(2)],
↳ B[Integer(1), -Integer(2)]
>>> A1.bracket(Am2)
-B[-3] + B[3]
>>> A1.bracket(A2)
B[-1] - B[1]
>>> A1.bracket(B2)
-A[-1] + A[3]
>>> A1.bracket(Bm2)
A[-1] - A[3]
>>> B2.bracket(B1)
0
>>> Bm2.bracket(B2)
0
>>> (A2 + Am2).bracket(B1 + A2 + B2 + Bm2)
-A[-3] + A[-1] - A[1] + A[3] + B[-4] - B[4]

```

The natural inclusion map ι is implemented as a coercion map:

```

sage: iota = ACE.coerce_map_from(O)
sage: b = O.basis()
sage: am1, a2, b4 = b[0, -1], b[0, 2], b[1, 4]
sage: iota(am1.bracket(a2)) == iota(am1).bracket(iota(a2))
True
sage: iota(am1.bracket(b4)) == iota(am1).bracket(iota(b4))
True
sage: iota(b4.bracket(a2)) == iota(b4).bracket(iota(a2))
True

```

(continues on next page)

(continued from previous page)

```

sage: am1 + B2
A[-1] + B[2]
sage: am1.bracket(B2)
-A[-3] + A[1]
sage: Bm2.bracket(a2)
-A[0] + A[4]

```

```

>>> from sage.all import *
>>> iota = ACE.coerce_map_from(0)
>>> b = O.basis()
>>> am1, a2, b4 = b[Integer(0), -Integer(1)], b[Integer(0), Integer(2)], ↵
↵ b[Integer(1), Integer(4)]
>>> iota(am1.bracket(a2)) == iota(am1).bracket(iota(a2))
True
>>> iota(am1.bracket(b4)) == iota(am1).bracket(iota(b4))
True
>>> iota(b4.bracket(a2)) == iota(b4).bracket(iota(a2))
True

>>> am1 + B2
A[-1] + B[2]
>>> am1.bracket(B2)
-A[-3] + A[1]
>>> Bm2.bracket(a2)
-A[0] + A[4]

```

We have the projection map ρ from Lemma 4.19 in [Ter2021b]:

$$\rho(\mathcal{A}_k) = A_k, \quad \rho(\mathcal{B}_k) = \operatorname{sgn}(k)B_{|k|}.$$

The kernel of ρ is the center $\mathcal{Z}$, which has a basis $\{B_k + B_{-k}\}_{k \in \mathbb{Z}}$:

```

sage: rho = ACE.projection()
sage: rho(A1)
A[1]
sage: rho(Am2)
A[-2]
sage: rho(B1)
1/2*G[1]
sage: rho(Bm2)
-1/2*G[2]
sage: all(rho(B[1,k] + B[1,-k]) == 0 for k in range(-6,6))
True
sage: all(B[0,m].bracket(B[1,k] + B[1,-k]) == 0
.....:      for k in range(-4,4) for m in range(-4,4))
True

```

```

>>> from sage.all import *
>>> rho = ACE.projection()
>>> rho(A1)
A[1]
>>> rho(Am2)

```

(continues on next page)

(continued from previous page)

```

A[-2]
>>> rho(B1)
1/2*G[1]
>>> rho(Bm2)
-1/2*G[2]
>>> all(rho(B[Integer(1),k] + B[Integer(1),-k]) == Integer(0) for k in range(-
↳ Integer(6), Integer(6)))
True
>>> all(B[Integer(0),m].bracket(B[Integer(1),k] + B[Integer(1),-k]) == Integer(0)
...     for k in range(-Integer(4), Integer(4)) for m in range(-Integer(4),
↳ Integer(4)))
True

```

Elementalias of *LieAlgebraElement***basis()**

Return the basis of self.

EXAMPLES:

```

sage: O = lie_algebras.OnsagerAlgebra(QQ).alternating_central_extension()
sage: O.basis()
Lazy family (Onsager ACE monomial(i))_{i in
Disjoint union of Family (Integer Ring, Integer Ring)}

```

```

>>> from sage.all import *
>>> O = lie_algebras.OnsagerAlgebra(QQ).alternating_central_extension()
>>> O.basis()
Lazy family (Onsager ACE monomial(i))_{i in
Disjoint union of Family (Integer Ring, Integer Ring)}

```

bracket_on_basis(x, y)Return the bracket of basis elements indexed by x and y where $x < y$.**EXAMPLES:**

```

sage: O = lie_algebras.OnsagerAlgebra(QQ).alternating_central_extension()
sage: O.bracket_on_basis((1,3), (1,9)) # [B, B]
0
sage: O.bracket_on_basis((0,8), (1,13)) # [A, B]
-A[-5] + A[21]
sage: O.bracket_on_basis((0,-9), (0, 7)) # [A, A]
B[-16] - B[16]

```

```

>>> from sage.all import *
>>> O = lie_algebras.OnsagerAlgebra(QQ).alternating_central_extension()
>>> O.bracket_on_basis((Integer(1),Integer(3)), (Integer(1),Integer(9))) #↳
↳ [B, B]
0
>>> O.bracket_on_basis((Integer(0),Integer(8)), (Integer(1),Integer(13))) #↳
↳ [A, B]
-A[-5] + A[21]

```

(continues on next page)

(continued from previous page)

```
>>> O.bracket_on_basis((Integer(0), -Integer(9)), (Integer(0), Integer(7))) #_
↪ [A, A]
B[-16] - B[16]
```

lie_algebra_generators()

Return the generators of `self` as a Lie algebra.

EXAMPLES:

```
sage: O = lie_algebras.OnsagerAlgebra(QQ).alternating_central_extension()
sage: O.lie_algebra_generators()
Lazy family (Onsager ACE monomial(i))_{i in
Disjoint union of Family (Integer Ring, Integer Ring)}
```

```
>>> from sage.all import *
>>> O = lie_algebras.OnsagerAlgebra(QQ).alternating_central_extension()
>>> O.lie_algebra_generators()
Lazy family (Onsager ACE monomial(i))_{i in
Disjoint union of Family (Integer Ring, Integer Ring)}
```

projection()

Return the projection map ρ from Lemma 4.19 in [Ter2021b] to the Onsager algebra.

EXAMPLES:

```
sage: O = lie_algebras.OnsagerAlgebra(QQ)
sage: ACE = O.alternating_central_extension()
sage: rho = ACE.projection()
sage: B = ACE.basis()
sage: A1, A2, Am2 = B[0,1], B[0,2], B[0,-2]
sage: B1, B2, Bm2 = B[1,1], B[1,2], B[1,-2]

sage: rho(A1)
A[1]
sage: rho(Am2)
A[-2]
sage: rho(B1)
1/2*G[1]
sage: rho(B2)
1/2*G[2]
sage: rho(Bm2)
-1/2*G[2]

sage: rho(A1.bracket(A2))
-G[1]
sage: rho(A1).bracket(rho(A2))
-G[1]
sage: rho(B1.bracket(Am2))
A[-3] - A[-1]
sage: rho(B1).bracket(rho(Am2))
A[-3] - A[-1]
```

```

>>> from sage.all import *
>>> O = lie_algebras.OnsagerAlgebra(QQ)
>>> ACE = O.alternating_central_extension()
>>> rho = ACE.projection()
>>> B = ACE.basis()
>>> A1, A2, Am2 = B[Integer(0), Integer(1)], B[Integer(0), Integer(2)],
↳ B[Integer(0), -Integer(2)]
>>> B1, B2, Bm2 = B[Integer(1), Integer(1)], B[Integer(1), Integer(2)],
↳ B[Integer(1), -Integer(2)]

>>> rho(A1)
A[1]
>>> rho(Am2)
A[-2]
>>> rho(B1)
1/2*G[1]
>>> rho(B2)
1/2*G[2]
>>> rho(Bm2)
-1/2*G[2]

>>> rho(A1.bracket(A2))
-G[1]
>>> rho(A1).bracket(rho(A2))
-G[1]
>>> rho(B1.bracket(Am2))
A[-3] - A[-1]
>>> rho(B1).bracket(rho(Am2))
A[-3] - A[-1]

```

some_elements()

Return some elements of self.

EXAMPLES:

```

sage: O = lie_algebras.OnsagerAlgebra(QQ).alternating_central_extension()
sage: O.some_elements()
[A[0], A[2], A[-1], B[4], B[-3], -2*A[-3] + A[2] + B[-1] + 3*B[2]]

```

```

>>> from sage.all import *
>>> O = lie_algebras.OnsagerAlgebra(QQ).alternating_central_extension()
>>> O.some_elements()
[A[0], A[2], A[-1], B[4], B[-3], -2*A[-3] + A[2] + B[-1] + 3*B[2]]

```

class sage.algebras.lie_algebras.onsager.**QuantumOnsagerAlgebra**(g, q, c)

Bases: `CombinatorialFreeModule`

The quantum Onsager algebra.

The *quantum Onsager algebra*, or *q-Onsager algebra*, is a quantum group analog of the Onsager algebra. It is the left (or right) coideal subalgebra of the quantum group $U_q(\widehat{\mathfrak{sl}}_2)$ and is the simplest example of a quantum symmetric pair coideal subalgebra of affine type.

The *q-Onsager algebra* depends on a parameter c such that $c(1) = 1$. The *q-Onsager algebra* with parameter c is denoted $U_q(\mathcal{O}_R)_c$, where R is the base ring of the defining Onsager algebra.

EXAMPLES:

We create the q -Onsager algebra and its generators:

```
sage: O = lie_algebras.OnsagerAlgebra(QQ)
sage: Q = O.quantum_group()
sage: G = Q.algebra_generators()
```

```
>>> from sage.all import *
>>> O = lie_algebras.OnsagerAlgebra(QQ)
>>> Q = O.quantum_group()
>>> G = Q.algebra_generators()
```

The generators are given as pairs, where $G[0, n]$ is the generator $B_{n\delta + \alpha_1}$ and $G[1, n]$ is the generator $B_{n\delta}$. We use the convention that $n\delta + \alpha_1 \equiv (-n - 1)\delta + \alpha_0$.

```
sage: G[0, 5]
B[5d+a1]
sage: G[0, -5]
B[4d+a0]
sage: G[1, 5]
B[5d]
sage: (G[0, 5] + G[0, -3]) * (G[1, 2] - G[0, 3])
B[2d+a0]*B[2d] - B[2d+a0]*B[3d+a1]
+ ((-q^4+1)/q^2)*B[1d]*B[6d+a1]
+ ((q^4-1)/q^2)*B[1d]*B[4d+a1] + B[2d]*B[5d+a1]
- B[5d+a1]*B[3d+a1] + ((q^2+1)/q^2)*B[7d+a1]
+ ((q^6+q^4-q^2-1)/q^2)*B[5d+a1] + (-q^4-q^2)*B[3d+a1]
sage: (G[0, 5] + G[0, -3] + G[1, 4]) * (G[0, 2] - G[1, 3])
-B[2d+a0]*B[3d] + B[2d+a0]*B[2d+a1]
+ ((q^4-1)/q^4)*B[1d]*B[7d+a1]
+ ((q^8-2*q^4+1)/q^4)*B[1d]*B[5d+a1]
+ (-q^4+1)*B[1d]*B[3d+a1] + ((q^4-1)/q^2)*B[2d]*B[6d+a1]
+ ((-q^4+1)/q^2)*B[2d]*B[4d+a1] - B[3d]*B[4d]
- B[3d]*B[5d+a1] + B[4d]*B[2d+a1] + B[5d+a1]*B[2d+a1]
+ ((-q^2-1)/q^4)*B[8d+a1] + ((-q^6-q^4+q^2+1)/q^4)*B[6d+a1]
+ (-q^6-q^4+q^2+1)*B[4d+a1] + (q^6+q^4)*B[2d+a1]
```

```
>>> from sage.all import *
>>> G[Integer(0), Integer(5)]
B[5d+a1]
>>> G[Integer(0), -Integer(5)]
B[4d+a0]
>>> G[Integer(1), Integer(5)]
B[5d]
>>> (G[Integer(0), Integer(5)] + G[Integer(0), -Integer(3)]) * (G[Integer(1),
↪ Integer(2)] - G[Integer(0), Integer(3)])
B[2d+a0]*B[2d] - B[2d+a0]*B[3d+a1]
+ ((-q^4+1)/q^2)*B[1d]*B[6d+a1]
+ ((q^4-1)/q^2)*B[1d]*B[4d+a1] + B[2d]*B[5d+a1]
- B[5d+a1]*B[3d+a1] + ((q^2+1)/q^2)*B[7d+a1]
+ ((q^6+q^4-q^2-1)/q^2)*B[5d+a1] + (-q^4-q^2)*B[3d+a1]
>>> (G[Integer(0), Integer(5)] + G[Integer(0), -Integer(3)] + G[Integer(1),
↪ Integer(4)]) * (G[Integer(0), Integer(2)] - G[Integer(1), Integer(3)])
```

(continues on next page)

(continued from previous page)

```

-B[2d+a0]*B[3d] + B[2d+a0]*B[2d+a1]
+ ((q^4-1)/q^4)*B[1d]*B[7d+a1]
+ ((q^8-2*q^4+1)/q^4)*B[1d]*B[5d+a1]
+ (-q^4+1)*B[1d]*B[3d+a1] + ((q^4-1)/q^2)*B[2d]*B[6d+a1]
+ ((-q^4+1)/q^2)*B[2d]*B[4d+a1] - B[3d]*B[4d]
- B[3d]*B[5d+a1] + B[4d]*B[2d+a1] + B[5d+a1]*B[2d+a1]
+ ((-q^2-1)/q^4)*B[8d+a1] + ((-q^6-q^4+q^2+1)/q^4)*B[6d+a1]
+ (-q^6-q^4+q^2+1)*B[4d+a1] + (q^6+q^4)*B[2d+a1]

```

We check the q -Dolan-Grady relations:

```

sage: def q_dolan_grady(a, b, q):
.....:     x = q*a*b - ~q*b*a
.....:     y = ~q*a*x - q*x*a
.....:     return a*y - y*a
sage: A0, A1 = G[0,-1], G[0,0]
sage: q = Q.q()
sage: q_dolan_grady(A1, A0, q) == (q^4 + 2*q^2 + 1) * (A0*A1 - A1*A0)
True
sage: q_dolan_grady(A0, A1, q) == (q^4 + 2*q^2 + 1) * (A1*A0 - A0*A1)
True

```

```

>>> from sage.all import *
>>> def q_dolan_grady(a, b, q):
...     x = q*a*b - ~q*b*a
...     y = ~q*a*x - q*x*a
...     return a*y - y*a
>>> A0, A1 = G[Integer(0),-Integer(1)], G[Integer(0),Integer(0)]
>>> q = Q.q()
>>> q_dolan_grady(A1, A0, q) == (q**Integer(4) + Integer(2)*q**Integer(2) +
↳ Integer(1)) * (A0*A1 - A1*A0)
True
>>> q_dolan_grady(A0, A1, q) == (q**Integer(4) + Integer(2)*q**Integer(2) +
↳ Integer(1)) * (A1*A0 - A0*A1)
True

```

REFERENCES:

- [BK2017]

algebra_generators()

Return the algebra generators of self.

EXAMPLES:

```

sage: O = lie_algebras.OnsagerAlgebra(QQ)
sage: Q = O.quantum_group()
sage: Q.algebra_generators()
Lazy family (generator map(i))_{i in Disjoint union of
Family (Integer Ring, Positive integers)}

```

```

>>> from sage.all import *
>>> O = lie_algebras.OnsagerAlgebra(QQ)

```

(continues on next page)

(continued from previous page)

```
>>> Q = O.quantum_group()
>>> Q.algebra_generators()
Lazy family (generator map(i))_{i in Disjoint union of
  Family (Integer Ring, Positive integers)}
```

c()Return the parameter c of `self`.

EXAMPLES:

```
sage: O = lie_algebras.OnsagerAlgebra(QQ)
sage: Q = O.quantum_group(c=-3)
sage: Q.c()
-3
```

```
>>> from sage.all import *
>>> O = lie_algebras.OnsagerAlgebra(QQ)
>>> Q = O.quantum_group(c=-Integer(3))
>>> Q.c()
-3
```

degree_on_basis(m)Return the degree of the basis element indexed by m .

EXAMPLES:

```
sage: O = lie_algebras.OnsagerAlgebra(QQ)
sage: Q = O.quantum_group()
sage: G = Q.algebra_generators()
sage: B0 = G[0,0]
sage: B1 = G[0,-1]
sage: Q.degree_on_basis(B0.leading_support())
1
sage: Q.degree_on_basis((B1^10 * B0^10).leading_support())
20
sage: ((B0 * B1)^3).maximal_degree()
6
```

```
>>> from sage.all import *
>>> O = lie_algebras.OnsagerAlgebra(QQ)
>>> Q = O.quantum_group()
>>> G = Q.algebra_generators()
>>> B0 = G[Integer(0), Integer(0)]
>>> B1 = G[Integer(0), -Integer(1)]
>>> Q.degree_on_basis(B0.leading_support())
1
>>> Q.degree_on_basis((B1**Integer(10) * B0**Integer(10)).leading_support())
20
>>> ((B0 * B1)**Integer(3)).maximal_degree()
6
```

gens()Return the algebra generators of `self`.

EXAMPLES:

```
sage: O = lie_algebras.OnsagerAlgebra(QQ)
sage: Q = O.quantum_group()
sage: Q.algebra_generators()
Lazy family (generator map(i))_{i in Disjoint union of
Family (Integer Ring, Positive integers)}
```

```
>>> from sage.all import *
>>> O = lie_algebras.OnsagerAlgebra(QQ)
>>> Q = O.quantum_group()
>>> Q.algebra_generators()
Lazy family (generator map(i))_{i in Disjoint union of
Family (Integer Ring, Positive integers)}
```

lie_algebra()

Return the underlying Lie algebra of self.

EXAMPLES:

```
sage: O = lie_algebras.OnsagerAlgebra(QQ)
sage: Q = O.quantum_group()
sage: Q.lie_algebra()
Onsager algebra over Rational Field
sage: Q.lie_algebra() is O
True
```

```
>>> from sage.all import *
>>> O = lie_algebras.OnsagerAlgebra(QQ)
>>> Q = O.quantum_group()
>>> Q.lie_algebra()
Onsager algebra over Rational Field
>>> Q.lie_algebra() is O
True
```

one_basis()

Return the basis element indexing 1.

EXAMPLES:

```
sage: O = lie_algebras.OnsagerAlgebra(QQ)
sage: Q = O.quantum_group()
sage: ob = Q.one_basis(); ob
1
sage: ob.parent()
Free abelian monoid indexed by
Disjoint union of Family (Integer Ring, Positive integers)
```

```
>>> from sage.all import *
>>> O = lie_algebras.OnsagerAlgebra(QQ)
>>> Q = O.quantum_group()
>>> ob = Q.one_basis(); ob
1
```

(continues on next page)

(continued from previous page)

```
>>> ob.parent()
Free abelian monoid indexed by
Disjoint union of Family (Integer Ring, Positive integers)
```

product_on_basis (*lhs, rhs*)

Return the product of the two basis elements *lhs* and *rhs*.

EXAMPLES:

```
sage: O = lie_algebras.OnsagerAlgebra(QQ)
sage: Q = O.quantum_group()
sage: I = Q._indices.gens()
sage: Q.product_on_basis(I[1,21]^2, I[1,31]^3)
B[21d]^2*B[31d]^3
sage: Q.product_on_basis(I[1,31]^3, I[1,21]^2)
B[21d]^2*B[31d]^3
sage: Q.product_on_basis(I[0,8], I[0,6])
B[8d+a1]*B[6d+a1]
sage: Q.product_on_basis(I[0,-8], I[0,6])
B[7d+a0]*B[6d+a1]
sage: Q.product_on_basis(I[0,-6], I[0,-8])
B[5d+a0]*B[7d+a0]
sage: Q.product_on_basis(I[0,-6], I[1,2])
B[5d+a0]*B[2d]
sage: Q.product_on_basis(I[1,6], I[0,2])
B[6d]*B[2d+a1]

sage: Q.product_on_basis(I[0,1], I[0,2])
1/q^2*B[2d+a1]*B[1d+a1] - B[1d]
sage: Q.product_on_basis(I[0,-3], I[0,-1])
1/q^2*B[a0]*B[2d+a0] + ((-q^2+1)/q^2)*B[1d+a0]^2 - B[2d]
sage: Q.product_on_basis(I[0,2], I[0,-1])
q^2*B[a0]*B[2d+a1] + ((q^4-1)/q^2)*B[1d+a1]*B[a1]
+ (-q^2+1)*B[1d] + q^2*B[3d]
sage: Q.product_on_basis(I[0,2], I[1,1])
B[1d]*B[2d+a1] + (q^2+1)*B[3d+a1] + (-q^2-1)*B[1d+a1]
sage: Q.product_on_basis(I[0,1], I[1,2])
((-q^4+1)/q^2)*B[1d]*B[2d+a1] + ((q^4-1)/q^2)*B[1d]*B[a1]
+ B[2d]*B[1d+a1] + (-q^4-q^2)*B[a0]
+ ((q^2+1)/q^2)*B[3d+a1] + ((q^6+q^4-q^2-1)/q^2)*B[1d+a1]
sage: Q.product_on_basis(I[1,2], I[0,-1])
B[a0]*B[2d] + ((-q^4+1)/q^2)*B[1d+a0]*B[1d]
+ ((q^4-1)/q^2)*B[1d]*B[a1] + ((q^2+1)/q^2)*B[2d+a0]
+ ((-q^2-1)/q^2)*B[1d+a1]
sage: Q.product_on_basis(I[1,2], I[0,-4])
((q^4-1)/q^2)*B[2d+a0]*B[1d] + B[3d+a0]*B[2d]
+ ((-q^4+1)/q^2)*B[4d+a0]*B[1d] + (-q^4-q^2)*B[1d+a0]
+ ((q^6+q^4-q^2-1)/q^2)*B[3d+a0] + ((q^2+1)/q^2)*B[5d+a0]
```

```
>>> from sage.all import *
>>> O = lie_algebras.OnsagerAlgebra(QQ)
>>> Q = O.quantum_group()
```

(continues on next page)

(continued from previous page)

```

>>> I = Q._indices.gens()
>>> Q.product_on_basis(I[Integer(1), Integer(21)]**Integer(2), I[Integer(1),
↪ Integer(31)]**Integer(3))
B[21d]^2*B[31d]^3
>>> Q.product_on_basis(I[Integer(1), Integer(31)]**Integer(3), I[Integer(1),
↪ Integer(21)]**Integer(2))
B[21d]^2*B[31d]^3
>>> Q.product_on_basis(I[Integer(0), Integer(8)], I[Integer(0), Integer(6)])
B[8d+a1]*B[6d+a1]
>>> Q.product_on_basis(I[Integer(0), -Integer(8)], I[Integer(0), Integer(6)])
B[7d+a0]*B[6d+a1]
>>> Q.product_on_basis(I[Integer(0), -Integer(6)], I[Integer(0), -Integer(8)])
B[5d+a0]*B[7d+a0]
>>> Q.product_on_basis(I[Integer(0), -Integer(6)], I[Integer(1), Integer(2)])
B[5d+a0]*B[2d]
>>> Q.product_on_basis(I[Integer(1), Integer(6)], I[Integer(0), Integer(2)])
B[6d]*B[2d+a1]

>>> Q.product_on_basis(I[Integer(0), Integer(1)], I[Integer(0), Integer(2)])
1/q^2*B[2d+a1]*B[1d+a1] - B[1d]
>>> Q.product_on_basis(I[Integer(0), -Integer(3)], I[Integer(0), -Integer(1)])
1/q^2*B[a0]*B[2d+a0] + ((-q^2+1)/q^2)*B[1d+a0]^2 - B[2d]
>>> Q.product_on_basis(I[Integer(0), Integer(2)], I[Integer(0), -Integer(1)])
q^2*B[a0]*B[2d+a1] + ((q^4-1)/q^2)*B[1d+a1]*B[a1]
+ (-q^2+1)*B[1d] + q^2*B[3d]
>>> Q.product_on_basis(I[Integer(0), Integer(2)], I[Integer(1), Integer(1)])
B[1d]*B[2d+a1] + (q^2+1)*B[3d+a1] + (-q^2-1)*B[1d+a1]
>>> Q.product_on_basis(I[Integer(0), Integer(1)], I[Integer(1), Integer(2)])
((-q^4+1)/q^2)*B[1d]*B[2d+a1] + ((q^4-1)/q^2)*B[1d]*B[a1]
+ B[2d]*B[1d+a1] + (-q^4-q^2)*B[a0]
+ ((q^2+1)/q^2)*B[3d+a1] + ((q^6+q^4-q^2-1)/q^2)*B[1d+a1]
>>> Q.product_on_basis(I[Integer(1), Integer(2)], I[Integer(0), -Integer(1)])
B[a0]*B[2d] + ((-q^4+1)/q^2)*B[1d+a0]*B[1d]
+ ((q^4-1)/q^2)*B[1d]*B[a1] + ((q^2+1)/q^2)*B[2d+a0]
+ ((-q^2-1)/q^2)*B[1d+a1]
>>> Q.product_on_basis(I[Integer(1), Integer(2)], I[Integer(0), -Integer(4)])
((q^4-1)/q^2)*B[2d+a0]*B[1d] + B[3d+a0]*B[2d]
+ ((-q^4+1)/q^2)*B[4d+a0]*B[1d] + (-q^4-q^2)*B[1d+a0]
+ ((q^6+q^4-q^2-1)/q^2)*B[3d+a0] + ((q^2+1)/q^2)*B[5d+a0]

```

q()Return the parameter q of self.**EXAMPLES:**

```

sage: O = lie_algebras.OnsagerAlgebra(QQ)
sage: Q = O.quantum_group()
sage: Q.q()
q

```

```

>>> from sage.all import *
>>> O = lie_algebras.OnsagerAlgebra(QQ)

```

(continues on next page)

(continued from previous page)

```
>>> Q = O.quantum_group()
>>> Q.q()
q
```

some_elements()

Return some elements of `self`.

EXAMPLES:

```
sage: O = lie_algebras.OnsagerAlgebra(QQ)
sage: Q = O.quantum_group()
sage: Q.some_elements()
[B[a1], B[3d+a1], B[a0], B[1d], B[4d]]
```

```
>>> from sage.all import *
>>> O = lie_algebras.OnsagerAlgebra(QQ)
>>> Q = O.quantum_group()
>>> Q.some_elements()
[B[a1], B[3d+a1], B[a0], B[1d], B[4d]]
```

9.2.16 The Poincare-Birkhoff-Witt Basis For A Universal Enveloping Algebra

AUTHORS:

- Travis Scrimshaw (2013-11-03): Initial version
- Travis Scrimshaw (2024-01-02): Adding the center

```
class sage.algebras.lie_algebras.poincare_birkhoff_witt.PoincareBirkhoffWittBasis(g, basis_key,
                                                                                   prefix,
                                                                                   **kws)
```

Bases: `CombinatorialFreeModule`

The Poincare-Birkhoff-Witt (PBW) basis of the universal enveloping algebra of a Lie algebra.

Consider a Lie algebra $\mathfrak{g}$ with ordered basis $(b_1, \dots, b_n)$. Then the universal enveloping algebra $U(\mathfrak{g})$ is generated by $b_1, \dots, b_n$ and subject to the relations

$$[b_i, b_j] = \sum_{k=1}^n c_{ij}^k b_k$$

where c_{ij}^k are the structure coefficients of $\mathfrak{g}$. The Poincare-Birkhoff-Witt (PBW) basis is given by the monomials $b_1^{e_1} b_2^{e_2} \cdots b_n^{e_n}$. Specifically, we can rewrite $b_j b_i = b_i b_j + [b_j, b_i]$ where $j > i$, and we can repeat this to sort any monomial into

$$b_{i_1} \cdots b_{i_k} = b_1^{e_1} \cdots b_n^{e_n} + LOT$$

where LOT are lower order terms. Thus the PBW basis is a filtered basis for $U(\mathfrak{g})$.

EXAMPLES:

We construct the PBW basis of $\mathfrak{sl}_2$:

```
sage: L = lie_algebras.three_dimensional_by_rank(QQ, 3, names=['E', 'F', 'H'])
sage: PBW = L.pbw_basis()
```

```
>>> from sage.all import *
>>> L = lie_algebras.three_dimensional_by_rank(QQ, Integer(3), names=['E', 'F', 'H
↪'])
>>> PBW = L.pbw_basis()
```

We then do some computations; in particular, we check that $[E, F] = H$:

```
sage: E, F, H = PBW.algebra_generators()
sage: E * F
PBW['E']*PBW['F']
sage: F * E
PBW['E']*PBW['F'] - PBW['H']
sage: E * F - F * E
PBW['H']
```

```
>>> from sage.all import *
>>> E, F, H = PBW.algebra_generators()
>>> E * F
PBW['E']*PBW['F']
>>> F * E
PBW['E']*PBW['F'] - PBW['H']
>>> E * F - F * E
PBW['H']
```

Next we construct another instance of the PBW basis, but sorted in the reverse order:

```
sage: def neg_key(x):
.....:     return -L.basis().keys().index(x)
sage: PBW2 = L.pbw_basis(prefix='PBW2', basis_key=neg_key)
```

```
>>> from sage.all import *
>>> def neg_key(x):
...     return -L.basis().keys().index(x)
>>> PBW2 = L.pbw_basis(prefix='PBW2', basis_key=neg_key)
```

We then check the multiplication is preserved:

```
sage: PBW2(E) * PBW2(F)
PBW2['F']*PBW2['E'] + PBW2['H']
sage: PBW2(E*F)
PBW2['F']*PBW2['E'] + PBW2['H']
sage: F * E + H
PBW['E']*PBW['F']
```

```
>>> from sage.all import *
>>> PBW2(E) * PBW2(F)
PBW2['F']*PBW2['E'] + PBW2['H']
>>> PBW2(E*F)
PBW2['F']*PBW2['E'] + PBW2['H']
>>> F * E + H
PBW['E']*PBW['F']
```

We now construct the PBW basis for Lie algebra of regular vector fields on $\mathbb{C}^\times$:


```

sage: L = lie_algebras.regular_vector_fields(QQ)
sage: PBW = L.pbw_basis()
sage: G = PBW.algebra_generators()
sage: G[2] * G[3]
PBW[2]*PBW[3]
sage: G[3] * G[2]
PBW[2]*PBW[3] + PBW[5]
sage: G[-2] * G[3] * G[2]
PBW[-2]*PBW[2]*PBW[3] + PBW[-2]*PBW[5]

```

```

>>> from sage.all import *
>>> L = lie_algebras.regular_vector_fields(QQ)
>>> PBW = L.pbw_basis()
>>> G = PBW.algebra_generators()
>>> G[Integer(2)] * G[Integer(3)]
PBW[2]*PBW[3]
>>> G[Integer(3)] * G[Integer(2)]
PBW[2]*PBW[3] + PBW[5]
>>> G[-Integer(2)] * G[Integer(3)] * G[Integer(2)]
PBW[-2]*PBW[2]*PBW[3] + PBW[-2]*PBW[5]

```

Todo

When the Lie algebra is finite dimensional, set the ordering of the basis elements, translate the structure coefficients, and work with fixed-length lists as the exponent vectors. This way we only will run any nontrivial sorting only once and avoid other potentially expensive comparisons between keys.

class Element

Bases: `IndexedFreeModuleElement`

algebra_generators()

Return the algebra generators of `self`.

EXAMPLES:

```

sage: L = lie_algebras.sl(QQ, 2)
sage: PBW = L.pbw_basis()
sage: PBW.algebra_generators()
Finite family {alpha[1]: PBW[alpha[1]], alphacheck[1]: PBW[alphacheck[1]], -
↪alpha[1]: PBW[-alpha[1]]}

```

```

>>> from sage.all import *
>>> L = lie_algebras.sl(QQ, Integer(2))
>>> PBW = L.pbw_basis()
>>> PBW.algebra_generators()
Finite family {alpha[1]: PBW[alpha[1]], alphacheck[1]: PBW[alphacheck[1]], -
↪alpha[1]: PBW[-alpha[1]]}

```

casimir_element(*order=2, *args, **kws*)

Return the Casimir element of `self`.

See also`casimir_element()`**INPUT:**

- `order` – (default: 2) the order of the Casimir element

EXAMPLES:

```

sage: L = LieAlgebra(QQ, cartan_type=['G', 2])
sage: U = L.pbw_basis()
sage: C = U.casimir_element(); C
1/4*PBW[alpha[2]]*PBW[-alpha[2]] + 1/12*PBW[alpha[1]]*PBW[-alpha[1]]
+ 1/12*PBW[alpha[1] + alpha[2]]*PBW[-alpha[1] - alpha[2]] + 1/
↪12*PBW[2*alpha[1] + alpha[2]]*PBW[-2*alpha[1] - alpha[2]]
+ 1/4*PBW[3*alpha[1] + alpha[2]]*PBW[-3*alpha[1] - alpha[2]]
+ 1/4*PBW[3*alpha[1] + 2*alpha[2]]*PBW[-3*alpha[1] - 2*alpha[2]]
+ 1/12*PBW[alphacheck[1]]^2 + 1/4*PBW[alphacheck[1]]*PBW[alphacheck[2]]
+ 1/4*PBW[alphacheck[2]]^2 - 5/12*PBW[alphacheck[1]] - 3/4*PBW[alphacheck[2]]
sage: all(g * C == C * g for g in U.algebra_generators())
True

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, cartan_type=['G', Integer(2)])
>>> U = L.pbw_basis()
>>> C = U.casimir_element(); C
1/4*PBW[alpha[2]]*PBW[-alpha[2]] + 1/12*PBW[alpha[1]]*PBW[-alpha[1]]
+ 1/12*PBW[alpha[1] + alpha[2]]*PBW[-alpha[1] - alpha[2]] + 1/
↪12*PBW[2*alpha[1] + alpha[2]]*PBW[-2*alpha[1] - alpha[2]]
+ 1/4*PBW[3*alpha[1] + alpha[2]]*PBW[-3*alpha[1] - alpha[2]]
+ 1/4*PBW[3*alpha[1] + 2*alpha[2]]*PBW[-3*alpha[1] - 2*alpha[2]]
+ 1/12*PBW[alphacheck[1]]^2 + 1/4*PBW[alphacheck[1]]*PBW[alphacheck[2]]
+ 1/4*PBW[alphacheck[2]]^2 - 5/12*PBW[alphacheck[1]] - 3/4*PBW[alphacheck[2]]
>>> all(g * C == C * g for g in U.algebra_generators())
True

```

center()

Return the center of self.

See also`CenterUEA`**EXAMPLES:**

```

sage: g = LieAlgebra(QQ, cartan_type=['A', 2])
sage: U = g.pbw_basis()
sage: U.center()
Center of Universal enveloping algebra of Lie algebra of ['A', 2]
in the Chevalley basis in the Poincare-Birkhoff-Witt basis

sage: g = lie_algebras.Heisenberg(GF(3), 4)

```

(continues on next page)

(continued from previous page)

```
sage: U = g.pbw_basis()
sage: U.center()
Center of Universal enveloping algebra of Heisenberg algebra of rank 4
over Finite Field of size 3 in the Poincare-Birkhoff-Witt basis
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['A', Integer(2)])
>>> U = g.pbw_basis()
>>> U.center()
Center of Universal enveloping algebra of Lie algebra of ['A', 2]
in the Chevalley basis in the Poincare-Birkhoff-Witt basis

>>> g = lie_algebras.Heisenberg(GF(Integer(3)), Integer(4))
>>> U = g.pbw_basis()
>>> U.center()
Center of Universal enveloping algebra of Heisenberg algebra of rank 4
over Finite Field of size 3 in the Poincare-Birkhoff-Witt basis
```

degree_on_basis(m)

Return the degree of the basis element indexed by m .

EXAMPLES:

```
sage: L = lie_algebras.sl(QQ, 2)
sage: PBW = L.pbw_basis()
sage: E,H,F = PBW.algebra_generators()
sage: PBW.degree_on_basis(E.leading_support())
1
sage: m = ((H*F)^10).trailing_support(key=PBW._monomial_key) # long time
sage: PBW.degree_on_basis(m) # long time
20
sage: ((H*F*E)^4).maximal_degree() # long time
12
```

```
>>> from sage.all import *
>>> L = lie_algebras.sl(QQ, Integer(2))
>>> PBW = L.pbw_basis()
>>> E,H,F = PBW.algebra_generators()
>>> PBW.degree_on_basis(E.leading_support())
1
>>> m = ((H*F)**Integer(10)).trailing_support(key=PBW._monomial_key) # long_
↳time
>>> PBW.degree_on_basis(m) # long time
20
>>> ((H*F*E)**Integer(4)).maximal_degree() # long time
12
```

gens()

Return the algebra generators of `self`.

EXAMPLES:

```

sage: L = lie_algebras.sl(QQ, 2)
sage: PBW = L.pbw_basis()
sage: PBW.algebra_generators()
Finite family {alpha[1]: PBW[alpha[1]], alphacheck[1]: PBW[alphacheck[1]], -
↪alpha[1]: PBW[-alpha[1]]}

```

```

>>> from sage.all import *
>>> L = lie_algebras.sl(QQ, Integer(2))
>>> PBW = L.pbw_basis()
>>> PBW.algebra_generators()
Finite family {alpha[1]: PBW[alpha[1]], alphacheck[1]: PBW[alphacheck[1]], -
↪alpha[1]: PBW[-alpha[1]]}

```

lie_algebra()

Return the underlying Lie algebra of self.

EXAMPLES:

```

sage: L = lie_algebras.sl(QQ, 2)
sage: PBW = L.pbw_basis()
sage: PBW.lie_algebra() is L
True

```

```

>>> from sage.all import *
>>> L = lie_algebras.sl(QQ, Integer(2))
>>> PBW = L.pbw_basis()
>>> PBW.lie_algebra() is L
True

```

one_basis()

Return the basis element indexing 1.

EXAMPLES:

```

sage: L = lie_algebras.three_dimensional_by_rank(QQ, 3, names=['E', 'F', 'H'])
sage: PBW = L.pbw_basis()
sage: ob = PBW.one_basis(); ob
1
sage: ob.parent()
Free abelian monoid indexed by {'E', 'F', 'H'}

```

```

>>> from sage.all import *
>>> L = lie_algebras.three_dimensional_by_rank(QQ, Integer(3), names=['E', 'F',
↪'H'])
>>> PBW = L.pbw_basis()
>>> ob = PBW.one_basis(); ob
1
>>> ob.parent()
Free abelian monoid indexed by {'E', 'F', 'H'}

```

product_on_basis(lhs, rhs)

Return the product of the two basis elements lhs and rhs.

EXAMPLES:

```

sage: L = lie_algebras.three_dimensional_by_rank(QQ, 3, names=['E', 'F', 'H'])
sage: PBW = L.pbw_basis()
sage: I = PBW.indices()
sage: PBW.product_on_basis(I.gen('E'), I.gen('F'))
PBW['E']*PBW['F']
sage: PBW.product_on_basis(I.gen('E'), I.gen('H'))
PBW['E']*PBW['H']
sage: PBW.product_on_basis(I.gen('H'), I.gen('E'))
PBW['E']*PBW['H'] + 2*PBW['E']
sage: PBW.product_on_basis(I.gen('F'), I.gen('E'))
PBW['E']*PBW['F'] - PBW['H']
sage: PBW.product_on_basis(I.gen('F'), I.gen('H'))
PBW['F']*PBW['H']
sage: PBW.product_on_basis(I.gen('H'), I.gen('F'))
PBW['F']*PBW['H'] - 2*PBW['F']
sage: PBW.product_on_basis(I.gen('H')**2, I.gen('F')**2)
PBW['F']^2*PBW['H']^2 - 8*PBW['F']^2*PBW['H'] + 16*PBW['F']^2

sage: E,F,H = PBW.algebra_generators()
sage: E*F - F*E
PBW['H']
sage: H * F * E
PBW['E']*PBW['F']*PBW['H'] - PBW['H']^2
sage: E * F * H * E
PBW['E']^2*PBW['F']*PBW['H'] + 2*PBW['E']^2*PBW['F']
- PBW['E']*PBW['H']^2 - 2*PBW['E']*PBW['H']

```

```

>>> from sage.all import *
>>> L = lie_algebras.three_dimensional_by_rank(QQ, Integer(3), names=['E', 'F',
↳ 'H'])
>>> PBW = L.pbw_basis()
>>> I = PBW.indices()
>>> PBW.product_on_basis(I.gen('E'), I.gen('F'))
PBW['E']*PBW['F']
>>> PBW.product_on_basis(I.gen('E'), I.gen('H'))
PBW['E']*PBW['H']
>>> PBW.product_on_basis(I.gen('H'), I.gen('E'))
PBW['E']*PBW['H'] + 2*PBW['E']
>>> PBW.product_on_basis(I.gen('F'), I.gen('E'))
PBW['E']*PBW['F'] - PBW['H']
>>> PBW.product_on_basis(I.gen('F'), I.gen('H'))
PBW['F']*PBW['H']
>>> PBW.product_on_basis(I.gen('H'), I.gen('F'))
PBW['F']*PBW['H'] - 2*PBW['F']
>>> PBW.product_on_basis(I.gen('H')**Integer(2), I.gen('F')**Integer(2))
PBW['F']^2*PBW['H']^2 - 8*PBW['F']^2*PBW['H'] + 16*PBW['F']^2

>>> E,F,H = PBW.algebra_generators()
>>> E*F - F*E
PBW['H']
>>> H * F * E
PBW['E']*PBW['F']*PBW['H'] - PBW['H']^2

```

(continues on next page)

(continued from previous page)

```
>>> E * F * H * E
PBW['E']^2*PBW['F']*PBW['H'] + 2*PBW['E']^2*PBW['F']
- PBW['E']*PBW['H']^2 - 2*PBW['E']*PBW['H']
```

```
class sage.algebras.lie_algebras.poincare_birkhoff_witt.PoincareBirkhoffWittBasisSemisimpleLieAlgebra
```

Bases: *PoincareBirkhoffWittBasis*

The Poincare-Birkhoff-Witt basis of a finite dimensional triangular Kac-Moody Lie algebra (i.e., a semisimple Lie algebra).

```
class Element
```

Bases: *Element*

```
transpose()
```

Return the transpose map of *self*.

This is the transpose map on the Lie algebra extended as an anti-involution of *self*.

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['D', 4])
sage: U = g.pbw_basis()
sage: e = U.e()
sage: f = U.f()
sage: elts = [e[1], e[1]*e[2], e[3]+e[4], e[1]*e[3]*e[4] + e[2],
....:         f[1], f[1]*f[2], f[3]+f[4], e[1]*e[3]*e[4] + e[2],
....:         e[1]*f[1], f[1]*e[1], (e[2]*f[2] - f[2]*e[2])^2]
sage: all((b*bp).transpose() == bp.transpose() * b.transpose()
....:      for b in elts for bp in elts)
True
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['D', Integer(4)])
>>> U = g.pbw_basis()
>>> e = U.e()
>>> f = U.f()
>>> elts = [e[Integer(1)], e[Integer(1)]*e[Integer(2)],
↳ e[Integer(3)]+e[Integer(4)], e[Integer(1)]*e[Integer(3)]*e[Integer(4)]↳
↳ + e[Integer(2)],
...       f[Integer(1)], f[Integer(1)]*f[Integer(2)],
↳ f[Integer(3)]+f[Integer(4)], e[Integer(1)]*e[Integer(3)]*e[Integer(4)]↳
↳ + e[Integer(2)],
...       e[Integer(1)]*f[Integer(1)], f[Integer(1)]*e[Integer(1)],
↳ (e[Integer(2)]*f[Integer(2)] - f[Integer(2)]*e[Integer(2)])**Integer(2)]
>>> all((b*bp).transpose() == bp.transpose() * b.transpose()
...      for b in elts for bp in elts)
True
```

```
contravariant_form(x, y)
```

Return the (universal) contravariant form of *x* and *y*.

Let $\varphi: U(\mathfrak{g}) \rightarrow U(\mathfrak{h})$ denote the projection onto the Cartan subalgebra and τ be the transpose map. The (universal) contravariant form is defined as

$$(x, y) = \varphi(\tau(x)y).$$

EXAMPLES:

```
sage: g = LieAlgebra(QQ, cartan_type=['G', 2])
sage: U = g.pbw_basis()
sage: f1, f2 = U.f()
sage: e1, e2 = U.e()
sage: U.contravariant_form(U.one(), U.one())
1
sage: U.contravariant_form(f1, f1)
PBW[alphacheck[1]]
sage: U.contravariant_form(f2, f2)
PBW[alphacheck[2]]
sage: U.contravariant_form(f1*f2, f1*f2)
PBW[alphacheck[1]]*PBW[alphacheck[2]] + 3*PBW[alphacheck[2]]
sage: U.contravariant_form(e1*e1*e2, e2*e1*e2)
0
sage: cas = U.casimir_element()
sage: ccc = U.contravariant_form(cas, cas); ccc
1/144*PBW[alphacheck[1]]^4 + 1/24*PBW[alphacheck[1]]^3*PBW[alphacheck[2]]
+ 5/48*PBW[alphacheck[1]]^2*PBW[alphacheck[2]]^2
+ 1/8*PBW[alphacheck[1]]*PBW[alphacheck[2]]^3 + 1/16*PBW[alphacheck[2]]^4
+ 5/72*PBW[alphacheck[1]]^3 + 1/3*PBW[alphacheck[1]]^2*PBW[alphacheck[2]]
+ 7/12*PBW[alphacheck[1]]*PBW[alphacheck[2]]^2 + 3/8*PBW[alphacheck[2]]^3
+ 25/144*PBW[alphacheck[1]]^2 + 5/8*PBW[alphacheck[1]]*PBW[alphacheck[2]]
+ 9/16*PBW[alphacheck[2]]^2
sage: ccc.parent() is U
True
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['G', Integer(2)])
>>> U = g.pbw_basis()
>>> f1, f2 = U.f()
>>> e1, e2 = U.e()
>>> U.contravariant_form(U.one(), U.one())
1
>>> U.contravariant_form(f1, f1)
PBW[alphacheck[1]]
>>> U.contravariant_form(f2, f2)
PBW[alphacheck[2]]
>>> U.contravariant_form(f1*f2, f1*f2)
PBW[alphacheck[1]]*PBW[alphacheck[2]] + 3*PBW[alphacheck[2]]
>>> U.contravariant_form(e1*e1*e2, e2*e1*e2)
0
>>> cas = U.casimir_element()
>>> ccc = U.contravariant_form(cas, cas); ccc
1/144*PBW[alphacheck[1]]^4 + 1/24*PBW[alphacheck[1]]^3*PBW[alphacheck[2]]
+ 5/48*PBW[alphacheck[1]]^2*PBW[alphacheck[2]]^2
+ 1/8*PBW[alphacheck[1]]*PBW[alphacheck[2]]^3 + 1/16*PBW[alphacheck[2]]^4
+ 5/72*PBW[alphacheck[1]]^3 + 1/3*PBW[alphacheck[1]]^2*PBW[alphacheck[2]]
```

(continues on next page)

(continued from previous page)

```

+ 7/12*PBW[alphacheck[1]]*PBW[alphacheck[2]]^2 + 3/8*PBW[alphacheck[2]]^3
+ 25/144*PBW[alphacheck[1]]^2 + 5/8*PBW[alphacheck[1]]*PBW[alphacheck[2]]
+ 9/16*PBW[alphacheck[2]]^2
>>> ccc.parent() is U
True

```

e (*i=None*)Return the generators e of self.

INPUT:

- i – (optional) if specified, return just the generator e_i

EXAMPLES:

```

sage: U = lie_algebras.so(QQ, 5).pbw_basis()
sage: U.e()
Finite family {1: PBW[alpha[1]], 2: PBW[alpha[2]]}
sage: U.e(1)
PBW[alpha[1]]

```

```

>>> from sage.all import *
>>> U = lie_algebras.so(QQ, Integer(5)).pbw_basis()
>>> U.e()
Finite family {1: PBW[alpha[1]], 2: PBW[alpha[2]]}
>>> U.e(Integer(1))
PBW[alpha[1]]

```

f (*i=None*)Return the generators f of self.

INPUT:

- i – (optional) if specified, return just the generator f_i

EXAMPLES:

```

sage: U = lie_algebras.so(QQ, 5).pbw_basis()
sage: U.f()
Finite family {1: PBW[-alpha[1]], 2: PBW[-alpha[2]]}
sage: U.f(1)
PBW[-alpha[1]]

```

```

>>> from sage.all import *
>>> U = lie_algebras.so(QQ, Integer(5)).pbw_basis()
>>> U.f()
Finite family {1: PBW[-alpha[1]], 2: PBW[-alpha[2]]}
>>> U.f(Integer(1))
PBW[-alpha[1]]

```

transpose ()

The transpose map.

EXAMPLES:


```
sage: g = LieAlgebra(QQ, cartan_type=['F', 4])
sage: U = g.pbw_basis()
sage: U.transpose
Generic endomorphism of Universal enveloping algebra of Lie algebra
of ['F', 4] in the Chevalley basis in the Poincare-Birkhoff-Witt basis
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['F', Integer(4)])
>>> U = g.pbw_basis()
>>> U.transpose
Generic endomorphism of Universal enveloping algebra of Lie algebra
of ['F', 4] in the Chevalley basis in the Poincare-Birkhoff-Witt basis
```

9.2.17 Quotients of Lie algebras

AUTHORS:

- Eero Hakavuori (2018-09-02): initial version

```
class sage.algebras.lie_algebras.quotient.LieQuotient_finite_dimensional_with_basis(I,
L,
names,
in-
dex_set,
cat-
e-
gory=None)
```

Bases: *LieAlgebraWithStructureCoefficients*

A quotient Lie algebra.

INPUT:

- I – an ideal or a list of generators of the ideal
- *ambient* – (optional) the Lie algebra to be quotiented; will be deduced from I if not given
- *names* – (optional) string or list of strings; names for the basis elements of the quotient. If *names* is a string, the basis will be named *names*₁, ..., *names*_{*n*}.

EXAMPLES:

The Engel Lie algebra as a quotient of the free nilpotent Lie algebra of step 3 with 2 generators:

```
sage: L = LieAlgebra(QQ, 2, step=3)
sage: L.inject_variables()
Defining X_1, X_2, X_12, X_112, X_122
sage: I = L.ideal(X_122)
sage: E = L.quotient(I); E
Lie algebra quotient L/I of dimension 4 over Rational Field where
L: Free Nilpotent Lie algebra on 5 generators (X_1, X_2, X_12, X_112, X_122) over
↳Rational Field
I: Ideal (X_122)
sage: E.category()
Join of Category of finite dimensional nilpotent Lie algebras with basis
over Rational Field and Category of subquotients of sets
```

(continues on next page)

(continued from previous page)

```

sage: E.basis().list()
[X_1, X_2, X_12, X_112]
sage: E.inject_variables()
Defining X_1, X_2, X_12, X_112
sage: X_1.bracket(X_2)
X_12
sage: X_1.bracket(X_12)
X_112
sage: X_2.bracket(X_12)
0

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(2), step=Integer(3))
>>> L.inject_variables()
Defining X_1, X_2, X_12, X_112, X_122
>>> I = L.ideal(X_122)
>>> E = L.quotient(I); E
Lie algebra quotient L/I of dimension 4 over Rational Field where
L: Free Nilpotent Lie algebra on 5 generators (X_1, X_2, X_12, X_112, X_122) over
↳Rational Field
I: Ideal (X_122)
>>> E.category()
Join of Category of finite dimensional nilpotent Lie algebras with basis
over Rational Field and Category of subquotients of sets
>>> E.basis().list()
[X_1, X_2, X_12, X_112]
>>> E.inject_variables()
Defining X_1, X_2, X_12, X_112
>>> X_1.bracket(X_2)
X_12
>>> X_1.bracket(X_12)
X_112
>>> X_2.bracket(X_12)
0

```

Shorthand for taking a quotient without creating an ideal first:

```

sage: E2 = L.quotient(X_122); E2
Lie algebra quotient L/I of dimension 4 over Rational Field where
L: Free Nilpotent Lie algebra on 5 generators (X_1, X_2, X_12, X_112, X_122) over
↳Rational Field
I: Ideal (X_122)
sage: E is E2
True

```

```

>>> from sage.all import *
>>> E2 = L.quotient(X_122); E2
Lie algebra quotient L/I of dimension 4 over Rational Field where
L: Free Nilpotent Lie algebra on 5 generators (X_1, X_2, X_12, X_112, X_122) over
↳Rational Field
I: Ideal (X_122)
>>> E is E2

```

(continues on next page)

(continued from previous page)

True

Custom names for the basis can be given:

```
sage: E.<X,Y,Z,W> = L.quotient(X_122)
sage: E.basis().list()
[X, Y, Z, W]
sage: X.bracket(Z)
W
sage: Y.bracket(Z)
0
```

```
>>> from sage.all import *
>>> E = L.quotient(X_122, names=('X', 'Y', 'Z', 'W',)); (X, Y, Z, W,) = E._first_
↳ngens(4)
>>> E.basis().list()
[X, Y, Z, W]
>>> X.bracket(Z)
W
>>> Y.bracket(Z)
0
```

The elements can be relabeled linearly by passing a string to the `names` parameter:

```
sage: E = L.quotient(X_122, names='Y')
sage: E.basis().list()
[Y_1, Y_2, Y_3, Y_4]
sage: E.inject_variables()
Defining Y_1, Y_2, Y_3, Y_4
sage: Y_1.bracket(Y_3)
Y_4
sage: Y_2.bracket(Y_3)
0
```

```
>>> from sage.all import *
>>> E = L.quotient(X_122, names='Y')
>>> E.basis().list()
[Y_1, Y_2, Y_3, Y_4]
>>> E.inject_variables()
Defining Y_1, Y_2, Y_3, Y_4
>>> Y_1.bracket(Y_3)
Y_4
>>> Y_2.bracket(Y_3)
0
```

Conversion from the ambient Lie algebra uses the quotient projection:

```
sage: L = LieAlgebra(QQ, 2, step=3)
sage: L.inject_variables()
Defining X_1, X_2, X_12, X_112, X_122
sage: E = L.quotient(X_122, names='Y')
sage: E(X_1), E(X_2), E(X_12), E(X_112), E(X_122)
```

(continues on next page)

(continued from previous page)

```
(Y_1, Y_2, Y_3, Y_4, 0)
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(2), step=Integer(3))
>>> L.inject_variables()
Defining X_1, X_2, X_12, X_112, X_122
>>> E = L.quotient(X_122, names='Y')
>>> E(X_1), E(X_2), E(X_12), E(X_112), E(X_122)
(Y_1, Y_2, Y_3, Y_4, 0)
```

A non-stratifiable Lie algebra as a quotient of the free nilpotent Lie algebra of step 4 on 2 generators by the relation $[X_2, [X_1, X_2]] = [X_1, [X_1, [X_1, X_2]]]$:

```
sage: L = LieAlgebra(QQ, 2, step=4)
sage: X_1, X_2 = L.homogeneous_component_basis(1)
sage: rel = L[X_2, [X_1, X_2]] - L[X_1, [X_1, [X_1, X_2]]]
sage: Q = L.quotient(rel, names='Y')
sage: Q.dimension()
5
sage: Q.inject_variables()
Defining Y_1, Y_2, Y_3, Y_4, Y_5
sage: lcs = Q.lower_central_series()
sage: [I.basis().list() for I in lcs]
[[Y_1, Y_2, Y_3, Y_4, Y_5], [Y_3, Y_4, Y_5], [Y_4, Y_5], [Y_5], []]
sage: Y_2.bracket(Y_3)
-Y_5
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(2), step=Integer(4))
>>> X_1, X_2 = L.homogeneous_component_basis(Integer(1))
>>> rel = L[X_2, [X_1, X_2]] - L[X_1, [X_1, [X_1, X_2]]]
>>> Q = L.quotient(rel, names='Y')
>>> Q.dimension()
5
>>> Q.inject_variables()
Defining Y_1, Y_2, Y_3, Y_4, Y_5
>>> lcs = Q.lower_central_series()
>>> [I.basis().list() for I in lcs]
[[Y_1, Y_2, Y_3, Y_4, Y_5], [Y_3, Y_4, Y_5], [Y_4, Y_5], [Y_5], []]
>>> Y_2.bracket(Y_3)
-Y_5
```

Quotients when the base ring is not a field are not implemented:

```
sage: L = lie_algebras.Heisenberg(ZZ, 1)
sage: L.quotient(L.an_element())
Traceback (most recent call last):
...
NotImplementedError: quotients over non-fields not implemented
```

```
>>> from sage.all import *
>>> L = lie_algebras.Heisenberg(ZZ, Integer(1))
```

(continues on next page)

(continued from previous page)

```
>>> L.quotient(L.an_element())
Traceback (most recent call last):
...
NotImplementedError: quotients over non-fields not implemented
```

ambient()Return the ambient Lie algebra of `self`.

EXAMPLES:

```
sage: L.<x,y,z> = LieAlgebra(QQ, 2, step=2)
sage: Q = L.quotient(z)
sage: Q.ambient() == L
True
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(2), step=Integer(2), names=('x', 'y', 'z'));
↳ (x, y, z) = L._first_ngens(3)
>>> Q = L.quotient(z)
>>> Q.ambient() == L
True
```

defining_ideal()

Return the ideal generating this quotient Lie algebra.

EXAMPLES:

```
sage: L = lie_algebras.Heisenberg(QQ, 1)
sage: p,q,z = L.basis()
sage: Q = L.quotient(p)
sage: Q.defining_ideal()
Ideal (p1) of Heisenberg algebra of rank 1 over Rational Field
```

```
>>> from sage.all import *
>>> L = lie_algebras.Heisenberg(QQ, Integer(1))
>>> p,q,z = L.basis()
>>> Q = L.quotient(p)
>>> Q.defining_ideal()
Ideal (p1) of Heisenberg algebra of rank 1 over Rational Field
```

from_vector(`v`, `order=None`, `coerce=False`)Return the element of `self` corresponding to the vector `v`.

INPUT:

- `v` – a vector in `self.module()` or `self.ambient().module()`

EXAMPLES:

An element from a vector of the intrinsic module:

```
sage: L.<X,Y,Z> = LieAlgebra(QQ, 3, abelian=True)
sage: Q = L.quotient(X + Y + Z)
sage: Q.dimension()
```

(continues on next page)

(continued from previous page)

```

2
sage: el = Q.from_vector([1, 2]); el
X + 2*Y
sage: el.parent() == Q
True

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(3), abelian=True, names=('X', 'Y', 'Z')); (X, Y, Z) = L._first_ngens(3)
>>> Q = L.quotient(X + Y + Z)
>>> Q.dimension()
2
>>> el = Q.from_vector([Integer(1), Integer(2)]); el
X + 2*Y
>>> el.parent() == Q
True

```

An element from a vector of the ambient module:

```

sage: el = Q.from_vector([1, 2, 3]); el
-2*X - Y
sage: el.parent() == Q
True

```

```

>>> from sage.all import *
>>> el = Q.from_vector([Integer(1), Integer(2), Integer(3)]); el
-2*X - Y
>>> el.parent() == Q
True

```

Check for the trivial ideal:

```

sage: L.<x,y,z> = LieAlgebra(GF(3), {('x','z'): {'x':1, 'y':1}, ('y','z'): {'y':1}})
sage: I = L.ideal([])
sage: Q = L.quotient(I)
sage: v = Q.an_element().to_vector()
sage: Q.from_vector(v)
x + y + z

```

```

>>> from sage.all import *
>>> L = LieAlgebra(GF(Integer(3)), {('x','z'): {'x':Integer(1), 'y':Integer(1)}, ('y','z'): {'y':Integer(1)}}, names=('x', 'y', 'z')); (x, y, z) = L._first_ngens(3)
>>> I = L.ideal([])
>>> Q = L.quotient(I)
>>> v = Q.an_element().to_vector()
>>> Q.from_vector(v)
x + y + z

```

lift(X)

Return some preimage of x under the quotient projection into `self`.

INPUT:

- x – an element of `self`

EXAMPLES:

```
sage: L.<x,y,z> = LieAlgebra(QQ, 2, step=2)
sage: Q = L.quotient(x + y)
sage: Q(y)
-x
sage: e1 = Q.lift(Q(y)); e1
-x
sage: e1.parent()
Free Nilpotent Lie algebra on 3 generators (x, y, z) over Rational Field
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(2), step=Integer(2), names=('x', 'y', 'z'));
↳ (x, y, z) = L._first_ngens(3)
>>> Q = L.quotient(x + y)
>>> Q(y)
-x
>>> e1 = Q.lift(Q(y)); e1
-x
>>> e1.parent()
Free Nilpotent Lie algebra on 3 generators (x, y, z) over Rational Field
```

retract (X)

Map x under the quotient projection to `self`.

INPUT:

- x – an element of the ambient Lie algebra

EXAMPLES:

```
sage: L = LieAlgebra(QQ, 3, step=2)
sage: L.inject_variables()
Defining X_1, X_2, X_3, X_12, X_13, X_23
sage: Q = L.quotient(X_1 + X_2 + X_3)
sage: Q.retract(X_1), Q.retract(X_2), Q.retract(X_3)
(X_1, X_2, -X_1 - X_2)
sage: all(Q.retract(Q.lift(X)) == X for X in Q.basis())
True
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(3), step=Integer(2))
>>> L.inject_variables()
Defining X_1, X_2, X_3, X_12, X_13, X_23
>>> Q = L.quotient(X_1 + X_2 + X_3)
>>> Q.retract(X_1), Q.retract(X_2), Q.retract(X_3)
(X_1, X_2, -X_1 - X_2)
>>> all(Q.retract(Q.lift(X)) == X for X in Q.basis())
True
```

9.2.18 Rank Two Heisenberg-Virasoro Algebras

AUTHORS:

- Travis Scrimshaw (2018-08): Initial version

class sage.algebras.lie_algebras.rank_two_heisenberg_virasoro.**RankTwoHeisenbergVirasoro**(R)
 Bases: *InfinitelyGeneratedLieAlgebra*, *IndexedGenerators*

The rank 2 Heisenberg-Virasoro algebra.

The *rank 2 Heisenberg-Virasoro* (Lie) algebra is the Lie algebra L spanned by the elements

$$\{t^\alpha, E(\alpha) \mid \alpha \in \mathbf{Z}^2 \setminus \{(0, 0)\}\} \cup \{K_1, K_2, K_3, K_4\},$$

which satisfy the relations

$$\begin{aligned} [t^\alpha, t^\beta] &= [K_i, L] = 0, \\ [t^\alpha, E(\beta)] &= \det \begin{pmatrix} \beta \\ \alpha \end{pmatrix} t^{\alpha+\beta} + \delta_{\alpha, -\beta} (\alpha_1 K_1 + \alpha_2 K_2), \\ [E(\alpha), E(\beta)] &= \det \begin{pmatrix} \beta \\ \alpha \end{pmatrix} E(\alpha + \beta) + \delta_{\alpha, -\beta} (\alpha_1 K_3 + \alpha_2 K_4), \end{aligned}$$

where $\alpha = (\alpha_1, \alpha_2)$ and δ_{xy} is the Kronecker delta.

EXAMPLES:

```
sage: L = lie_algebras.RankTwoHeisenbergVirasoro(QQ)
sage: K1, K2, K3, K4 = L.K()
sage: E2m1 = L.E(2, -1)
sage: Em21 = L.E(-2, 1)
sage: t2m1 = L.t(2, -1)
sage: t53 = L.t(5, 3)

sage: Em21.bracket(t2m1)
-2*K1 + K2
sage: t53.bracket(E2m1)
11*t(7, 2)
sage: E2m1.bracket(Em21)
2*K3 - K4
sage: E2m1.bracket(t2m1)
0

sage: all(x.bracket(y) == 0 for x in [K1, K2, K3, K4] for y in [E2m1, Em21, t2m1])
True
```

```
>>> from sage.all import *
>>> L = lie_algebras.RankTwoHeisenbergVirasoro(QQ)
>>> K1, K2, K3, K4 = L.K()
>>> E2m1 = L.E(Integer(2), -Integer(1))
>>> Em21 = L.E(-Integer(2), Integer(1))
>>> t2m1 = L.t(Integer(2), -Integer(1))
>>> t53 = L.t(Integer(5), Integer(3))

>>> Em21.bracket(t2m1)
-2*K1 + K2
```

(continues on next page)

(continued from previous page)

```

>>> t53.bracket(E2m1)
11*t(7, 2)
>>> E2m1.bracket(Em21)
2*K3 - K4
>>> E2m1.bracket(t2m1)
0
>>> all(x.bracket(y) == Integer(0) for x in [K1,K2,K3,K4] for y in [E2m1, Em21, ↵
↵t2m1])
True

```

REFERENCES:

- [LT2018]

E(a, b)Return the basis element $E(a, b)$ of `self`.

EXAMPLES:

```

sage: L = lie_algebras.RankTwoHeisenbergVirasoro(QQ)
sage: L.E(1,-2)
E(1, -2)

```

```

>>> from sage.all import *
>>> L = lie_algebras.RankTwoHeisenbergVirasoro(QQ)
>>> L.E(Integer(1),-Integer(2))
E(1, -2)

```

class ElementBases: *LieAlgebraElement***K**($i=None$)Return the basis element K_i of `self`.

EXAMPLES:

```

sage: L = lie_algebras.RankTwoHeisenbergVirasoro(QQ)
sage: L.K(1)
K1
sage: list(L.K())
[K1, K2, K3, K4]

```

```

>>> from sage.all import *
>>> L = lie_algebras.RankTwoHeisenbergVirasoro(QQ)
>>> L.K(Integer(1))
K1
>>> list(L.K())
[K1, K2, K3, K4]

```

bracket_on_basis(i, j)Return the bracket of basis elements indexed by i and j , where $i < j$.

EXAMPLES:

```

sage: L = lie_algebras.RankTwoHeisenbergVirasoro(QQ)
sage: v = L._v
sage: L.bracket_on_basis(('K', 2), ('t', v(3, -1)))
0
sage: L.bracket_on_basis(('K', 4), ('E', v(3, -1)))
0
sage: L.bracket_on_basis(('t', v(3, -1)), ('t', v(4, 3)))
0
sage: L.bracket_on_basis(('t', v(3, -1)), ('E', v(4, 3)))
-13*t(7, 2)
sage: L.bracket_on_basis(('t', v(2, 2)), ('E', v(1, 1)))
0
sage: L.bracket_on_basis(('t', v(3, -1)), ('E', v(-3, 1)))
3*K1 - K2
sage: L.bracket_on_basis(('E', v(3, -1)), ('E', v(4, 3)))
-13*E(7, 2)
sage: L.bracket_on_basis(('E', v(2, 2)), ('E', v(1, 1)))
0
sage: L.bracket_on_basis(('E', v(3, -1)), ('E', v(-3, 1)))
3*K3 - K4

```

```

>>> from sage.all import *
>>> L = lie_algebras.RankTwoHeisenbergVirasoro(QQ)
>>> v = L._v
>>> L.bracket_on_basis(('K', Integer(2)), ('t', v(Integer(3), -Integer(1))))
0
>>> L.bracket_on_basis(('K', Integer(4)), ('E', v(Integer(3), -Integer(1))))
0
>>> L.bracket_on_basis(('t', v(Integer(3), -Integer(1))), ('t', v(Integer(4),
↪ Integer(3))))
0
>>> L.bracket_on_basis(('t', v(Integer(3), -Integer(1))), ('E', v(Integer(4),
↪ Integer(3))))
-13*t(7, 2)
>>> L.bracket_on_basis(('t', v(Integer(2), Integer(2))), ('E', v(Integer(1),
↪ Integer(1))))
0
>>> L.bracket_on_basis(('t', v(Integer(3), -Integer(1))), ('E', v(-Integer(3),
↪ Integer(1))))
3*K1 - K2
>>> L.bracket_on_basis(('E', v(Integer(3), -Integer(1))), ('E', v(Integer(4),
↪ Integer(3))))
-13*E(7, 2)
>>> L.bracket_on_basis(('E', v(Integer(2), Integer(2))), ('E', v(Integer(1),
↪ Integer(1))))
0
>>> L.bracket_on_basis(('E', v(Integer(3), -Integer(1))), ('E', v(-Integer(3),
↪ Integer(1))))
3*K3 - K4

```

some_elements()

Return some elements of self.

EXAMPLES:

```
sage: L = lie_algebras.RankTwoHeisenbergVirasoro(QQ)
sage: L.some_elements()
[E(1, 1), E(-2, -2), E(0, 1),
 t(1, 1), t(4, -1), t(2, 3),
 K2, K4,
 K3 - 1/2*t(-1, 3) + E(1, -3) + E(2, 2)]
```

```
>>> from sage.all import *
>>> L = lie_algebras.RankTwoHeisenbergVirasoro(QQ)
>>> L.some_elements()
[E(1, 1), E(-2, -2), E(0, 1),
 t(1, 1), t(4, -1), t(2, 3),
 K2, K4,
 K3 - 1/2*t(-1, 3) + E(1, -3) + E(2, 2)]
```

$t(a, b)$

Return the basis element $t^{(a,b)}$ of self.

EXAMPLES:

```
sage: L = lie_algebras.RankTwoHeisenbergVirasoro(QQ)
sage: L.t(1, -2)
t(1, -2)
```

```
>>> from sage.all import *
>>> L = lie_algebras.RankTwoHeisenbergVirasoro(QQ)
>>> L.t(Integer(1), -Integer(2))
t(1, -2)
```

9.2.19 Representations of Lie algebras

AUTHORS:

- Travis Scrimshaw (2023-08-31): initial version

```
class sage.algebras.lie_algebras.representation.FaithfulRepresentationNilpotentPBW(L,
                                                                                     minimal=False)
```

Bases: `CombinatorialFreeModule`, `Representation_abstract`

Return a faithful representation of a nilpotent Lie algebra constructed using the PBW basis.

Let L be a k -step nilpotent Lie algebra. Define a weight function on elements in L by the lower central series of L . Then a faithful representation of L is $U(L)/U(L)^{k+1}$, where $U(L)^{k+1}$ is the (twosided) ideal of $U(L)$ generated by all monomials of weight at least $k+1$.

We can also expand the ideal keeping the property that $I \cap Z(L) = 0$. The resulting quotient $U(L)/I$ remains faithful and is a minimal faithful representation of L in the sense that it has no faithful submodules or quotients. (Note: this is not necessarily the smallest dimensional faithful representation of L .)

We consider an example of the rank 2 Heisenberg Lie algebra, but with a non-standard basis given by $a = p_1 + z$, $b = q_1$, and $c = q_1 + z$:

```

sage: scoeffs = {('a','b'): {'b':-1, 'c':1}, ('a','c'): {'b':-1, 'c':1}}
sage: L.<a,b,c> = LieAlgebra(QQ, scoeffs)
sage: TestSuite(L).run(elements=list(L.basis()))
sage: L.is_nilpotent()
True
sage: L.derived_series()
(Lie algebra on 3 generators (a, b, c) over Rational Field,
 Ideal (b - c) of Lie algebra on 3 generators (a, b, c) over Rational Field,
 Ideal () of Lie algebra on 3 generators (a, b, c) over Rational Field)
sage: F = L.faithful_representation()
sage: L.an_element() * F.an_element()
2*F[1, 0, 0] + 8*F[1, 1, 0] + 3*F[2, 0, 0] + 4*F[0, 1, 0]
+ 4*F[0, 2, 0] + 4*F[0, 0, 1]

sage: MF = L.faithful_representation(algorithm='minimal')
sage: MF.dimension()
3
sage: [MF.representation_matrix(be) for be in L.basis()]
[
[0 0 0] [ 0 0 0] [ 0 0 0]
[0 0 0] [ 0 0 -1] [ 1 0 -1]
[1 0 0], [ 0 0 0], [ 0 0 0]
]

```

```

>>> from sage.all import *
>>> scoeffs = {('a','b'): {'b':-Integer(1), 'c':Integer(1)}, ('a','c'): {'b':-
↳ Integer(1), 'c':Integer(1)}}
>>> L = LieAlgebra(QQ, scoeffs, names=('a', 'b', 'c')); (a, b, c) = L._first_
↳ ngens(3)
>>> TestSuite(L).run(elements=list(L.basis()))
>>> L.is_nilpotent()
True
>>> L.derived_series()
(Lie algebra on 3 generators (a, b, c) over Rational Field,
 Ideal (b - c) of Lie algebra on 3 generators (a, b, c) over Rational Field,
 Ideal () of Lie algebra on 3 generators (a, b, c) over Rational Field)
>>> F = L.faithful_representation()
>>> L.an_element() * F.an_element()
2*F[1, 0, 0] + 8*F[1, 1, 0] + 3*F[2, 0, 0] + 4*F[0, 1, 0]
+ 4*F[0, 2, 0] + 4*F[0, 0, 1]

>>> MF = L.faithful_representation(algorithm='minimal')
>>> MF.dimension()
3
>>> [MF.representation_matrix(be) for be in L.basis()]
[
[0 0 0] [ 0 0 0] [ 0 0 0]
[0 0 0] [ 0 0 -1] [ 1 0 -1]
[1 0 0], [ 0 0 0], [ 0 0 0]
]

```

An example with `minimal=True` for $H_2 \oplus A_1$, where A_1 is a 1-dimensional Abelian Lie algebra:

```

sage: scoeffs = {('a','b'): {'b':-1, 'c':1}, ('a','c'): {'b':-1, 'c':1}}
sage: L.<a,b,c,d> = LieAlgebra(QQ, scoeffs)
sage: F = L.faithful_representation(); F
Faithful 11 dimensional representation of Lie algebra on 4
generators (a, b, c, d) over Rational Field
sage: MF = L.faithful_representation(algorithm='minimal'); MF
Minimal faithful representation of Lie algebra on 4
generators (a, b, c, d) over Rational Field
sage: MF.dimension()
4

```

```

>>> from sage.all import *
>>> scoeffs = {('a','b'): {'b':-Integer(1), 'c':Integer(1)}, ('a','c'): {'b':-
↳ Integer(1), 'c':Integer(1)}}
>>> L = LieAlgebra(QQ, scoeffs, names=('a', 'b', 'c', 'd')); (a, b, c, d,) = L._
↳ first_ngens(4)
>>> F = L.faithful_representation(); F
Faithful 11 dimensional representation of Lie algebra on 4
generators (a, b, c, d) over Rational Field
>>> MF = L.faithful_representation(algorithm='minimal'); MF
Minimal faithful representation of Lie algebra on 4
generators (a, b, c, d) over Rational Field
>>> MF.dimension()
4

```

INPUT:

- `minimal` – boolean (default: `False`); whether to construct the minimal basis or not

REFERENCES:

- [BEdG2009]

class `Element`

Bases: `IndexedFreeModuleElement`

class `sage.algebras.lie_algebras.representation.FaithfulRepresentationPBWPosChar(L)`

Bases: `CombinatorialFreeModule`, `Representation_abstract`

A faithful representation of a finite dimensional Lie algebra in positive characteristic.

Warning

This is often a very large dimensional representation relative to the dimension of the Lie algebra.

ALGORITHM:

We implement the algorithm given in [deG2000] Section 6.6. Let L be a finite dimensional Lie algebra over a ring of characteristic p with basis $(b_1, \dots, b_n)$. We compute (monic) p -polynomials f_i such that $A = \text{ad}(b_i)$ (the adjoint action of b_i) solves $f_i(A) = 0$ by using minimal polynomial of A . The $(f_1, \dots, f_n)$ is a Gröbner basis for an ideal I of the universal enveloping algebra $U(L)$ such that the quotient $U(L)/I$ is a faithful representation of L .

EXAMPLES:

```

sage: sl2 = LieAlgebra(GF(3), cartan_type=['A',1])
sage: F = sl2.fiducial_representation()
sage: F
Faithful representation with p-multiplicities (1, 3, 1) of Lie algebra
of ['A', 1] in the Chevalley basis
sage: F.dimension()
243

```

```

>>> from sage.all import *
>>> sl2 = LieAlgebra(GF(Integer(3)), cartan_type=['A',Integer(1)])
>>> F = sl2.fiducial_representation()
>>> F
Faithful representation with p-multiplicities (1, 3, 1) of Lie algebra
of ['A', 1] in the Chevalley basis
>>> F.dimension()
243

```

class Element

Bases: `IndexedFreeModuleElement`

groebner_basis()

Return the defining Gröbner basis of self.

EXAMPLES:

```

sage: sp4 = LieAlgebra(GF(3), cartan_type=['C',2])
sage: F = sp4.fiducial_representation()
sage: F.groebner_basis()
[PBW[alpha[2]]^3,
 PBW[alpha[1]]^3,
 PBW[alpha[1] + alpha[2]]^3,
 PBW[2*alpha[1] + alpha[2]]^3,
 2*PBW[alphacheck[1]]^27 + PBW[alphacheck[1]],
 2*PBW[alphacheck[2]]^27 + PBW[alphacheck[2]],
 PBW[-alpha[2]]^3,
 PBW[-alpha[1]]^3,
 PBW[-alpha[1] - alpha[2]]^3,
 PBW[-2*alpha[1] - alpha[2]]^3]

```

```

>>> from sage.all import *
>>> sp4 = LieAlgebra(GF(Integer(3)), cartan_type=['C',Integer(2)])
>>> F = sp4.fiducial_representation()
>>> F.groebner_basis()
[PBW[alpha[2]]^3,
 PBW[alpha[1]]^3,
 PBW[alpha[1] + alpha[2]]^3,
 PBW[2*alpha[1] + alpha[2]]^3,
 2*PBW[alphacheck[1]]^27 + PBW[alphacheck[1]],
 2*PBW[alphacheck[2]]^27 + PBW[alphacheck[2]],
 PBW[-alpha[2]]^3,
 PBW[-alpha[1]]^3,
 PBW[-alpha[1] - alpha[2]]^3,
 PBW[-2*alpha[1] - alpha[2]]^3]

```

p_exponents()

Return the p -exponents of `self`.

Let p be the characteristic of the base ring of `self`. The p -exponents are the exponents m_i such that the i -th p -polynomial f_i is of degree p^{m_i} .

EXAMPLES:

```
sage: sp4 = LieAlgebra(GF(3), cartan_type=['C', 2])
sage: F = sp4.faithful_representation()
sage: F.p_exponents()
(1, 1, 1, 1, 3, 3, 1, 1, 1, 1)
```

```
>>> from sage.all import *
>>> sp4 = LieAlgebra(GF(Integer(3)), cartan_type=['C', Integer(2)])
>>> F = sp4.faithful_representation()
>>> F.p_exponents()
(1, 1, 1, 1, 3, 3, 1, 1, 1, 1)
```

```
class sage.algebras.lie_algebras.representation.RepresentationByMorphism(lie_algebra, f,
                                                                           index_set,
                                                                           on_basis, category,
                                                                           **kwargs)
```

Bases: `CombinatorialFreeModule`, `Representation_abstract`

Representation of a Lie algebra defined by a Lie algebra morphism.

INPUT:

- `lie_algebra` – a Lie algebra
- `f` – the Lie algebra morphism defining the action of the basis elements of `lie_algebra`
- `index_set` – (optional) the index set of the module basis
- `on_basis` – boolean (default: `False`); the function f defines a map from the basis elements or from a generic element of `lie_algebra`

If f is encoded as a dict or Family, then the keys must be indices of the basis of `lie_algebra` and the values being the corresponding matrix defining the action. This sets `on_basis=True`.

EXAMPLES:

```
sage: L.<x,y> = LieAlgebra(QQ, {('x','y'): {'y':1}})
sage: f = {x: Matrix([[1,0],[0,0]]), y: Matrix([[0,1],[0,0]])}
sage: L.representation(f)
Representation of Lie algebra on 2 generators (x, y) over Rational Field defined_
↳by:
      [1 0]
x |--> [0 0]
      [0 1]
y |--> [0 0]
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, {('x','y'): {'y':Integer(1)}}, names=('x', 'y')); (x, y) = L._first_ngens(2)
↳= L._first_ngens(2)
>>> f = {x: Matrix([[Integer(1), Integer(0)], [Integer(0), Integer(0)]]), y: }
```

(continues on next page)

(continued from previous page)

```

↪Matrix([[Integer(0), Integer(1)], [Integer(0), Integer(0)]])
>>> L.representation(f)
Representation of Lie algebra on 2 generators (x, y) over Rational Field defined_
↪by:
      [1 0]
x |--> [0 0]
      [0 1]
y |--> [0 0]

```

We construct the direct sum of two copies of the trivial representation for an infinite dimensional Lie algebra:

```

sage: L = lie_algebras.Affine(QQ, ['E', 6, 1])
sage: R = L.representation(lambda b: matrix.zero(QQ, 2), index_set=['a', 'b'])
sage: x = L.an_element()
sage: v = R.an_element(); v
2*R['a'] + 2*R['b']
sage: x * v
0

```

```

>>> from sage.all import *
>>> L = lie_algebras.Affine(QQ, ['E', Integer(6), Integer(1)])
>>> R = L.representation(lambda b: matrix.zero(QQ, Integer(2)), index_set=['a', 'b'
↪'])
>>> x = L.an_element()
>>> v = R.an_element(); v
2*R['a'] + 2*R['b']
>>> x * v
0

```

We construct a finite dimensional representation of the affine Lie algebra of type $A_2^{(1)}$:

```

sage: L = lie_algebras.Affine(QQ, ['A', 2, 1]).derived_subalgebra()
sage: Phi_plus = list(RootSystem(['A', 2]).root_lattice().positive_roots())
sage: def aff_action(key):
.....:     mat = matrix.zero(QQ, 3)
.....:     if key == 'c': # central element
.....:         return mat
.....:     b, ell = key
.....:     if b in Phi_plus: # positive root
.....:         ind = tuple(sorted(b.to_ambient().support()))
.....:         mat[ind] = 1
.....:         if ind[0] + 1 != ind[1]:
.....:             mat[ind] = -1
.....:     elif -b in Phi_plus: # negative root
.....:         ind = tuple(sorted(b.to_ambient().support(), reverse=True))
.....:         mat[ind] = 1
.....:         if ind[0] - 1 != ind[1]:
.....:             mat[ind] = -1
.....:     else: # must be in the Cartan
.....:         i = b.leading_support()
.....:         mat[i, i] = -1
.....:         mat[i-1, i-1] = 1

```

(continues on next page)

(continued from previous page)

```

.....:     return mat
sage: F = Family(L.basis(), aff_action, name="lifted natural repr")
sage: R = L.representation(index_set=range(1,4), on_basis=F)
sage: x = L.an_element(); x
(E[alpha[2]] + E[alpha[1]] + h1 + h2 + E[-alpha[2]] + E[-alpha[1]])#t^0
+ (E[-alpha[1] - alpha[2]])#t^1 + (E[alpha[1] + alpha[2]])#t^-1 + c
sage: v = R.an_element(); v
2*R[1] + 2*R[2] + 3*R[3]
sage: x * v
R[1] + 5*R[2] - 3*R[3]
sage: R._test_representation() # verify that it is a representation

```

```

>>> from sage.all import *
>>> L = lie_algebras.Affine(QQ, ['A', Integer(2), Integer(1)]).derived_subalgebra()
>>> Phi_plus = list(RootSystem(['A', Integer(2)]).root_lattice().positive_roots())
>>> def aff_action(key):
...     mat = matrix.zero(QQ, Integer(3))
...     if key == 'c': # central element
...         return mat
...     b, ell = key
...     if b in Phi_plus: # positive root
...         ind = tuple(sorted(b.to_ambient().support()))
...         mat[ind] = Integer(1)
...         if ind[Integer(0)] + Integer(1) != ind[Integer(1)]:
...             mat[ind] = -Integer(1)
...     elif -b in Phi_plus: # negative root
...         ind = tuple(sorted(b.to_ambient().support(), reverse=True))
...         mat[ind] = Integer(1)
...         if ind[Integer(0)] - Integer(1) != ind[Integer(1)]:
...             mat[ind] = -Integer(1)
...     else: # must be in the Cartan
...         i = b.leading_support()
...         mat[i,i] = -Integer(1)
...         mat[i-Integer(1), i-Integer(1)] = Integer(1)
...     return mat
>>> F = Family(L.basis(), aff_action, name="lifted natural repr")
>>> R = L.representation(index_set=range(Integer(1), Integer(4)), on_basis=F)
>>> x = L.an_element(); x
(E[alpha[2]] + E[alpha[1]] + h1 + h2 + E[-alpha[2]] + E[-alpha[1]])#t^0
+ (E[-alpha[1] - alpha[2]])#t^1 + (E[alpha[1] + alpha[2]])#t^-1 + c
>>> v = R.an_element(); v
2*R[1] + 2*R[2] + 3*R[3]
>>> x * v
R[1] + 5*R[2] - 3*R[3]
>>> R._test_representation() # verify that it is a representation

```

```
class Element
```

```
    Bases: IndexedFreeModuleElement
```

```
class sage.algebras.lie_algebras.representation.Representation_abstract(lie_algebra)
```

```
    Bases: object
```

```
Mixin class for (left) representations of Lie algebras.
```

INPUT:

- `lie_algebra` – a Lie algebra

`lie_algebra()`

Return the Lie algebra whose representation `self` is.

EXAMPLES:

```
sage: L = lie_algebras.sl(QQ, 4)
sage: R = L.trivial_representation()
sage: R.lie_algebra() is L
True
```

```
>>> from sage.all import *
>>> L = lie_algebras.sl(QQ, Integer(4))
>>> R = L.trivial_representation()
>>> R.lie_algebra() is L
True
```

`representation_matrix(elt)`

Return the matrix for the action of `elt` on `self`.

EXAMPLES:

```
sage: H1 = lie_algebras.Heisenberg(QQ, 1)
sage: F = H1.faithful_representation(algorithm='minimal')
sage: P1 = F.representation_matrix(H1.gen(0)); P1
[0 0 0]
[0 0 0]
[1 0 0]
sage: Q1 = F.representation_matrix(H1.gen(1)); Q1
[ 0 0 0]
[ 0 0 -1]
[ 0 0 0]
sage: Z = P1.commutator(Q1); Z
[0 0 0]
[1 0 0]
[0 0 0]
sage: P1.commutator(Z) == Q1.commutator(Z) == 0
True
sage: (H1.gen(0) * F.an_element()).to_vector()
(0, 0, 2)
sage: P1 * F.an_element().to_vector()
(0, 0, 2)
sage: (H1.gen(1) * F.an_element()).to_vector()
(0, -3, 0)
sage: Q1 * F.an_element().to_vector()
(0, -3, 0)
sage: (H1.basis()['z'] * F.an_element()).to_vector()
(0, 2, 0)
sage: Z * F.an_element().to_vector()
(0, 2, 0)
```

```

>>> from sage.all import *
>>> H1 = lie_algebras.Heisenberg(QQ, Integer(1))
>>> F = H1.faithful_representation(algorithm='minimal')
>>> P1 = F.representation_matrix(H1.gen(Integer(0))); P1
[0 0 0]
[0 0 0]
[1 0 0]
>>> Q1 = F.representation_matrix(H1.gen(Integer(1))); Q1
[ 0  0  0]
[ 0  0 -1]
[ 0  0  0]
>>> Z = P1.commutator(Q1); Z
[0 0 0]
[1 0 0]
[0 0 0]
>>> P1.commutator(Z) == Q1.commutator(Z) == Integer(0)
True
>>> (H1.gen(Integer(0)) * F.an_element()).to_vector()
(0, 0, 2)
>>> P1 * F.an_element().to_vector()
(0, 0, 2)
>>> (H1.gen(Integer(1)) * F.an_element()).to_vector()
(0, -3, 0)
>>> Q1 * F.an_element().to_vector()
(0, -3, 0)
>>> (H1.basis()['z'] * F.an_element()).to_vector()
(0, 2, 0)
>>> Z * F.an_element().to_vector()
(0, 2, 0)

```

side()

Return that `self` is a left representation.

OUTPUT: the string 'left'

EXAMPLES:

```

sage: L = lie_algebras.sl(QQ, 4)
sage: R = L.trivial_representation()
sage: R.side()
'left'

```

```

>>> from sage.all import *
>>> L = lie_algebras.sl(QQ, Integer(4))
>>> R = L.trivial_representation()
>>> R.side()
'left'

```

class sage.algebras.lie_algebras.representation.TrivialRepresentation(*lie_algebra*, ***kwargs*)

Bases: CombinatorialFreeModule, Representation_abstract

The trivial representation of a Lie algebra.

The trivial representation of a Lie algebra L over a commutative ring R is the 1-dimensional R -module on which every element of L acts by zero.

INPUT:

- `lie_algebra` – a Lie algebra

REFERENCES:

- [Wikipedia article Trivial_representation](#)

class `Element`

Bases: `IndexedFreeModuleElement`

9.2.20 Lie Algebras Given By Structure Coefficients

AUTHORS:

- Travis Scrimshaw (2013-05-03): Initial version

```
class sage.algebras.lie_algebras.structure_coefficients.LieAlgebraWithStructureCoefficients(R,
                                                                                          s_co-
                                                                                          eff,
                                                                                          names,
                                                                                          in-
                                                                                          dex_set,
                                                                                          cat-
                                                                                          e-
                                                                                          gory=None,
                                                                                          pre-
                                                                                          fix=None,
                                                                                          bracket=None,
                                                                                          la-
                                                                                          tex_bracket=None,
                                                                                          string_quote=None,
                                                                                          **kwds)
```

Bases: `FinitelyGeneratedLieAlgebra`, `IndexedGenerators`

A Lie algebra with a set of specified structure coefficients.

The structure coefficients are specified as a dictionary d whose keys are pairs of basis indices, and whose values are dictionaries which in turn are indexed by basis indices. The value of d at a pair (u, v) of basis indices is the dictionary whose w -th entry (for w a basis index) is the coefficient of b_w in the Lie bracket $[b_u, b_v]$ (where b_x means the basis element with index x).

INPUT:

- R – a ring, to be used as the base ring
- `s_coeff` – dictionary, indexed by pairs of basis indices (see below), and whose values are dictionaries which are indexed by (single) basis indices and whose values are elements of R
- `names` – list or tuple of strings
- `index_set` – (default: `names`) list or tuple of hashable and comparable elements

OUTPUT:

A Lie algebra over R which (as an R -module) is free with a basis indexed by the elements of `index_set`. The i -th basis element is displayed using the name `names[i]`. If we let b_i denote this i -th basis element, then the Lie bracket is given by the requirement that the b_k -coefficient of $[b_i, b_j]$ is `s_coeff[(i, j)][k]` if `s_coeff[(i, j)]` exists, otherwise `-s_coeff[(j, i)][k]` if `s_coeff[(j, i)]` exists, otherwise 0.

EXAMPLES:

We create the Lie algebra of $\mathbf{Q}^3$ under the Lie bracket defined by $\times$ (cross-product):

```
sage: L = LieAlgebra(QQ, 'x,y,z', {'(x','y)': {'z':1}, ('y','z)': {'x':1}, ('z','x')
↪': {'y':1}})
sage: (x,y,z) = L.gens()
sage: L.bracket(x, y)
z
sage: L.bracket(y, x)
-z
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, 'x,y,z', {'(x','y)': {'z':Integer(1)}, ('y','z)': {'x':
↪Integer(1)}, ('z','x)': {'y':Integer(1)}})
>>> (x,y,z) = L.gens()
>>> L.bracket(x, y)
z
>>> L.bracket(y, x)
-z
```

class Element

Bases: *StructureCoefficientsElement*

change_ring(*R*)

Return a Lie algebra with identical structure coefficients over *R*.

INPUT:

- *R* – a ring

EXAMPLES:

```
sage: L.<x,y,z> = LieAlgebra(ZZ, {'(x','y)': {'z':1}})
sage: L.structure_coefficients()
Finite family {'(x','y)': z}
sage: LQQ = L.change_ring(QQ)
sage: LQQ.structure_coefficients()
Finite family {'(x','y)': z}
sage: LSR = LQQ.change_ring(SR) #_
↪needs sage.symbolic
sage: LSR.structure_coefficients() #_
↪needs sage.symbolic
Finite family {'(x','y)': z}
```

```
>>> from sage.all import *
>>> L = LieAlgebra(ZZ, {'(x','y)': {'z':Integer(1)}}, names=('x', 'y', 'z'));
↪ (x, y, z) = L._first_ngens(3)
>>> L.structure_coefficients()
Finite family {'(x','y)': z}
>>> LQQ = L.change_ring(QQ)
>>> LQQ.structure_coefficients()
Finite family {'(x','y)': z}
>>> LSR = LQQ.change_ring(SR) #_
↪needs sage.symbolic
>>> LSR.structure_coefficients() #_
```

(continues on next page)

(continued from previous page)

```
↪needs sage.symbolic
Finite family {('x', 'y'): z}
```

dimension()

Return the dimension of self.

EXAMPLES:

```
sage: L = LieAlgebra(QQ, 'x,y', {('x','y'):{'x':1}})
sage: L.dimension()
2
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, 'x,y', {('x','y'):{'x':Integer(1)}})
>>> L.dimension()
2
```

from_vector(v, order=None, coerce=True)

Return an element of self from the vector v.

EXAMPLES:

```
sage: L.<x,y,z> = LieAlgebra(QQ, {('x','y'):{'z':1}})
sage: L.from_vector([1, 2, -2])
x + 2*y - 2*z
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, {('x','y'):{'z':Integer(1)}}, names=('x', 'y', 'z,));
↪ (x, y, z,) = L._first_ngens(3)
>>> L.from_vector([Integer(1), Integer(2), -Integer(2)])
x + 2*y - 2*z
```

module(sparse=True)

Return self as a free module.

EXAMPLES:

```
sage: L.<x,y,z> = LieAlgebra(QQ, {('x','y'):{'z':1}})
sage: L.module()
Sparse vector space of dimension 3 over Rational Field
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, {('x','y'):{'z':Integer(1)}}, names=('x', 'y', 'z,));
↪ (x, y, z,) = L._first_ngens(3)
>>> L.module()
Sparse vector space of dimension 3 over Rational Field
```

monomial(k)

Return the monomial indexed by k.

EXAMPLES:

```
sage: L.<x,y,z> = LieAlgebra(QQ, {'x','y': {'z':1}})
sage: L.monomial('x')
x
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, {'x','y': {'z':Integer(1)}}, names=('x', 'y', 'z'));
↪ (x, y, z) = L._first_ngens(3)
>>> L.monomial('x')
x
```

some_elements()

Return some elements of `self`.

EXAMPLES:

```
sage: L = lie_algebras.three_dimensional(QQ, 4, 1, -1, 2)
sage: L.some_elements()
[X, Y, Z, X + Y + Z]
```

```
>>> from sage.all import *
>>> L = lie_algebras.three_dimensional(QQ, Integer(4), Integer(1), -
↪ Integer(1), Integer(2))
>>> L.some_elements()
[X, Y, Z, X + Y + Z]
```

structure_coefficients (*include_zeros=False*)

Return the dictionary of structure coefficients of `self`.

EXAMPLES:

```
sage: L = LieAlgebra(QQ, 'x,y,z', {'x','y': {'x':1}})
sage: L.structure_coefficients()
Finite family {'x', 'y': x}
sage: S = L.structure_coefficients(True); S
Finite family {'x', 'y': x, ('x', 'z'): 0, ('y', 'z'): 0}
sage: S['x','z'].parent() is L
True
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, 'x,y,z', {'x','y': {'x':Integer(1)}})
>>> L.structure_coefficients()
Finite family {'x', 'y': x}
>>> S = L.structure_coefficients(True); S
Finite family {'x', 'y': x, ('x', 'z'): 0, ('y', 'z'): 0}
>>> S['x','z'].parent() is L
True
```

term (*k, c=None*)

Return the term indexed by `i` with coefficient `c`.

EXAMPLES:

```
sage: L.<x,y,z> = LieAlgebra(QQ, {'x','y': {'z':1}})
sage: L.term('x', 4)
4*x
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, {'x','y': {'z':Integer(1)}}, names=('x', 'y', 'z,));
↪ (x, y, z,) = L._first_ngens(3)
>>> L.term('x', Integer(4))
4*x
```

zero()

Return the element 0 in self.

EXAMPLES:

```
sage: L.<x,y,z> = LieAlgebra(QQ, {'x','y': {'z':1}})
sage: L.zero()
0
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, {'x','y': {'z':Integer(1)}}, names=('x', 'y', 'z,));
↪ (x, y, z,) = L._first_ngens(3)
>>> L.zero()
0
```

9.2.21 Subalgebras and ideals of Lie algebras

AUTHORS:

- Eero Hakavuori (2018-08-29): initial version

```
class sage.algebras.lie_algebras.subalgebra.LieSubalgebra_finite_dimensional_with_basis(ambient,
gens,
ideal,
order=None,
category=None)
```

Bases: `Parent`, `UniqueRepresentation`

A Lie subalgebra of a finite dimensional Lie algebra with basis.

INPUT:

- `ambient` – the Lie algebra containing the subalgebra
- `gens` – list of generators of the subalgebra
- `ideal` – boolean (default: `False`); if `True`, then `gens` is interpreted as the generating set of an ideal instead of a subalgebra
- `order` – (optional) the key used to sort the indices of `ambient`
- `category` – (optional) a subcategory of subobjects of finite dimensional Lie algebras with basis

EXAMPLES:

Subalgebras and ideals are defined by giving a list of generators:

```
sage: L = lie_algebras.Heisenberg(QQ, 1)
sage: X, Y, Z = L.basis()
sage: S = L.subalgebra([X, Z]); S
Subalgebra generated by (p1, z) of Heisenberg algebra of rank 1 over Rational
↪Field
sage: I = L.ideal([X, Z]); I
Ideal (p1, z) of Heisenberg algebra of rank 1 over Rational Field
```

```
>>> from sage.all import *
>>> L = lie_algebras.Heisenberg(QQ, Integer(1))
>>> X, Y, Z = L.basis()
>>> S = L.subalgebra([X, Z]); S
Subalgebra generated by (p1, z) of Heisenberg algebra of rank 1 over Rational
↪Field
>>> I = L.ideal([X, Z]); I
Ideal (p1, z) of Heisenberg algebra of rank 1 over Rational Field
```

An ideal is in general larger than the subalgebra with the same generators:

```
sage: S = L.subalgebra(Y)
sage: S.basis()
Family (q1,)
sage: I = L.ideal(Y)
sage: I.basis()
Family (q1, z)
```

```
>>> from sage.all import *
>>> S = L.subalgebra(Y)
>>> S.basis()
Family (q1,)
>>> I = L.ideal(Y)
>>> I.basis()
Family (q1, z)
```

The zero dimensional subalgebra can be created by giving 0 as a generator or with an empty list of generators:

```
sage: L.<X,Y,Z> = LieAlgebra(QQ, {'X','Y': {'Z': 1}})
sage: S1 = L.subalgebra(0)
sage: S2 = L.subalgebra([])
sage: S1 is S2
True
sage: S1.basis()
Family ()
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, {'X','Y': {'Z': Integer(1)}}, names=('X', 'Y', 'Z'));
↪(X, Y, Z,) = L._first_ngens(3)
>>> S1 = L.subalgebra(Integer(0))
>>> S2 = L.subalgebra([])
```

(continues on next page)

(continued from previous page)

```
>>> S1 is S2
True
>>> S1.basis()
Family ()
```

Elements of the ambient Lie algebra can be reduced modulo an ideal or subalgebra:

```
sage: # needs sage.symbolic
sage: L.<X,Y,Z> = LieAlgebra(SR, {'X','Y': {'Z': 1}})
sage: I = L.ideal(Y)
sage: I.reduce(X + 2*Y + 3*Z)
X
sage: S = L.subalgebra(Y)
sage: S.reduce(X + 2*Y + 3*Z)
X + 3*Z
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> L = LieAlgebra(SR, {'X','Y': {'Z': Integer(1)}}, names=('X', 'Y', 'Z'));
↳ (X, Y, Z) = L._first_ngens(3)
>>> I = L.ideal(Y)
>>> I.reduce(X + Integer(2)*Y + Integer(3)*Z)
X
>>> S = L.subalgebra(Y)
>>> S.reduce(X + Integer(2)*Y + Integer(3)*Z)
X + 3*Z
```

The reduction gives elements in a fixed complementary subspace. When the base ring is a field, the complementary subspace is spanned by those basis elements which are not leading supports of the basis:

```
sage: # needs sage.symbolic
sage: I = L.ideal(X + Y)
sage: I.basis()
Family (X + Y, Z)
sage: el = var('x')*X + var('y')*Y + var('z')*Z; el
x*X + y*Y + z*Z
sage: I.reduce(el)
(x-y)*X
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> I = L.ideal(X + Y)
>>> I.basis()
Family (X + Y, Z)
>>> el = var('x')*X + var('y')*Y + var('z')*Z; el
x*X + y*Y + z*Z
>>> I.reduce(el)
(x-y)*X
```

Giving a different order may change the reduction of elements:

```

sage: I = L.ideal(X + Y, order=lambda s: ['Z', 'Y', 'X'].index(s)) #_
↳needs sage.symbolic
sage: I.basis() #_
↳needs sage.symbolic
Family (Z, X + Y)
sage: I.reduce(e1) #_
↳needs sage.symbolic
(-x+y)*Y

```

```

>>> from sage.all import *
>>> I = L.ideal(X + Y, order=lambda s: ['Z', 'Y', 'X'].index(s)) #_
↳needs sage.symbolic
>>> I.basis() #_
↳needs sage.symbolic
Family (Z, X + Y)
>>> I.reduce(e1) #_
↳needs sage.symbolic
(-x+y)*Y

```

A subalgebra of a subalgebra is a subalgebra of the original:

```

sage: sc = {( 'X', 'Y'): { 'Z': 1}, ( 'X', 'Z'): { 'W': 1}}
sage: L.<X,Y,Z,W> = LieAlgebra(QQ, sc)
sage: S1 = L.subalgebra([Y, Z, W]); S1
Subalgebra generated by (Y, Z, W) of Lie algebra on 4 generators (X, Y, Z, W)
↳over Rational Field
sage: S2 = S1.subalgebra(S1.gens()[1:]); S2
Subalgebra generated by (Z, W) of Lie algebra on 4 generators (X, Y, Z, W) over
↳Rational Field
sage: S3 = S2.subalgebra(S2.gens()[1:]); S3
Subalgebra generated by (W) of Lie algebra on 4 generators (X, Y, Z, W) over
↳Rational Field

```

```

>>> from sage.all import *
>>> sc = {( 'X', 'Y'): { 'Z': Integer(1)}, ( 'X', 'Z'): { 'W': Integer(1)}}
>>> L = LieAlgebra(QQ, sc, names=( 'X', 'Y', 'Z', 'W',)); (X, Y, Z, W,) = L._first_
↳ngens(4)
>>> S1 = L.subalgebra([Y, Z, W]); S1
Subalgebra generated by (Y, Z, W) of Lie algebra on 4 generators (X, Y, Z, W)
↳over Rational Field
>>> S2 = S1.subalgebra(S1.gens()[Integer(1):]); S2
Subalgebra generated by (Z, W) of Lie algebra on 4 generators (X, Y, Z, W) over
↳Rational Field
>>> S3 = S2.subalgebra(S2.gens()[Integer(1):]); S3
Subalgebra generated by (W) of Lie algebra on 4 generators (X, Y, Z, W) over
↳Rational Field

```

An ideal of an ideal is not necessarily an ideal of the original:

```

sage: I = L.ideal(Y); I
Ideal (Y) of Lie algebra on 4 generators (X, Y, Z, W) over Rational Field
sage: J = I.ideal(Z); J
Ideal (Z) of Ideal (Y) of Lie algebra on 4 generators (X, Y, Z, W) over Rational
(continues on next page)

```

(continued from previous page)

```

↪Field
sage: J.basis()
Family (Z,)
sage: J.is_ideal(L)
False
sage: K = L.ideal(J.basis().list())
sage: K.basis()
Family (Z, W)

```

```

>>> from sage.all import *
>>> I = L.ideal(Y); I
Ideal (Y) of Lie algebra on 4 generators (X, Y, Z, W) over Rational Field
>>> J = I.ideal(Z); J
Ideal (Z) of Ideal (Y) of Lie algebra on 4 generators (X, Y, Z, W) over Rational_
↪Field
>>> J.basis()
Family (Z,)
>>> J.is_ideal(L)
False
>>> K = L.ideal(J.basis().list())
>>> K.basis()
Family (Z, W)

```

class ElementBases: *LieSubalgebraElementWrapper***adjoint_matrix**(*sparse=False*)

Return the matrix of the adjoint action of self.

EXAMPLES:

```

sage: MS = MatrixSpace(QQ, 2)
sage: m = MS([[0, -1], [1, 0]])
sage: L = LieAlgebra(associative=MS)
sage: S = L.subalgebra([m])
sage: x = S.basis()[0]
sage: x.parent() is S
True
sage: x.adjoint_matrix()
[0]

sage: m1 = MS([[0, 1], [0, 0]])
sage: m2 = MS([[0, 0], [1, 0]])
sage: S = L.subalgebra([m1, m2])
sage: e, f = S.lie_algebra_generators()
sage: ascii_art([b.value.value for b in S.basis()])
[ [0 1]  [0 0]  [-1 0] ]
[ [0 0], [1 0], [0 1] ]
sage: E = e.adjoint_matrix(); E
[ 0 0 2]
[ 0 0 0]
[ 0 -1 0]

```

(continues on next page)

(continued from previous page)

```

sage: F = f.adjoint_matrix(); F
[ 0  0  0]
[ 0  0 -2]
[ 1  0  0]
sage: h = e.bracket(f)
sage: E * F - F * E == h.adjoint_matrix()
True

```

```

>>> from sage.all import *
>>> MS = MatrixSpace(QQ, Integer(2))
>>> m = MS([[Integer(0), -Integer(1)], [Integer(1), Integer(0)]])
>>> L = LieAlgebra(associative=MS)
>>> S = L.subalgebra([m])
>>> x = S.basis()[Integer(0)]
>>> x.parent() is S
True
>>> x.adjoint_matrix()
[0]

>>> m1 = MS([[Integer(0), Integer(1)], [Integer(0), Integer(0)]])
>>> m2 = MS([[Integer(0), Integer(0)], [Integer(1), Integer(0)]])
>>> S = L.subalgebra([m1, m2])
>>> e, f = S.lie_algebra_generators()
>>> ascii_art([b.value.value for b in S.basis()])
[ [0 1] [0 0] [-1 0] ]
[ [0 0], [1 0], [0 1] ]
>>> E = e.adjoint_matrix(); E
[ 0  0  2]
[ 0  0  0]
[ 0 -1  0]
>>> F = f.adjoint_matrix(); F
[ 0  0  0]
[ 0  0 -2]
[ 1  0  0]
>>> h = e.bracket(f)
>>> E * F - F * E == h.adjoint_matrix()
True

```

ambient()

Return the ambient Lie algebra of self.

EXAMPLES:

```

sage: L.<x,y> = LieAlgebra(QQ, abelian=True)
sage: S = L.subalgebra(x)
sage: S.ambient() is L
True

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, abelian=True, names=('x', 'y')); (x, y) = L._first_
↳ngens(2)
>>> S = L.subalgebra(x)

```

(continues on next page)

(continued from previous page)

```
>>> S.ambient() is L
True
```

basis()

Return a basis of `self`.

EXAMPLES:

A basis of a subalgebra:

```
sage: sc = {( 'a', 'b') : { 'c': 1}, ( 'a', 'c') : { 'd': 1}}
sage: L.<a,b,c,d> = LieAlgebra(QQ, sc)
sage: L.subalgebra([a + b, c + d]).basis()
Family (a + b, c, d)
```

```
>>> from sage.all import *
>>> sc = {( 'a', 'b') : { 'c': Integer(1)}, ( 'a', 'c') : { 'd': Integer(1)}}
>>> L = LieAlgebra(QQ, sc, names=( 'a', 'b', 'c', 'd',)); (a, b, c, d,) = L._
↳first_ngens(4)
>>> L.subalgebra([a + b, c + d]).basis()
Family (a + b, c, d)
```

A basis of an ideal:

```
sage: sc = {( 'x', 'y') : { 'z': 1}, ( 'x', 'z') : { 'w': 1}}
sage: L.<x,y,z,w> = LieAlgebra(QQ, sc)
sage: L.ideal([x + y + z + w]).basis()
Family (x + y, z, w)
```

```
>>> from sage.all import *
>>> sc = {( 'x', 'y') : { 'z': Integer(1)}, ( 'x', 'z') : { 'w': Integer(1)}}
>>> L = LieAlgebra(QQ, sc, names=( 'x', 'y', 'z', 'w',)); (x, y, z, w,) = L._
↳first_ngens(4)
>>> L.ideal([x + y + z + w]).basis()
Family (x + y, z, w)
```

This also works for Lie algebras whose natural basis elements are not comparable (but have a well-defined basis ordering):

```
sage: sl3 = LieAlgebra(QQ, cartan_type=[ 'A', 2])
sage: D = sl3.derived_subalgebra()
sage: len(D.basis())
8
sage: e = list(sl3.e())
sage: sl3.ideal(e).dimension()
8
sage: sl3.subalgebra(e).dimension()
3
```

```
>>> from sage.all import *
>>> sl3 = LieAlgebra(QQ, cartan_type=[ 'A', Integer(2)])
>>> D = sl3.derived_subalgebra()
```

(continues on next page)

(continued from previous page)

```

>>> len(D.basis())
8
>>> e = list(s13.e())
>>> s13.ideal(e).dimension()
8
>>> s13.subalgebra(e).dimension()
3

```

basis_matrix()

Return the basis matrix of `self` as a submodule of the ambient Lie algebra.

EXAMPLES:

```

sage: L.<X,Y,Z> = LieAlgebra(ZZ, {'X','Y': {'Z': 3}})
sage: S1 = L.subalgebra([4*X + Y, Y])
sage: S1.basis_matrix()
[ 4  0  0]
[ 0  1  0]
[ 0  0 12]
sage: K.<X,Y,Z> = LieAlgebra(QQ, {'X','Y': {'Z': 3}})
sage: S2 = K.subalgebra([4*X + Y, Y])
sage: S2.basis_matrix()
[1 0 0]
[0 1 0]
[0 0 1]

```

```

>>> from sage.all import *
>>> L = LieAlgebra(ZZ, {'X','Y': {'Z': Integer(3)}}, names=('X', 'Y', 'Z',
↳ )); (X, Y, Z,) = L._first_ngens(3)
>>> S1 = L.subalgebra([Integer(4)*X + Y, Y])
>>> S1.basis_matrix()
[ 4  0  0]
[ 0  1  0]
[ 0  0 12]
>>> K = LieAlgebra(QQ, {'X','Y': {'Z': Integer(3)}}, names=('X', 'Y', 'Z',
↳ )); (X, Y, Z,) = K._first_ngens(3)
>>> S2 = K.subalgebra([Integer(4)*X + Y, Y])
>>> S2.basis_matrix()
[1 0 0]
[0 1 0]
[0 0 1]

```

from_vector (*v*, *order=None*, *coerce=False*)

Return the element of `self` corresponding to the vector *v*.

INPUT:

- *v* – a vector in `self.module()` or `self.ambient().module()`

EXAMPLES:

An element from a vector of the intrinsic module:

```

sage: L.<X,Y,Z> = LieAlgebra(ZZ, abelian=True)
sage: L.dimension()
3
sage: S = L.subalgebra([X, Y])
sage: S.dimension()
2
sage: el = S.from_vector([1, 2]); el
X + 2*Y
sage: el.parent() == S
True

```

```

>>> from sage.all import *
>>> L = LieAlgebra(ZZ, abelian=True, names=('X', 'Y', 'Z')); (X, Y, Z) = L._
↳first_ngens(3)
>>> L.dimension()
3
>>> S = L.subalgebra([X, Y])
>>> S.dimension()
2
>>> el = S.from_vector([Integer(1), Integer(2)]); el
X + 2*Y
>>> el.parent() == S
True

```

An element from a vector of the ambient module

```
sage: el = S.from_vector([1, 2, 0]); el
X + 2*Y
sage: el.parent() == S
True
```

gens()

Return the generating set of `self`.

EXAMPLES:

```

sage: L.<x,y,z> = LieAlgebra(QQ, {('x','y'): {'z': 1}})
sage: S = L.subalgebra(x)
sage: S.gens()
(x,)

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, {('x','y'): {'z': Integer(1)}}, names=('x', 'y', 'z',
↳)); (x, y, z) = L._first_ngens(3)
>>> S = L.subalgebra(x)
>>> S.gens()
(x,)

```

indices()

Return the set of indices for the basis of `self`.

EXAMPLES:

```

sage: L.<x,y,z> = LieAlgebra(QQ, abelian=True)
sage: S = L.subalgebra([x, y])
sage: S.indices()
{0, 1}

```

(continues on next page)

(continued from previous page)

```
sage: [S.basis()[k] for k in S.indices()]
[x, y]
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, abelian=True, names=('x', 'y', 'z')); (x, y, z) = L._
↳first_ngens(3)
>>> S = L.subalgebra([x, y])
>>> S.indices()
{0, 1}
>>> [S.basis()[k] for k in S.indices()]
[x, y]
```

is_ideal(A)

Return if self is an ideal of A.

EXAMPLES:

Some subalgebras are ideals:

```
sage: L.<x,y,z> = LieAlgebra(QQ, {('x','y'): {'z': 1}})
sage: S1 = L.subalgebra([x])
sage: S1.is_ideal(L)
False
sage: S2 = L.subalgebra([x, y])
sage: S2.is_ideal(L)
True
sage: S3 = L.subalgebra([y, z])
sage: S3.is_ideal(L)
True
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, {('x','y'): {'z': Integer(1)}}, names=('x', 'y', 'z',
↳)); (x, y, z,) = L._first_ngens(3)
>>> S1 = L.subalgebra([x])
>>> S1.is_ideal(L)
False
>>> S2 = L.subalgebra([x, y])
>>> S2.is_ideal(L)
True
>>> S3 = L.subalgebra([y, z])
>>> S3.is_ideal(L)
True
```

All ideals are ideals:

```
sage: L.<x,y> = LieAlgebra(QQ, {('x','y'): {'x': 1}})
sage: I = L.ideal(x)
sage: I.is_ideal(L)
True
sage: I.is_ideal(I)
True
```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, {('x','y'): {'x': Integer(1)}}, names=('x', 'y')); (x,
↪ y,) = L._first_ngens(2)
>>> I = L.ideal(x)
>>> I.is_ideal(L)
True
>>> I.is_ideal(I)
True

```

leading_monomials()

Return the set of leading monomials of the basis of `self`.

EXAMPLES:

A basis of an ideal and the corresponding leading monomials:

```

sage: sc = {('a','b'): {'c': 2}, ('a','c'): {'d': 4}}
sage: L.<a,b,c,d> = LieAlgebra(ZZ, sc)
sage: I = L.ideal(a + b)
sage: I.basis()
Family (a + b, 2*c, 4*d)
sage: I.leading_monomials()
Family (b, c, d)

```

```

>>> from sage.all import *
>>> sc = {('a','b'): {'c': Integer(2)}, ('a','c'): {'d': Integer(4)}}
>>> L = LieAlgebra(ZZ, sc, names=('a', 'b', 'c', 'd')); (a, b, c, d,) = L._
↪first_ngens(4)
>>> I = L.ideal(a + b)
>>> I.basis()
Family (a + b, 2*c, 4*d)
>>> I.leading_monomials()
Family (b, c, d)

```

A different ordering can give different leading monomials:

```

sage: key = lambda s: ['d','c','b','a'].index(s)
sage: I = L.ideal(a + b, order=key)
sage: I.basis()
Family (4*d, 2*c, a + b)
sage: I.leading_monomials()
Family (d, c, a)

```

```

>>> from sage.all import *
>>> key = lambda s: ['d','c','b','a'].index(s)
>>> I = L.ideal(a + b, order=key)
>>> I.basis()
Family (4*d, 2*c, a + b)
>>> I.leading_monomials()
Family (d, c, a)

```

lie_algebra_generators()

Return the generating set of `self` as a Lie algebra.

EXAMPLES:

The Lie algebra generators of a subalgebra are the original generators:

```
sage: L.<x,y,z> = LieAlgebra(QQ, {'x','y': {'z': 1}})
sage: S = L.subalgebra(x)
sage: S.lie_algebra_generators()
(x,)
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, {'x','y': {'z': Integer(1)}}, names=('x', 'y', 'z',
↳)); (x, y, z,) = L._first_ngens(3)
>>> S = L.subalgebra(x)
>>> S.lie_algebra_generators()
(x,)
```

The Lie algebra generators of an ideal is usually a larger set:

```
sage: I = L.ideal(x)
sage: I.lie_algebra_generators()
Family (x, z)
```

```
>>> from sage.all import *
>>> I = L.ideal(x)
>>> I.lie_algebra_generators()
Family (x, z)
```

lift(X)

Coerce an element x of `self` into the ambient Lie algebra.

INPUT:

- x – an element of `self`

EXAMPLES:

```
sage: L.<x,y> = LieAlgebra(QQ, abelian=True)
sage: S = L.subalgebra(x)
sage: sx = S(x); sx
x
sage: sx.parent()
Subalgebra generated by (x) of Abelian Lie algebra on 2 generators (x, y)
↳over Rational Field
sage: a = S.lift(sx); a
x
sage: a.parent()
Abelian Lie algebra on 2 generators (x, y) over Rational Field
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, abelian=True, names=('x', 'y')); (x, y,) = L._first_
↳ngens(2)
>>> S = L.subalgebra(x)
>>> sx = S(x); sx
x
>>> sx.parent()
```

(continues on next page)

(continued from previous page)

```

Subalgebra generated by (x) of Abelian Lie algebra on 2 generators (x, y)
↳ Over Rational Field
>>> a = S.lift(sx); a
x
>>> a.parent()
Abelian Lie algebra on 2 generators (x, y) over Rational Field

```

module (*sparse=False*)Return the submodule of the ambient Lie algebra corresponding to *self*.

EXAMPLES:

```

sage: L.<X,Y,Z> = LieAlgebra(ZZ, {'X','Y': {'Z': 3}})
sage: S = L.subalgebra([X, Y])
sage: S.module()
Free module of degree 3 and rank 3 over Integer Ring
User basis matrix:
[1 0 0]
[0 1 0]
[0 0 3]

```

```

>>> from sage.all import *
>>> L = LieAlgebra(ZZ, {'X','Y': {'Z': Integer(3)}}, names=('X', 'Y', 'Z',
↳ )); (X, Y, Z,) = L._first_ngens(3)
>>> S = L.subalgebra([X, Y])
>>> S.module()
Free module of degree 3 and rank 3 over Integer Ring
User basis matrix:
[1 0 0]
[0 1 0]
[0 0 3]

```

retract (*X*)Retract *X* to *self*.

INPUT:

- *X* – an element of the ambient Lie algebra

EXAMPLES:

Retraction to a subalgebra of a free nilpotent Lie algebra:

```

sage: L = LieAlgebra(QQ, 3, step=2)
sage: L.inject_variables()
Defining X_1, X_2, X_3, X_12, X_13, X_23
sage: S = L.subalgebra([X_1, X_2])
sage: e1 = S.retract(2*X_1 + 3*X_2 + 5*X_12); e1
2*X_1 + 3*X_2 + 5*X_12
sage: e1.parent()
Subalgebra generated by (X_1, X_2) of Free Nilpotent Lie algebra on
6 generators (X_1, X_2, X_3, X_12, X_13, X_23) over Rational Field

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, Integer(3), step=Integer(2))
>>> L.inject_variables()
Defining X_1, X_2, X_3, X_12, X_13, X_23
>>> S = L.subalgebra([X_1, X_2])
>>> e1 = S.retract(Integer(2)*X_1 + Integer(3)*X_2 + Integer(5)*X_12); e1
2*X_1 + 3*X_2 + 5*X_12
>>> e1.parent()
Subalgebra generated by (X_1, X_2) of Free Nilpotent Lie algebra on
6 generators (X_1, X_2, X_3, X_12, X_13, X_23) over Rational Field

```

Retraction raises an error if the element is not contained in the subalgebra:

```

sage: S.retract(X_3)
Traceback (most recent call last):
...
ValueError: the element X_3 is not in Subalgebra generated
by (X_1, X_2) of Free Nilpotent Lie algebra on 6 generators
(X_1, X_2, X_3, X_12, X_13, X_23) over Rational Field

```

```

>>> from sage.all import *
>>> S.retract(X_3)
Traceback (most recent call last):
...
ValueError: the element X_3 is not in Subalgebra generated
by (X_1, X_2) of Free Nilpotent Lie algebra on 6 generators
(X_1, X_2, X_3, X_12, X_13, X_23) over Rational Field

```

zero()

Return the element 0.

EXAMPLES:

```

sage: L.<x,y> = LieAlgebra(QQ, abelian=True)
sage: S = L.subalgebra(x)
sage: S.zero()
0
sage: S.zero() == S(L.zero())
True

```

```

>>> from sage.all import *
>>> L = LieAlgebra(QQ, abelian=True, names=('x', 'y')); (x, y) = L._first_
↳ngens(2)
>>> S = L.subalgebra(x)
>>> S.zero()
0
>>> S.zero() == S(L.zero())
True

```

9.2.22 Symplectic Derivation Lie Algebras

AUTHORS:

- Travis Scrimshaw (2020-10): Initial version

class sage.algebras.lie_algebras.symplectic_derivation.SymplecticDerivationLieAlgebra(*R*,
g)

Bases: *InfinitelyGeneratedLieAlgebra*, *IndexedGenerators*

The symplectic derivation Lie algebra.

Fix a $g \geq 4$ and let R be a commutative ring. Let $H = R^{2g}$ be equipped with a symplectic form μ with the basis $a_1, \dots, a_g, b_1, \dots, b_g$ such that

$$\mu(a_i, a_j) = \mu(b_i, b_j) = 0, \quad \mu(a_i, b_j) = -\mu(b_j, a_i) = \delta_{ij},$$

for all i, j . The *symplectic derivation Lie algebra* is the Lie algebra

$$\mathfrak{c}_g := \bigoplus_{w \geq 0} S^{w+2} H$$

with the Lie bracket on basis elements

$$[x_1 \cdots x_{m+2}, y_1 \cdots y_{n+2}] = \sum_{i,j} \mu(x_i, y_j) x_1 \cdots \widehat{x}_i \cdots x_{m+2} \cdot y_1 \cdots \widehat{y}_j \cdots y_{n+2},$$

where $\widehat{}$ denotes that factor is missing. When $R = \mathbf{Q}$, this corresponds to the classical Poisson bracket on $C^\infty(\mathbf{R}^{2g})$ restricted to polynomials with coefficients in $\mathbf{Q}$.

EXAMPLES:

```
sage: L = lie_algebras.SymplecticDerivation(QQ, 5)
sage: elts = L.some_elements()
sage: list(elts)
[a1*a2, b1*b3, a1*a1*a2, b3*b4,
 a1*a4*b3, a1*a2 - 1/2*a1*a2*a2*a5 + a1*a1*a2*b1*b4]
sage: [[elts[i].bracket(elts[j]) for i in range(len(elts))]
....:  for j in range(len(elts))]
[[0, -a2*b3, 0, 0, 0, -a1*a1*a2*a2*b4],
 [a2*b3, 0, 2*a1*a2*b3, 0, a4*b3*b3, a2*b3 - 1/2*a2*a2*a5*b3 + 2*a1*a2*b1*b3*b4],
 [0, -2*a1*a2*b3, 0, 0, 0, -2*a1*a1*a1*a2*a2*b4],
 [0, 0, 0, 0, a1*b3*b3, 0],
 [0, -a4*b3*b3, 0, -a1*b3*b3, 0, -a1*a1*a1*a2*b1*b3 - a1*a1*a2*a4*b3*b4],
 [a1*a1*a2*a2*b4, -a2*b3 + 1/2*a2*a2*a5*b3 - 2*a1*a2*b1*b3*b4,
 ↪ 2*a1*a1*a1*a2*a2*b4,
 0, a1*a1*a1*a2*b1*b3 + a1*a1*a2*a4*b3*b4, 0]]
sage: x = L.monomial(Partition([8,8,6,6,4,2,2,1,1,1])); x
a1*a1*a1*a2*a2*a4*b1*b1*b3*b3
sage: [L[x, elt] for elt in elts]
[-2*a1*a1*a1*a2*a2*a4*b1*b3*b3,
 3*a1*a1*a2*a2*a4*b1*b1*b3*b3*b3,
 -4*a1*a1*a1*a1*a2*a2*a2*a4*b1*b3*b3,
 a1*a1*a1*a2*a2*b1*b1*b3*b3*b3,
 -2*a1*a1*a1*a2*a2*a4*b1*b3*b3*b3,
 -2*a1*a1*a1*a2*a2*a2*a4*b1*b3*b3 + a1*a1*a1*a2*a2*a2*a4*a5*b1*b3*b3
 + a1*a1*a1*a1*a2*a2*a2*b1*b1*b1*b3*b3 ↪
 ↪ a1*a1*a1*a1*a2*a2*a2*a4*b1*b1*b3*b3*b4]
```

```

>>> from sage.all import *
>>> L = lie_algebras.SymplecticDerivation(QQ, Integer(5))
>>> elts = L.some_elements()
>>> list(elts)
[a1*a2, b1*b3, a1*a1*a2, b3*b4,
 a1*a4*b3, a1*a2 - 1/2*a1*a2*a2*a5 + a1*a1*a2*b1*b4]
>>> [[elts[i].bracket(elts[j]) for i in range(len(elts))]
...   for j in range(len(elts))]
[[0, -a2*b3, 0, 0, 0, -a1*a1*a2*a2*b4],
 [a2*b3, 0, 2*a1*a2*b3, 0, a4*b3*b3, a2*b3 - 1/2*a2*a2*a5*b3 + 2*a1*a2*b1*b3*b4],
 [0, -2*a1*a2*b3, 0, 0, 0, -2*a1*a1*a1*a2*a2*b4],
 [0, 0, 0, 0, a1*b3*b3, 0],
 [0, -a4*b3*b3, 0, -a1*b3*b3, 0, -a1*a1*a1*a2*b1*b3 - a1*a1*a2*a4*b3*b4],
 [a1*a1*a2*a2*b4, -a2*b3 + 1/2*a2*a2*a5*b3 - 2*a1*a2*b1*b3*b4,
 ↪ 2*a1*a1*a1*a2*a2*b4,
 0, a1*a1*a1*a2*b1*b3 + a1*a1*a2*a4*b3*b4, 0]]
>>> x = L.monomial(Partition([Integer(8), Integer(8), Integer(6), Integer(6),
 ↪ Integer(4), Integer(2), Integer(2), Integer(1), Integer(1), Integer(1)])); x
a1*a1*a1*a2*a2*a4*b1*b1*b3*b3
>>> [L[x, elt] for elt in elts]
[-2*a1*a1*a1*a2*a2*a2*a4*b1*b3*b3,
 3*a1*a1*a2*a2*a4*b1*b1*b3*b3*b3,
 -4*a1*a1*a1*a1*a2*a2*a2*a4*b1*b3*b3,
 a1*a1*a1*a2*a2*b1*b1*b3*b3*b3,
 -2*a1*a1*a1*a2*a2*a4*a4*b1*b3*b3*b3,
 -2*a1*a1*a1*a2*a2*a2*a4*b1*b3*b3 + a1*a1*a1*a2*a2*a2*a4*a5*b1*b3*b3
 + a1*a1*a1*a1*a1*a2*a2*a2*b1*b1*b3*b3 ↪
 ↪ a1*a1*a1*a1*a2*a2*a2*a4*b1*b1*b3*b3*b4]

```

REFERENCES:

- [Harako2020]

class Element

Bases: [LieAlgebraElement](#)

bracket_on_basis(*x*, *y*)

Return the bracket of basis elements indexed by *x* and *y*, where *i* < *j*.

EXAMPLES:

```

sage: L = lie_algebras.SymplecticDerivation(QQ, 5)
sage: L.bracket_on_basis([5,2,1], [5,1,1])
0
sage: L.bracket_on_basis([6,1], [3,1,1])
-2*a1*a1*a3
sage: L.bracket_on_basis([9,2,1], [4,1,1])
-a1*a1*a1*a2
sage: L.bracket_on_basis([5,5,2], [6,1,1])
0
sage: L.bracket_on_basis([5,5,5], [10,3])
3*a3*a5*a5
sage: L.bracket_on_basis([10,10,10], [5,3])
-3*a3*b5*b5

```

```

>>> from sage.all import *
>>> L = lie_algebras.SymplecticDerivation(QQ, Integer(5))
>>> L.bracket_on_basis([Integer(5), Integer(2), Integer(1)], [Integer(5),
↪ Integer(1), Integer(1)])
0
>>> L.bracket_on_basis([Integer(6), Integer(1)], [Integer(3), Integer(1),
↪ Integer(1)])
-2*a1*a1*a3
>>> L.bracket_on_basis([Integer(9), Integer(2), Integer(1)], [Integer(4),
↪ Integer(1), Integer(1)])
-a1*a1*a1*a2
>>> L.bracket_on_basis([Integer(5), Integer(5), Integer(2)], [Integer(6),
↪ Integer(1), Integer(1)])
0
>>> L.bracket_on_basis([Integer(5), Integer(5), Integer(5)], [Integer(10),
↪ Integer(3)])
3*a3*a5*a5
>>> L.bracket_on_basis([Integer(10), Integer(10), Integer(10)], [Integer(5),
↪ Integer(3)])
-3*a3*b5*b5

```

degree_on_basis(x)

Return the degree of the basis element indexed by x.

EXAMPLES:

```

sage: L = lie_algebras.SymplecticDerivation(QQ, 5)
sage: L.degree_on_basis([5, 2, 1])
1
sage: L.degree_on_basis([1, 1])
0
sage: elt = L.monomial(Partition([5, 5, 2, 1])) + 3*L.monomial(Partition([3, 3, 2,
↪ 1]))
sage: elt.degree()
2

```

```

>>> from sage.all import *
>>> L = lie_algebras.SymplecticDerivation(QQ, Integer(5))
>>> L.degree_on_basis([Integer(5), Integer(2), Integer(1)])
1
>>> L.degree_on_basis([Integer(1), Integer(1)])
0
>>> elt = L.monomial(Partition([Integer(5), Integer(5), Integer(2),
↪ Integer(1)])) + Integer(3)*L.monomial(Partition([Integer(3), Integer(3),
↪ Integer(2), Integer(1)]))
>>> elt.degree()
2

```

some_elements()

Return some elements of self.

EXAMPLES:


```
sage: L = lie_algebras.SymplecticDerivation(QQ, 5)
sage: L.some_elements()
[a1*a2, b1*b3, a1*a1*a2, b3*b4, a1*a4*b3,
 a1*a2 - 1/2*a1*a2*a2*a5 + a1*a1*a2*b1*b4]
```

```
>>> from sage.all import *
>>> L = lie_algebras.SymplecticDerivation(QQ, Integer(5))
>>> L.some_elements()
[a1*a2, b1*b3, a1*a1*a2, b3*b4, a1*a4*b3,
 a1*a2 - 1/2*a1*a2*a2*a5 + a1*a1*a2*b1*b4]
```

9.2.23 Verma Modules

AUTHORS:

- Travis Scrimshaw (2017-06-30): Initial version

Todo

Implement a `sage.categories.pushout.ConstructionFunctor` and return as the `construction()`.

```
class sage.algebras.lie_algebras.verma_module.ModulePrinting(vector_name='v')
```

Bases: object

Helper mixin class for printing the module vectors.

```
class sage.algebras.lie_algebras.verma_module.VermaModule(g, weight, basis_key=None, prefix='f',
                                                         **kws)
```

Bases: `ModulePrinting`, `CombinatorialFreeModule`

A Verma module.

Let λ be a weight and $\mathfrak{g}$ be a Kac–Moody Lie algebra with a fixed Borel subalgebra $\mathfrak{b} = \mathfrak{h} \oplus \mathfrak{g}^+$. The *Verma module* M_λ is a $U(\mathfrak{g})$ -module given by

$$M_\lambda := U(\mathfrak{g}) \otimes_{U(\mathfrak{b})} F_\lambda,$$

where F_λ is the $U(\mathfrak{b})$ module such that $h \in U(\mathfrak{h})$ acts as multiplication by $\langle \lambda, h \rangle$ and $U(\mathfrak{g}^+)F_\lambda = 0$.

INPUT:

- $\mathfrak{g}$ – a Lie algebra
- weight – a weight

EXAMPLES:

```
sage: L = lie_algebras.sl(QQ, 3)
sage: La = L.cartan_type().root_system().weight_lattice().fundamental_weights()
sage: M = L.verma_module(2*La[1] + 3*La[2])
sage: pbw = M.pbw_basis()
sage: E1,E2,F1,F2,H1,H2 = [pbw(g) for g in L.gens()]
sage: v = M.highest_weight_vector()
sage: x = F2^3 * F1 * v
sage: x
```

(continues on next page)

(continued from previous page)

```

f[-alpha[2]]^3*f[-alpha[1]]*v[2*Lambda[1] + 3*Lambda[2]]
sage: F1 * x
f[-alpha[2]]^3*f[-alpha[1]]^2*v[2*Lambda[1] + 3*Lambda[2]]
+ 3*f[-alpha[2]]^2*f[-alpha[1]]*f[-alpha[1] - alpha[2]]*v[2*Lambda[1] + 3*Lambda[2]]
↪ 3*Lambda[2]]
sage: E1 * x
2*f[-alpha[2]]^3*v[2*Lambda[1] + 3*Lambda[2]]
sage: H1 * x
3*f[-alpha[2]]^3*f[-alpha[1]]*v[2*Lambda[1] + 3*Lambda[2]]
sage: H2 * x
-2*f[-alpha[2]]^3*f[-alpha[1]]*v[2*Lambda[1] + 3*Lambda[2]]

```

```

>>> from sage.all import *
>>> L = lie_algebras.sl(QQ, Integer(3))
>>> La = L.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> M = L.verma_module(Integer(2)*La[Integer(1)] + Integer(3)*La[Integer(2)])
>>> pbw = M.pbw_basis()
>>> E1, E2, F1, F2, H1, H2 = [pbw(g) for g in L.gens()]
>>> v = M.highest_weight_vector()
>>> x = F2**Integer(3) * F1 * v
>>> x
f[-alpha[2]]^3*f[-alpha[1]]*v[2*Lambda[1] + 3*Lambda[2]]
>>> F1 * x
f[-alpha[2]]^3*f[-alpha[1]]^2*v[2*Lambda[1] + 3*Lambda[2]]
+ 3*f[-alpha[2]]^2*f[-alpha[1]]*f[-alpha[1] - alpha[2]]*v[2*Lambda[1] + 3*Lambda[2]]
↪ 3*Lambda[2]]
>>> E1 * x
2*f[-alpha[2]]^3*v[2*Lambda[1] + 3*Lambda[2]]
>>> H1 * x
3*f[-alpha[2]]^3*f[-alpha[1]]*v[2*Lambda[1] + 3*Lambda[2]]
>>> H2 * x
-2*f[-alpha[2]]^3*f[-alpha[1]]*v[2*Lambda[1] + 3*Lambda[2]]

```

REFERENCES:

- [Wikipedia article Verma_module](#)

class ElementBases: `IndexedFreeModuleElement`**contravariant_form**(*x*, *y*)Return the contravariant form of *x* and *y*.

Let $C(x, y)$ denote the (universal) contravariant form on $U(\mathfrak{g})$. Then the contravariant form on $M(\lambda)$ is given by evaluating $C(x, y) \in U(\mathfrak{h})$ at λ .

EXAMPLES:

```

sage: g = LieAlgebra(QQ, cartan_type=['A', 1])
sage: La = g.cartan_type().root_system().weight_lattice().fundamental_
↪ weights()
sage: M = g.verma_module(2*La[1])
sage: U = M.pbw_basis()
sage: v = M.highest_weight_vector()

```

(continues on next page)

(continued from previous page)

```

sage: e, h, f = U.algebra_generators()
sage: elts = [f^k * v for k in range(8)]; elts
[v[2*Lambda[1]], f[-alpha[1]]*v[2*Lambda[1]],
 f[-alpha[1]]^2*v[2*Lambda[1]], f[-alpha[1]]^3*v[2*Lambda[1]],
 f[-alpha[1]]^4*v[2*Lambda[1]], f[-alpha[1]]^5*v[2*Lambda[1]],
 f[-alpha[1]]^6*v[2*Lambda[1]], f[-alpha[1]]^7*v[2*Lambda[1]]]
sage: matrix([M.contravariant_form(x, y) for x in elts] for y in elts])
[1 0 0 0 0 0 0 0]
[0 2 0 0 0 0 0 0]
[0 0 4 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]

```

```

>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['A', Integer(1)])
>>> La = g.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> M = g.verma_module(Integer(2)*La[Integer(1)])
>>> U = M.pbw_basis()
>>> v = M.highest_weight_vector()
>>> e, h, f = U.algebra_generators()
>>> elts = [f**k * v for k in range(Integer(8))]; elts
[v[2*Lambda[1]], f[-alpha[1]]*v[2*Lambda[1]],
 f[-alpha[1]]^2*v[2*Lambda[1]], f[-alpha[1]]^3*v[2*Lambda[1]],
 f[-alpha[1]]^4*v[2*Lambda[1]], f[-alpha[1]]^5*v[2*Lambda[1]],
 f[-alpha[1]]^6*v[2*Lambda[1]], f[-alpha[1]]^7*v[2*Lambda[1]]]
>>> matrix([M.contravariant_form(x, y) for x in elts] for y in elts])
[1 0 0 0 0 0 0 0]
[0 2 0 0 0 0 0 0]
[0 0 4 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0]

```

degree_on_basis(m)

Return the degree (or weight) of the basis element indexed by m.

EXAMPLES:

```

sage: L = lie_algebras.sl(QQ, 3)
sage: La = L.cartan_type().root_system().weight_lattice().fundamental_
->weights()
sage: M = L.verma_module(2*La[1] + 3*La[2])
sage: v = M.highest_weight_vector()
sage: M.degree_on_basis(v.leading_support())
2*Lambda[1] + 3*Lambda[2]
sage: pbw = M.pbw_basis()

```

(continues on next page)

(continued from previous page)

```
sage: G = list(pbw.gens())
sage: f1, f2 = L.f()
sage: x = pbw(f1.bracket(f2)) * pbw(f1) * v
sage: x.degree()
-Lambda[1] + 3*Lambda[2]
```

```
>>> from sage.all import *
>>> L = lie_algebras.sl(QQ, Integer(3))
>>> La = L.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> M = L.verma_module(Integer(2)*La[Integer(1)] + Integer(3)*La[Integer(2)])
>>> v = M.highest_weight_vector()
>>> M.degree_on_basis(v.leading_support())
2*Lambda[1] + 3*Lambda[2]

>>> pbw = M.pbw_basis()
>>> G = list(pbw.gens())
>>> f1, f2 = L.f()
>>> x = pbw(f1.bracket(f2)) * pbw(f1) * v
>>> x.degree()
-Lambda[1] + 3*Lambda[2]
```

dual()Return the dual module $M(\lambda)^\vee$ in Category $\mathcal{O}$.

EXAMPLES:

```
sage: L = lie_algebras.sl(QQ, 2)
sage: La = L.cartan_type().root_system().weight_space().fundamental_weights()
sage: M = L.verma_module(2*La[1])
sage: Mc = M.dual()

sage: Mp = L.verma_module(-2*La[1])
sage: Mp.dual() is Mp
True
```

```
>>> from sage.all import *
>>> L = lie_algebras.sl(QQ, Integer(2))
>>> La = L.cartan_type().root_system().weight_space().fundamental_weights()
>>> M = L.verma_module(Integer(2)*La[Integer(1)])
>>> Mc = M.dual()

>>> Mp = L.verma_module(-Integer(2)*La[Integer(1)])
>>> Mp.dual() is Mp
True
```

gens()Return the generators of self as a $U(\mathfrak{g})$ -module.

EXAMPLES:

```
sage: L = lie_algebras.sp(QQ, 6)
sage: La = L.cartan_type().root_system().weight_lattice().fundamental_
```

(continues on next page)

(continued from previous page)

```

↪weights()
sage: M = L.verma_module(La[1] - 3*La[2])
sage: M.gens()
(v[Lambda[1] - 3*Lambda[2]],)

```

```

>>> from sage.all import *
>>> L = lie_algebras.sp(QQ, Integer(6))
>>> La = L.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> M = L.verma_module(La[Integer(1)] - Integer(3)*La[Integer(2)])
>>> M.gens()
(v[Lambda[1] - 3*Lambda[2]],)

```

highest_weight()

Return the highest weight of self.

EXAMPLES:

```

sage: L = lie_algebras.so(QQ, 7)
sage: La = L.cartan_type().root_system().weight_space().fundamental_weights()
sage: M = L.verma_module(4*La[1] - 3/2*La[2])
sage: M.highest_weight()
4*Lambda[1] - 3/2*Lambda[2]

```

```

>>> from sage.all import *
>>> L = lie_algebras.so(QQ, Integer(7))
>>> La = L.cartan_type().root_system().weight_space().fundamental_weights()
>>> M = L.verma_module(Integer(4)*La[Integer(1)] - Integer(3)/
↪Integer(2)*La[Integer(2)])
>>> M.highest_weight()
4*Lambda[1] - 3/2*Lambda[2]

```

highest_weight_vector()

Return the highest weight vector of self.

EXAMPLES:

```

sage: L = lie_algebras.sp(QQ, 6)
sage: La = L.cartan_type().root_system().weight_lattice().fundamental_
↪weights()
sage: M = L.verma_module(La[1] - 3*La[2])
sage: M.highest_weight_vector()
v[Lambda[1] - 3*Lambda[2]]

```

```

>>> from sage.all import *
>>> L = lie_algebras.sp(QQ, Integer(6))
>>> La = L.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> M = L.verma_module(La[Integer(1)] - Integer(3)*La[Integer(2)])
>>> M.highest_weight_vector()
v[Lambda[1] - 3*Lambda[2]]

```

homogeneous_component_basis(d)

Return a basis for the d-th homogeneous component of self.

EXAMPLES:

```

sage: L = lie_algebras.sl(QQ, 3)
sage: P = L.cartan_type().root_system().weight_lattice()
sage: La = P.fundamental_weights()
sage: al = P.simple_roots()
sage: mu = 2*La[1] + 3*La[2]
sage: M = L.verma_module(mu)
sage: M.homogeneous_component_basis(mu - al[2])
[f[-alpha[2]]*v[2*Lambda[1] + 3*Lambda[2]]]
sage: M.homogeneous_component_basis(mu - 3*al[2])
[f[-alpha[2]]^3*v[2*Lambda[1] + 3*Lambda[2]]]
sage: M.homogeneous_component_basis(mu - 3*al[2] - 2*al[1])
[f[-alpha[2]]*f[-alpha[1] - alpha[2]]^2*v[2*Lambda[1] + 3*Lambda[2]],
 f[-alpha[2]]^2*f[-alpha[1]]*f[-alpha[1] - alpha[2]]*v[2*Lambda[1] +
 ↪ 3*Lambda[2]],
 f[-alpha[2]]^3*f[-alpha[1]]^2*v[2*Lambda[1] + 3*Lambda[2]]]
sage: M.homogeneous_component_basis(mu - La[1])
Family ()

```

```

>>> from sage.all import *
>>> L = lie_algebras.sl(QQ, Integer(3))
>>> P = L.cartan_type().root_system().weight_lattice()
>>> La = P.fundamental_weights()
>>> al = P.simple_roots()
>>> mu = Integer(2)*La[Integer(1)] + Integer(3)*La[Integer(2)]
>>> M = L.verma_module(mu)
>>> M.homogeneous_component_basis(mu - al[Integer(2)])
[f[-alpha[2]]*v[2*Lambda[1] + 3*Lambda[2]]]
>>> M.homogeneous_component_basis(mu - Integer(3)*al[Integer(2)])
[f[-alpha[2]]^3*v[2*Lambda[1] + 3*Lambda[2]]]
>>> M.homogeneous_component_basis(mu - Integer(3)*al[Integer(2)] -
 ↪ Integer(2)*al[Integer(1)])
[f[-alpha[2]]*f[-alpha[1] - alpha[2]]^2*v[2*Lambda[1] + 3*Lambda[2]],
 f[-alpha[2]]^2*f[-alpha[1]]*f[-alpha[1] - alpha[2]]*v[2*Lambda[1] +
 ↪ 3*Lambda[2]],
 f[-alpha[2]]^3*f[-alpha[1]]^2*v[2*Lambda[1] + 3*Lambda[2]]]
>>> M.homogeneous_component_basis(mu - La[Integer(1)])
Family ()

```

is_projective()Return if `self` is a projective module in Category $\mathcal{O}$.A Verma module M_λ is projective (in Category $\mathcal{O}$ if and only if λ is *Verma dominant* in the sense

$$\langle \lambda + \rho, \alpha^\vee \rangle \notin \mathbf{Z}_{<0}$$

for all positive roots α .

EXAMPLES:

```

sage: L = lie_algebras.sl(QQ, 3)
sage: La = L.cartan_type().root_system().weight_space().fundamental_weights()
sage: L.verma_module(La[1] + La[2]).is_projective()

```

(continues on next page)

(continued from previous page)

```

True
sage: L.verma_module(-La[1] - La[2]).is_projective()
True
sage: L.verma_module(3/2*La[1] + 1/2*La[2]).is_projective()
True
sage: L.verma_module(3/2*La[1] + 1/3*La[2]).is_projective()
True
sage: L.verma_module(-3*La[1] + 2/3*La[2]).is_projective()
False

```

```

>>> from sage.all import *
>>> L = lie_algebras.sl(QQ, Integer(3))
>>> La = L.cartan_type().root_system().weight_space().fundamental_weights()
>>> L.verma_module(La[Integer(1)] + La[Integer(2)]).is_projective()
True
>>> L.verma_module(-La[Integer(1)] - La[Integer(2)]).is_projective()
True
>>> L.verma_module(Integer(3)/Integer(2)*La[Integer(1)] + Integer(1)/
↳ Integer(2)*La[Integer(2)]).is_projective()
True
>>> L.verma_module(Integer(3)/Integer(2)*La[Integer(1)] + Integer(1)/
↳ Integer(3)*La[Integer(2)]).is_projective()
True
>>> L.verma_module(-Integer(3)*La[Integer(1)] + Integer(2)/
↳ Integer(3)*La[Integer(2)]).is_projective()
False

```

is_simple()

Return if self is a simple module.

A Verma module M_λ is simple if and only if λ is *Verma antidominant* in the sense

$$\langle \lambda + \rho, \alpha^\vee \rangle \notin \mathbf{Z}_{>0}$$

for all positive roots α .

EXAMPLES:

```

sage: L = lie_algebras.sl(QQ, 3)
sage: La = L.cartan_type().root_system().weight_space().fundamental_weights()
sage: L.verma_module(La[1] + La[2]).is_simple()
False
sage: L.verma_module(-La[1] - La[2]).is_simple()
True
sage: L.verma_module(3/2*La[1] + 1/2*La[2]).is_simple()
False
sage: L.verma_module(3/2*La[1] + 1/3*La[2]).is_simple()
True
sage: L.verma_module(-3*La[1] + 2/3*La[2]).is_simple()
True

```

```

>>> from sage.all import *
>>> L = lie_algebras.sl(QQ, Integer(3))

```

(continues on next page)

(continued from previous page)

```

>>> La = L.cartan_type().root_system().weight_space().fundamental_weights()
>>> L.verma_module(La[Integer(1)] + La[Integer(2)]).is_simple()
False
>>> L.verma_module(-La[Integer(1)] - La[Integer(2)]).is_simple()
True
>>> L.verma_module(Integer(3)/Integer(2)*La[Integer(1)] + Integer(1)/
↳Integer(2)*La[Integer(2)]).is_simple()
False
>>> L.verma_module(Integer(3)/Integer(2)*La[Integer(1)] + Integer(1)/
↳Integer(3)*La[Integer(2)]).is_simple()
True
>>> L.verma_module(-Integer(3)*La[Integer(1)] + Integer(2)/
↳Integer(3)*La[Integer(2)]).is_simple()
True

```

is_singular()

Return if self is a singular Verma module.

A Verma module M_λ is *singular* if there does not exist a dominant weight $\tilde{\lambda}$ that is in the dot orbit of λ . We call a Verma module *regular* otherwise.

EXAMPLES:

```

sage: L = lie_algebras.sl(QQ, 3)
sage: La = L.cartan_type().root_system().weight_lattice().fundamental_
↳weights()
sage: M = L.verma_module(La[1] + La[2])
sage: M.is_singular()
False
sage: M = L.verma_module(La[1] - La[2])
sage: M.is_singular()
True
sage: M = L.verma_module(2*La[1] - 10*La[2])
sage: M.is_singular()
False
sage: M = L.verma_module(-2*La[1] - 2*La[2])
sage: M.is_singular()
False
sage: M = L.verma_module(-4*La[1] - La[2])
sage: M.is_singular()
True

```

```

>>> from sage.all import *
>>> L = lie_algebras.sl(QQ, Integer(3))
>>> La = L.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> M = L.verma_module(La[Integer(1)] + La[Integer(2)])
>>> M.is_singular()
False
>>> M = L.verma_module(La[Integer(1)] - La[Integer(2)])
>>> M.is_singular()
True
>>> M = L.verma_module(Integer(2)*La[Integer(1)] - Integer(10)*La[Integer(2)])
>>> M.is_singular()

```

(continues on next page)

(continued from previous page)

```

False
>>> M = L.verma_module(-Integer(2)*La[Integer(1)] - Integer(2)*La[Integer(2)])
>>> M.is_singular()
False
>>> M = L.verma_module(-Integer(4)*La[Integer(1)] - La[Integer(2)])
>>> M.is_singular()
True

```

lie_algebra()

Return the underlying Lie algebra of `self`.

EXAMPLES:

```

sage: L = lie_algebras.so(QQ, 9)
sage: La = L.cartan_type().root_system().weight_space().fundamental_weights()
sage: M = L.verma_module(La[3] - 1/2*La[1])
sage: M.lie_algebra()
Lie algebra of ['B', 4] in the Chevalley basis

```

```

>>> from sage.all import *
>>> L = lie_algebras.so(QQ, Integer(9))
>>> La = L.cartan_type().root_system().weight_space().fundamental_weights()
>>> M = L.verma_module(La[Integer(3)] - Integer(1)/Integer(2)*La[Integer(1)])
>>> M.lie_algebra()
Lie algebra of ['B', 4] in the Chevalley basis

```

pbw_basis()

Return the PBW basis of the underlying Lie algebra used to define `self`.

EXAMPLES:

```

sage: L = lie_algebras.so(QQ, 8)
sage: La = L.cartan_type().root_system().weight_lattice().fundamental_
↪weights()
sage: M = L.verma_module(La[2] - 2*La[3])
sage: M.pbw_basis()
Universal enveloping algebra of Lie algebra of ['D', 4] in the Chevalley basis
in the Poincare-Birkhoff-Witt basis

```

```

>>> from sage.all import *
>>> L = lie_algebras.so(QQ, Integer(8))
>>> La = L.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> M = L.verma_module(La[Integer(2)] - Integer(2)*La[Integer(3)])
>>> M.pbw_basis()
Universal enveloping algebra of Lie algebra of ['D', 4] in the Chevalley basis
in the Poincare-Birkhoff-Witt basis

```

poincare_birkhoff_witt_basis()

Return the PBW basis of the underlying Lie algebra used to define `self`.

EXAMPLES:

```

sage: L = lie_algebras.so(QQ, 8)
sage: La = L.cartan_type().root_system().weight_lattice().fundamental_
↪weights()
sage: M = L.verma_module(La[2] - 2*La[3])
sage: M.pbw_basis()
Universal enveloping algebra of Lie algebra of ['D', 4] in the Chevalley basis
in the Poincare-Birkhoff-Witt basis

```

```

>>> from sage.all import *
>>> L = lie_algebras.so(QQ, Integer(8))
>>> La = L.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> M = L.verma_module(La[Integer(2)] - Integer(2)*La[Integer(3)])
>>> M.pbw_basis()
Universal enveloping algebra of Lie algebra of ['D', 4] in the Chevalley basis
in the Poincare-Birkhoff-Witt basis

```

weight_space_basis(*d*)

Return a basis for the *d*-th homogeneous component of *self*.

EXAMPLES:

```

sage: L = lie_algebras.sl(QQ, 3)
sage: P = L.cartan_type().root_system().weight_lattice()
sage: La = P.fundamental_weights()
sage: al = P.simple_roots()
sage: mu = 2*La[1] + 3*La[2]
sage: M = L.verma_module(mu)
sage: M.homogeneous_component_basis(mu - al[2])
[f[-alpha[2]]*v[2*Lambda[1] + 3*Lambda[2]]]
sage: M.homogeneous_component_basis(mu - 3*al[2])
[f[-alpha[2]]^3*v[2*Lambda[1] + 3*Lambda[2]]]
sage: M.homogeneous_component_basis(mu - 3*al[2] - 2*al[1])
[f[-alpha[2]]*f[-alpha[1] - alpha[2]]^2*v[2*Lambda[1] + 3*Lambda[2]],
 f[-alpha[2]]^2*f[-alpha[1]]*f[-alpha[1] - alpha[2]]*v[2*Lambda[1] + ↪
↪3*Lambda[2]],
 f[-alpha[2]]^3*f[-alpha[1]]^2*v[2*Lambda[1] + 3*Lambda[2]]]
sage: M.homogeneous_component_basis(mu - La[1])
Family ()

```

```

>>> from sage.all import *
>>> L = lie_algebras.sl(QQ, Integer(3))
>>> P = L.cartan_type().root_system().weight_lattice()
>>> La = P.fundamental_weights()
>>> al = P.simple_roots()
>>> mu = Integer(2)*La[Integer(1)] + Integer(3)*La[Integer(2)]
>>> M = L.verma_module(mu)
>>> M.homogeneous_component_basis(mu - al[Integer(2)])
[f[-alpha[2]]*v[2*Lambda[1] + 3*Lambda[2]]]
>>> M.homogeneous_component_basis(mu - Integer(3)*al[Integer(2)])
[f[-alpha[2]]^3*v[2*Lambda[1] + 3*Lambda[2]]]
>>> M.homogeneous_component_basis(mu - Integer(3)*al[Integer(2)] - ↪
↪Integer(2)*al[Integer(1)])
[f[-alpha[2]]*f[-alpha[1] - alpha[2]]^2*v[2*Lambda[1] + 3*Lambda[2]],

```

(continues on next page)

(continued from previous page)

```

f[-alpha[2]]^2*f[-alpha[1]]*f[-alpha[1] - alpha[2]]*v[2*Lambda[1] +
↪3*Lambda[2]],
f[-alpha[2]]^3*f[-alpha[1]]^2*v[2*Lambda[1] + 3*Lambda[2]]]
>>> M.homogeneous_component_basis(mu - La[Integer(1)])
Family ()

```

```

class sage.algebras.lie_algebras.verma_module.VermaModuleHomset(X, Y, category=None,
                                                                base=None, check=True)

```

Bases: `Homset`

The set of morphisms from a Verma module to another module in Category $\mathcal{O}$ considered as $U(\mathfrak{g})$ -representations.

This currently assumes the codomain is a Verma module, its dual, or a simple module.

Let $M_{w \cdot \lambda}$ and $M_{w' \cdot \lambda'}$ be Verma modules, $\cdot$ is the dot action, and $\lambda + \rho$, $\lambda' + \rho$ are dominant weights. Then we have

$$\dim \operatorname{hom}(M_{w \cdot \lambda}, M_{w' \cdot \lambda'}) = 1$$

if and only if $\lambda = \lambda'$ and $w' \leq w$ in Bruhat order. Otherwise the homset is 0 dimensional.

If the codomain is a dual Verma module $M_\mu^\vee$, then the homset is $\delta_{\lambda\mu}$ dimensional. When $\mu = \lambda$, the image is the simple module L_λ .

Element

alias of `VermaModuleMorphism`

basis()

Return a basis of `self`.

EXAMPLES:

```

sage: L = lie_algebras.sl(QQ, 3)
sage: La = L.cartan_type().root_system().weight_lattice().fundamental_
↪weights()
sage: M = L.verma_module(La[1] + La[2])
sage: Mp = L.verma_module(M.highest_weight().dot_action([2]))
sage: H = Hom(Mp, M)
sage: list(H.basis()) == [H.natural_map()]
True

sage: Mp = L.verma_module(La[1] + 2*La[2])
sage: H = Hom(Mp, M)
sage: H.basis()
Family ()

```

```

>>> from sage.all import *
>>> L = lie_algebras.sl(QQ, Integer(3))
>>> La = L.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> M = L.verma_module(La[Integer(1)] + La[Integer(2)])
>>> Mp = L.verma_module(M.highest_weight().dot_action([Integer(2)]))
>>> H = Hom(Mp, M)
>>> list(H.basis()) == [H.natural_map()]
True

```

(continues on next page)

(continued from previous page)

```

>>> Mp = L.verma_module(La[Integer(1)] + Integer(2)*La[Integer(2)])
>>> H = Hom(Mp, M)
>>> H.basis()
Family ()

```

dimension()

Return the dimension of `self` (as a vector space over the base ring).

EXAMPLES:

```

sage: L = lie_algebras.sl(QQ, 3)
sage: La = L.cartan_type().root_system().weight_lattice().fundamental_
      ↪ weights()
sage: M = L.verma_module(La[1] + La[2])
sage: Mp = L.verma_module(M.highest_weight().dot_action([2]))
sage: H = Hom(Mp, M)
sage: H.dimension()
1

sage: Mp = L.verma_module(La[1] + 2*La[2])
sage: H = Hom(Mp, M)
sage: H.dimension()
0

```

```

>>> from sage.all import *
>>> L = lie_algebras.sl(QQ, Integer(3))
>>> La = L.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> M = L.verma_module(La[Integer(1)] + La[Integer(2)])
>>> Mp = L.verma_module(M.highest_weight().dot_action([Integer(2)]))
>>> H = Hom(Mp, M)
>>> H.dimension()
1

>>> Mp = L.verma_module(La[Integer(1)] + Integer(2)*La[Integer(2)])
>>> H = Hom(Mp, M)
>>> H.dimension()
0

```

highest_weight_image()

Return the image of the highest weight vector of the domain in the codomain.

EXAMPLES:

```

sage: g = LieAlgebra(QQ, cartan_type=['C', 3])
sage: La = g.cartan_type().root_system().weight_lattice().fundamental_
      ↪ weights()
sage: M = g.verma_module(La[1] + 2*La[3])
sage: Mc = M.dual()
sage: H = Hom(M, Mc)
sage: H.highest_weight_image()
v[Lambda[1] + 2*Lambda[3]]^*
sage: L = H.natural_map().image()
sage: Hp = Hom(M, L)

```

(continues on next page)

(continued from previous page)

```
sage: Hp.highest_weight_image()
u[Lambda[1] + 2*Lambda[3]]
```

```
>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['C', Integer(3)])
>>> La = g.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> M = g.verma_module(La[Integer(1)] + Integer(2)*La[Integer(3)])
>>> Mc = M.dual()
>>> H = Hom(M, Mc)
>>> H.highest_weight_image()
v[Lambda[1] + 2*Lambda[3]]^*
>>> L = H.natural_map().image()
>>> Hp = Hom(M, L)
>>> Hp.highest_weight_image()
u[Lambda[1] + 2*Lambda[3]]
```

natural_map()

Return the “natural map” of self.

EXAMPLES:

```
sage: L = lie_algebras.sl(QQ, 3)
sage: La = L.cartan_type().root_system().weight_lattice().fundamental_
↪weights()
sage: M = L.verma_module(La[1] + La[2])
sage: Mp = L.verma_module(M.highest_weight().dot_action([2]))
sage: H = Hom(Mp, M)
sage: H.natural_map()
Verma module morphism:
  From: Verma module with highest weight 3*Lambda[1] - 3*Lambda[2]
        of Lie algebra of ['A', 2] in the Chevalley basis
  To:   Verma module with highest weight Lambda[1] + Lambda[2]
        of Lie algebra of ['A', 2] in the Chevalley basis
  Defn: v[3*Lambda[1] - 3*Lambda[2]] |-->
        f[-alpha[2]]^2*v[Lambda[1] + Lambda[2]]

sage: Mp = L.verma_module(La[1] + 2*La[2])
sage: H = Hom(Mp, M)
sage: H.natural_map()
Verma module morphism:
  From: Verma module with highest weight Lambda[1] + 2*Lambda[2]
        of Lie algebra of ['A', 2] in the Chevalley basis
  To:   Verma module with highest weight Lambda[1] + Lambda[2]
        of Lie algebra of ['A', 2] in the Chevalley basis
  Defn: v[Lambda[1] + 2*Lambda[2]] |--> 0
```

```
>>> from sage.all import *
>>> L = lie_algebras.sl(QQ, Integer(3))
>>> La = L.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> M = L.verma_module(La[Integer(1)] + La[Integer(2)])
>>> Mp = L.verma_module(M.highest_weight().dot_action([Integer(2)]))
>>> H = Hom(Mp, M)
```

(continues on next page)

(continued from previous page)

```

>>> H.natural_map()
Verma module morphism:
  From: Verma module with highest weight 3*Lambda[1] - 3*Lambda[2]
        of Lie algebra of ['A', 2] in the Chevalley basis
  To:   Verma module with highest weight Lambda[1] + Lambda[2]
        of Lie algebra of ['A', 2] in the Chevalley basis
  Defn: v[3*Lambda[1] - 3*Lambda[2]] |-->
        f[-alpha[2]]^2*v[Lambda[1] + Lambda[2]]

>>> Mp = L.verma_module(La[Integer(1)] + Integer(2)*La[Integer(2)])
>>> H = Hom(Mp, M)
>>> H.natural_map()
Verma module morphism:
  From: Verma module with highest weight Lambda[1] + 2*Lambda[2]
        of Lie algebra of ['A', 2] in the Chevalley basis
  To:   Verma module with highest weight Lambda[1] + Lambda[2]
        of Lie algebra of ['A', 2] in the Chevalley basis
  Defn: v[Lambda[1] + 2*Lambda[2]] |--> 0

```

singular_vector()

Return the singular vector in the codomain corresponding to the domain's highest weight element or `None` if no such element exists.

ALGORITHM:

We essentially follow the algorithm laid out in [deG2005]. We split the main computation into two cases. If there exists an i such that $\langle \lambda + \rho, \alpha_i^\vee \rangle = m > 0$ (i.e., the weight λ is i -dominant with respect to the dot action), then we use the $\mathfrak{sl}_2$ relation on $M_{s_i \cdot \lambda} \rightarrow M_\lambda$ to construct the singular vector $f_i^m v_\lambda$. Otherwise we find the shortest root α such that $\langle \lambda + \rho, \alpha^\vee \rangle > 0$ and explicitly compute the kernel with respect to the weight basis elements. We iterate this until we reach μ .

EXAMPLES:

```

sage: L = lie_algebras.sp(QQ, 6)
sage: La = L.cartan_type().root_system().weight_space().fundamental_weights()
sage: la = La[1] - La[3]
sage: mu = la.dot_action([1, 2])
sage: M = L.verma_module(la)
sage: Mp = L.verma_module(mu)
sage: H = Hom(Mp, M)
sage: v = H.singular_vector(); v
f[-alpha[2]]*f[-alpha[1]]^3*v[Lambda[1] - Lambda[3]]
+ 3*f[-alpha[1]]^2*f[-alpha[1] - alpha[2]]*v[Lambda[1] - Lambda[3]]
sage: v.degree() == Mp.highest_weight()
True

```

```

>>> from sage.all import *
>>> L = lie_algebras.sp(QQ, Integer(6))
>>> La = L.cartan_type().root_system().weight_space().fundamental_weights()
>>> la = La[Integer(1)] - La[Integer(3)]
>>> mu = la.dot_action([Integer(1), Integer(2)])
>>> M = L.verma_module(la)
>>> Mp = L.verma_module(mu)

```

(continues on next page)

(continued from previous page)

```
>>> H = Hom(Mp, M)
>>> v = H.singular_vector(); v
f[-alpha[2]]*f[-alpha[1]]^3*v[Lambda[1] - Lambda[3]]
+ 3*f[-alpha[1]]^2*f[-alpha[1] - alpha[2]]*v[Lambda[1] - Lambda[3]]
>>> v.degree() == Mp.highest_weight()
True
```

```
sage: L = LieAlgebra(QQ, cartan_type=['F', 4])
sage: La = L.cartan_type().root_system().weight_space().fundamental_weights()
sage: la = La[1] + La[2] - La[3]
sage: mu = la.dot_action([1,2,3,2])
sage: M = L.verma_module(la)
sage: Mp = L.verma_module(mu)
sage: H = Hom(Mp, M)
sage: v = H.singular_vector()
sage: pbw = M.pbw_basis()
sage: E = [pbw(e) for e in L.e()]
sage: all(e * v == M.zero() for e in E) # long time
True
sage: v.degree() == Mp.highest_weight()
True
```

```
>>> from sage.all import *
>>> L = LieAlgebra(QQ, cartan_type=['F', Integer(4)])
>>> La = L.cartan_type().root_system().weight_space().fundamental_weights()
>>> la = La[Integer(1)] + La[Integer(2)] - La[Integer(3)]
>>> mu = la.dot_action([Integer(1), Integer(2), Integer(3), Integer(2)])
>>> M = L.verma_module(la)
>>> Mp = L.verma_module(mu)
>>> H = Hom(Mp, M)
>>> v = H.singular_vector()
>>> pbw = M.pbw_basis()
>>> E = [pbw(e) for e in L.e()]
>>> all(e * v == M.zero() for e in E) # long time
True
>>> v.degree() == Mp.highest_weight()
True
```

When $w \cdot \lambda \notin \lambda + Q^-$, there does not exist a singular vector:

```
sage: L = lie_algebras.sl(QQ, 4)
sage: La = L.cartan_type().root_system().weight_space().fundamental_weights()
sage: la = 3/7*La[1] - 1/2*La[3]
sage: mu = la.dot_action([1,2])
sage: M = L.verma_module(la)
sage: Mp = L.verma_module(mu)
sage: H = Hom(Mp, M)
sage: H.singular_vector() is None
True
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```

>>> L = lie_algebras.sl(QQ, Integer(4))
>>> La = L.cartan_type().root_system().weight_space().fundamental_weights()
>>> la = Integer(3)/Integer(7)*La[Integer(1)] - Integer(1)/
↳Integer(2)*La[Integer(3)]
>>> mu = la.dot_action([Integer(1), Integer(2)])
>>> M = L.verma_module(la)
>>> Mp = L.verma_module(mu)
>>> H = Hom(Mp, M)
>>> H.singular_vector() is None
True

```

When we need to apply a non-simple reflection, we can compute the singular vector (see [Issue #36793](#)):

```

sage: g = LieAlgebra(QQ, cartan_type=['A', 2])
sage: La = g.cartan_type().root_system().weight_lattice().fundamental_
↳weights()
sage: M = g.verma_module((0*La[1]).dot_action([1]))
sage: Mp = g.verma_module((0*La[1]).dot_action([1, 2]))
sage: H = Hom(Mp, M)
sage: v = H.singular_vector(); v
1/2*f[-alpha[2]]*f[-alpha[1]]*v[-2*Lambda[1] + Lambda[2]]
+ f[-alpha[1] - alpha[2]]*v[-2*Lambda[1] + Lambda[2]]
sage: pbw = M.pbw_basis()
sage: E = [pbw(e) for e in g.e()]
sage: all(e * v == M.zero() for e in E)
True
sage: v.degree() == Mp.highest_weight()
True

```

```

>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['A', Integer(2)])
>>> La = g.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> M = g.verma_module((Integer(0)*La[Integer(1)]).dot_action([Integer(1)]))
>>> Mp = g.verma_module((Integer(0)*La[Integer(1)]).dot_action([Integer(1),
↳Integer(2)]))
>>> H = Hom(Mp, M)
>>> v = H.singular_vector(); v
1/2*f[-alpha[2]]*f[-alpha[1]]*v[-2*Lambda[1] + Lambda[2]]
+ f[-alpha[1] - alpha[2]]*v[-2*Lambda[1] + Lambda[2]]
>>> pbw = M.pbw_basis()
>>> E = [pbw(e) for e in g.e()]
>>> all(e * v == M.zero() for e in E)
True
>>> v.degree() == Mp.highest_weight()
True

```

zero()

Return the zero morphism of `self`.

EXAMPLES:

```

sage: L = lie_algebras.sp(QQ, 6)

```

(continues on next page)

(continued from previous page)

```

sage: La = L.cartan_type().root_system().weight_space().fundamental_weights()
sage: M = L.verma_module(La[1] + 2/3*La[2])
sage: Mp = L.verma_module(La[2] - La[3])
sage: H = Hom(Mp, M)
sage: H.zero()
Verma module morphism:
  From: Verma module with highest weight Lambda[2] - Lambda[3]
        of Lie algebra of ['C', 3] in the Chevalley basis
  To:   Verma module with highest weight Lambda[1] + 2/3*Lambda[2]
        of Lie algebra of ['C', 3] in the Chevalley basis
  Defn: v[Lambda[2] - Lambda[3]] |--> 0

```

```

>>> from sage.all import *
>>> L = lie_algebras.sp(QQ, Integer(6))
>>> La = L.cartan_type().root_system().weight_space().fundamental_weights()
>>> M = L.verma_module(La[Integer(1)] + Integer(2)/Integer(3)*La[Integer(2)])
>>> Mp = L.verma_module(La[Integer(2)] - La[Integer(3)])
>>> H = Hom(Mp, M)
>>> H.zero()
Verma module morphism:
  From: Verma module with highest weight Lambda[2] - Lambda[3]
        of Lie algebra of ['C', 3] in the Chevalley basis
  To:   Verma module with highest weight Lambda[1] + 2/3*Lambda[2]
        of Lie algebra of ['C', 3] in the Chevalley basis
  Defn: v[Lambda[2] - Lambda[3]] |--> 0

```

class sage.algebras.lie_algebras.verma_module.**VermaModuleMorphism**(parent, scalar)

Bases: [Morphism](#)

A morphism of a Verma module to another module in Category $\mathcal{O}$.

image()

Return the image of self.

EXAMPLES:

```

sage: g = LieAlgebra(QQ, cartan_type=['B', 2])
sage: La = g.cartan_type().root_system().weight_lattice().fundamental_
->weights()
sage: M = g.verma_module(La[1] + 2*La[2])
sage: Mp = g.verma_module(La[1] + 3*La[2])
sage: phi = Hom(M, Mp).natural_map()
sage: phi.image()
Free module generated by {} over Rational Field
sage: Mc = M.dual()
sage: phi = Hom(M, Mc).natural_map()
sage: L = phi.image(); L
Simple module with highest weight Lambda[1] + 2*Lambda[2] of
Lie algebra of ['B', 2] in the Chevalley basis
sage: psi = Hom(M, L).natural_map()
sage: psi.image()
Simple module with highest weight Lambda[1] + 2*Lambda[2] of
Lie algebra of ['B', 2] in the Chevalley basis

```

```

>>> from sage.all import *
>>> g = LieAlgebra(QQ, cartan_type=['B', Integer(2)])
>>> La = g.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> M = g.verma_module(La[Integer(1)] + Integer(2)*La[Integer(2)])
>>> Mp = g.verma_module(La[Integer(1)] + Integer(3)*La[Integer(2)])
>>> phi = Hom(M, Mp).natural_map()
>>> phi.image()
Free module generated by {} over Rational Field
>>> Mc = M.dual()
>>> phi = Hom(M, Mc).natural_map()
>>> L = phi.image(); L
Simple module with highest weight Lambda[1] + 2*Lambda[2] of
Lie algebra of ['B', 2] in the Chevalley basis
>>> psi = Hom(M, L).natural_map()
>>> psi.image()
Simple module with highest weight Lambda[1] + 2*Lambda[2] of
Lie algebra of ['B', 2] in the Chevalley basis

```

is_injective()

Return if self is injective or not.

A morphism $\phi : M \rightarrow M'$ from a Verma module M to another Verma module M' is injective if and only if $\dim \text{hom}(M, M') = 1$ and $\phi \neq 0$. If M' is a dual Verma or simple module, then the result is not injective.

EXAMPLES:

```

sage: L = lie_algebras.sl(QQ, 3)
sage: La = L.cartan_type().root_system().weight_lattice().fundamental_
↪weights()
sage: M = L.verma_module(La[1] + La[2])
sage: Mp = L.verma_module(M.highest_weight().dot_action([1,2]))
sage: Mpp = L.verma_module(M.highest_weight().dot_action([1,2]) + La[1])
sage: phi = Hom(Mp, M).natural_map()
sage: phi.is_injective()
True
sage: (0 * phi).is_injective()
False
sage: psi = Hom(Mpp, Mp).natural_map()
sage: psi.is_injective()
False

```

```

>>> from sage.all import *
>>> L = lie_algebras.sl(QQ, Integer(3))
>>> La = L.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> M = L.verma_module(La[Integer(1)] + La[Integer(2)])
>>> Mp = L.verma_module(M.highest_weight().dot_action([Integer(1),
↪Integer(2)]))
>>> Mpp = L.verma_module(M.highest_weight().dot_action([Integer(1),
↪Integer(2)] + La[Integer(1)]))
>>> phi = Hom(Mp, M).natural_map()
>>> phi.is_injective()
True
>>> (Integer(0) * phi).is_injective()

```

(continues on next page)

(continued from previous page)

```
False
>>> psi = Hom(Mpp, Mp).natural_map()
>>> psi.is_injective()
False
```

is_surjective()

Return if `self` is surjective or not.

A morphism $\phi : M \rightarrow M'$ from a Verma module M to another Verma module M' is surjective if and only if the domain is equal to the codomain and it is not the zero morphism.

If M' is a simple module, then this surjective if and only if $\dim \text{hom}(M, M') = 1$ and $\phi \neq 0$.

EXAMPLES:

```
sage: L = lie_algebras.sl(QQ, 3)
sage: La = L.cartan_type().root_system().weight_lattice().fundamental_
      ↪weights()
sage: M = L.verma_module(La[1] + La[2])
sage: Mp = L.verma_module(M.highest_weight().dot_action([1,2]))
sage: phi = Hom(M, Mp).natural_map()
sage: phi.is_surjective()
True
sage: (0 * phi).is_surjective()
False
sage: psi = Hom(Mp, M).natural_map()
sage: psi.is_surjective()
False
```

```
>>> from sage.all import *
>>> L = lie_algebras.sl(QQ, Integer(3))
>>> La = L.cartan_type().root_system().weight_lattice().fundamental_weights()
>>> M = L.verma_module(La[Integer(1)] + La[Integer(2)])
>>> Mp = L.verma_module(M.highest_weight().dot_action([Integer(1),
      ↪Integer(2)]))
>>> phi = Hom(M, Mp).natural_map()
>>> phi.is_surjective()
True
>>> (Integer(0) * phi).is_surjective()
False
>>> psi = Hom(Mp, M).natural_map()
>>> psi.is_surjective()
False
```

9.2.24 Virasoro Algebra and Related Lie Algebras

AUTHORS:

- Travis Scrimshaw (2013-05-03): Initial version

class sage.algebras.lie_algebras.virasoro.**ChargelessRepresentation**(V, a, b)

Bases: `CombinatorialFreeModule`

A chargeless representation of the Virasoro algebra.

Let L be the Virasoro algebra over the field F of characteristic 0. For $\alpha, \beta \in R$, we denote $V_{a,b}$ as the (a,b) -chargeless representation of L , which is the F -span of $\{v_k \mid k \in \mathbf{Z}\}$ with L action

$$\begin{aligned} d_n \cdot v_k &= (an + b - k)v_{n+k}, \\ c \cdot v_k &= 0, \end{aligned}$$

This comes from the action of $d_n = -t^{n+1} \frac{d}{dt}$ on $F[t, t^{-1}]$ (recall that L is the central extension of the algebra of derivations of $F[t, t^{-1}]$), where

$$V_{a,b} = F[t, t^{-1}]t^{a-b}(dt)^{-a}$$

and $v_k = t^{a-b+k}(dz)^{-a}$.

The chargeless representations are either irreducible or contains exactly two simple subquotients, one of which is the trivial representation and the other is $F[t, t^{-1}]/F$. The non-trivial simple subquotients are called the *intermediate series*.

The module $V_{a,b}$ is irreducible if and only if $a \neq 0, -1$ or $b \notin \mathbf{Z}$. When $a = 0$ and $b \in \mathbf{Z}$, then there exists a subrepresentation isomorphic to the trivial representation. If $a = -1$ and $b \in \mathbf{Z}$, then there exists a subrepresentation V such that $V_{a,b}/V$ is isomorphic to $K \frac{dt}{t}$ and V is irreducible.

In characteristic p , the non-trivial simple subquotient is isomorphic to $F[t, t^{-1}]/F[t^p, t^{-p}]$. For $p \neq 2, 3$, then the action is given as above.

EXAMPLES:

We first construct the irreducible $V_{1/2, 3/4}$ and do some basic computations:

```
sage: L = lie_algebras.VirasoroAlgebra(QQ)
sage: M = L.chargeless_representation(1/2, 3/4)
sage: d = L.basis()
sage: v = M.basis()
sage: d[3] * v[2]
1/4*v[5]
sage: d[3] * v[-1]
13/4*v[2]
sage: (d[3] - d[-2]) * (v[-1] + 1/2*v[0] - v[4])
-3/4*v[-3] + 1/8*v[-2] - v[2] + 9/8*v[3] + 7/4*v[7]
```

```
>>> from sage.all import *
>>> L = lie_algebras.VirasoroAlgebra(QQ)
>>> M = L.chargeless_representation(Integer(1)/Integer(2), Integer(3)/Integer(4))
>>> d = L.basis()
>>> v = M.basis()
>>> d[Integer(3)] * v[Integer(2)]
1/4*v[5]
>>> d[Integer(3)] * v[-Integer(1)]
13/4*v[2]
>>> (d[Integer(3)] - d[-Integer(2)]) * (v[-Integer(1)] + Integer(1)/
↪ Integer(2)*v[Integer(0)] - v[Integer(4)])
-3/4*v[-3] + 1/8*v[-2] - v[2] + 9/8*v[3] + 7/4*v[7]
```

We construct the reducible $V_{0,2}$ and the trivial subrepresentation given by the span of v_2 . We verify this for $\{d_i \mid -10 \leq i < 10\}$:

```
sage: M = L.chargeless_representation(0, 2)
sage: v = M.basis()
sage: all(d[i] * v[2] == M.zero() for i in range(-10, 10))
True
```

```
>>> from sage.all import *
>>> M = L.chargeless_representation(Integer(0), Integer(2))
>>> v = M.basis()
>>> all(d[i] * v[Integer(2)] == M.zero() for i in range(-Integer(10),
↪Integer(10)))
True
```

REFERENCES:

- [Mat1992]
- [IK2010]

class Element

Bases: `IndexedFreeModuleElement`

degree_on_basis(*i*)

Return the degree of the basis element indexed by *i*, which is *i*.

EXAMPLES:

```
sage: L = lie_algebras.VirasoroAlgebra(QQ)
sage: M = L.chargeless_representation(1/2, 3/4)
sage: M.degree_on_basis(-3)
-3
```

```
>>> from sage.all import *
>>> L = lie_algebras.VirasoroAlgebra(QQ)
>>> M = L.chargeless_representation(Integer(1)/Integer(2), Integer(3)/
↪Integer(4))
>>> M.degree_on_basis(-Integer(3))
-3
```

parameters()

Return the parameters (a, b) of self.

EXAMPLES:

```
sage: L = lie_algebras.VirasoroAlgebra(QQ)
sage: M = L.chargeless_representation(1/2, 3/4)
sage: M.parameters()
(1/2, 3/4)
```

```
>>> from sage.all import *
>>> L = lie_algebras.VirasoroAlgebra(QQ)
>>> M = L.chargeless_representation(Integer(1)/Integer(2), Integer(3)/
↪Integer(4))
>>> M.parameters()
(1/2, 3/4)
```

virasoro_algebra()

Return the Virasoro algebra `self` is a representation of.

EXAMPLES:

```
sage: L = lie_algebras.VirasoroAlgebra(QQ)
sage: M = L.chargeless_representation(1/2, 3/4)
sage: M.virasoro_algebra() is L
True
```

```
>>> from sage.all import *
>>> L = lie_algebras.VirasoroAlgebra(QQ)
>>> M = L.chargeless_representation(Integer(1)/Integer(2), Integer(3)/
↳ Integer(4))
>>> M.virasoro_algebra() is L
True
```

class `sage.algebras.lie_algebras.virasoro.LieAlgebraRegularVectorFields(R)`

Bases: *InfinitelyGeneratedLieAlgebra*, *IndexedGenerators*

The Lie algebra of regular vector fields on $\mathbf{C}^\times$.

This is the Lie algebra with basis $\{d_i\}_{i \in \mathbf{Z}}$ and subject to the relations

$$[d_i, d_j] = (i - j)d_{i+j}.$$

This is also known as the Witt (Lie) algebra.

Note

This differs from some conventions (e.g., [Ka1990]), where we have $d'_i \mapsto -d_i$.

REFERENCES:

- [Wikipedia article Witt_algebra](#)

See also

WittLieAlgebra_charp

class **Element**

Bases: *LieAlgebraElement*

bracket_on_basis(*i, j*)

Return the bracket of basis elements indexed by *x* and *y* where *x* < *y*.

(This particular implementation actually does not require *x* < *y*.)

EXAMPLES:

```
sage: L = lie_algebras.regular_vector_fields(QQ)
sage: L.bracket_on_basis(2, -2)
4*d[0]
sage: L.bracket_on_basis(2, 4)
```

(continues on next page)

(continued from previous page)

```
-2*d[6]
sage: L.bracket_on_basis(4, 4)
0
```

```
>>> from sage.all import *
>>> L = lie_algebras.regular_vector_fields(QQ)
>>> L.bracket_on_basis(Integer(2), -Integer(2))
4*d[0]
>>> L.bracket_on_basis(Integer(2), Integer(4))
-2*d[6]
>>> L.bracket_on_basis(Integer(4), Integer(4))
0
```

degree_on_basis(i)

Return the degree of the basis element indexed by i , which is i .

EXAMPLES:

```
sage: L = lie_algebras.regular_vector_fields(QQ)
sage: L.degree_on_basis(2)
2
```

```
>>> from sage.all import *
>>> L = lie_algebras.regular_vector_fields(QQ)
>>> L.degree_on_basis(Integer(2))
2
```

lie_algebra_generators()

Return the generators of `self` as a Lie algebra.

EXAMPLES:

```
sage: L = lie_algebras.regular_vector_fields(QQ)
sage: L.lie_algebra_generators()
Lazy family (generator map(i))_{i in Integer Ring}
```

```
>>> from sage.all import *
>>> L = lie_algebras.regular_vector_fields(QQ)
>>> L.lie_algebra_generators()
Lazy family (generator map(i))_{i in Integer Ring}
```

some_elements()

Return some elements of `self`.

EXAMPLES:

```
sage: L = lie_algebras.regular_vector_fields(QQ)
sage: L.some_elements()
[d[0], d[2], d[-2], d[-1] + d[0] - 3*d[1]]
```

```
>>> from sage.all import *
>>> L = lie_algebras.regular_vector_fields(QQ)
```

(continues on next page)

(continued from previous page)

```
>>> L.some_elements()
[d[0], d[2], d[-2], d[-1] + d[0] - 3*d[1]]
```

class sage.algebras.lie_algebras.virasoro.**VermaModule**(V, c, h)

Bases: `CombinatorialFreeModule`

A Verma module of the Virasoro algebra.

The Virasoro algebra admits a triangular decomposition

$$V_- \oplus R d_0 \oplus R \hat{c} \oplus V_+,$$

where V_- (resp. V_+) is the span of $\{d_i \mid i < 0\}$ (resp. $\{d_i \mid i > 0\}$). We can construct the *Verma module* $M_{c,h}$ as the induced representation of the $R d_0 \oplus R \hat{c} \oplus V_+$ representation $R_{c,H} = Rv$, where

$$V_+ v = 0, \quad \hat{c} v = c v, \quad d_0 v = h v.$$

Therefore, we have a basis of $M_{c,h}$

$$\{L_{i_1} \cdots L_{i_k} v \mid i_1 \leq \cdots \leq i_k < 0\}.$$

Moreover, the Verma modules are the free objects in the category of highest weight representations of V and are indecomposable. The Verma module $M_{c,h}$ is irreducible for generic values of c and h and when it is reducible, the quotient by the maximal submodule is the unique irreducible highest weight representation $V_{c,h}$.

EXAMPLES:

We construct a Verma module and do some basic computations:

```
sage: L = lie_algebras.VirasoroAlgebra(QQ)
sage: M = L.verma_module(3, 0)
sage: d = L.basis()
sage: v = M.highest_weight_vector()
sage: d[3] * v
0
sage: d[-3] * v
d[-3]*v
sage: d[-1] * (d[-3] * v)
2*d[-4]*v + d[-3]*d[-1]*v
sage: d[2] * (d[-1] * (d[-3] * v))
12*d[-2]*v + 5*d[-1]*d[-1]*v
```

```
>>> from sage.all import *
>>> L = lie_algebras.VirasoroAlgebra(QQ)
>>> M = L.verma_module(Integer(3), Integer(0))
>>> d = L.basis()
>>> v = M.highest_weight_vector()
>>> d[Integer(3)] * v
0
>>> d[-Integer(3)] * v
d[-3]*v
>>> d[-Integer(1)] * (d[-Integer(3)] * v)
2*d[-4]*v + d[-3]*d[-1]*v
>>> d[Integer(2)] * (d[-Integer(1)] * (d[-Integer(3)] * v))
12*d[-2]*v + 5*d[-1]*d[-1]*v
```


We verify that $d_{-1}v$ is a singular vector for $\{d_i \mid 1 \leq i < 20\}$:

```
sage: w = M.basis() [-1]; w
d[-1]*v
sage: all(d[i] * w == M.zero() for i in range(1,20))
True
```

```
>>> from sage.all import *
>>> w = M.basis() [-Integer(1)]; w
d[-1]*v
>>> all(d[i] * w == M.zero() for i in range(Integer(1), Integer(20)))
True
```

We also verify a singular vector for $V_{-2,1}$:

```
sage: M = L.verma_module(-2, 1)
sage: B = M.basis()
sage: w = B[-1,-1] - 2 * B[-2]
sage: d = L.basis()
sage: all(d[i] * w == M.zero() for i in range(1,20))
True
```

```
>>> from sage.all import *
>>> M = L.verma_module(-Integer(2), Integer(1))
>>> B = M.basis()
>>> w = B[-Integer(1), -Integer(1)] - Integer(2) * B[-Integer(2)]
>>> d = L.basis()
>>> all(d[i] * w == M.zero() for i in range(Integer(1), Integer(20)))
True
```

REFERENCES:

- [Wikipedia article Virasoro_algebra#Representation_theory](#)

class Element

Bases: `IndexedFreeModuleElement`

central_charge()

Return the central charge of `self`.

EXAMPLES:

```
sage: L = lie_algebras.VirasoroAlgebra(QQ)
sage: M = L.verma_module(3, 0)
sage: M.central_charge()
3
```

```
>>> from sage.all import *
>>> L = lie_algebras.VirasoroAlgebra(QQ)
>>> M = L.verma_module(Integer(3), Integer(0))
>>> M.central_charge()
3
```

conformal_weight()

Return the conformal weight of `self`.

EXAMPLES:

```
sage: L = lie_algebras.VirasoroAlgebra(QQ)
sage: M = L.verma_module(3, 0)
sage: M.conformal_weight()
3
```

```
>>> from sage.all import *
>>> L = lie_algebras.VirasoroAlgebra(QQ)
>>> M = L.verma_module(Integer(3), Integer(0))
>>> M.conformal_weight()
3
```

degree_on_basis(d)Return the degree of the basis element indexed by d , which is the sum of the entries of d .

EXAMPLES:

```
sage: L = lie_algebras.VirasoroAlgebra(QQ)
sage: M = L.verma_module(-2/7, 3)
sage: M.degree_on_basis((-3, -3, -1))
-7
```

```
>>> from sage.all import *
>>> L = lie_algebras.VirasoroAlgebra(QQ)
>>> M = L.verma_module(-Integer(2)/Integer(7), Integer(3))
>>> M.degree_on_basis((-Integer(3), -Integer(3), -Integer(1)))
-7
```

highest_weight_vector()Return the highest weight vector of `self`.

EXAMPLES:

```
sage: L = lie_algebras.VirasoroAlgebra(QQ)
sage: M = L.verma_module(-2/7, 3)
sage: M.highest_weight_vector()
v
```

```
>>> from sage.all import *
>>> L = lie_algebras.VirasoroAlgebra(QQ)
>>> M = L.verma_module(-Integer(2)/Integer(7), Integer(3))
>>> M.highest_weight_vector()
v
```

virasoro_algebra()Return the Virasoro algebra `self` is a representation of.

EXAMPLES:

```
sage: L = lie_algebras.VirasoroAlgebra(QQ)
sage: M = L.verma_module(1/2, 3/4)
sage: M.virasoro_algebra() is L
True
```

```
>>> from sage.all import *
>>> L = lie_algebras.VirasoroAlgebra(QQ)
>>> M = L.verma_module(Integer(1)/Integer(2), Integer(3)/Integer(4))
>>> M.virasoro_algebra() is L
True
```

class sage.algebras.lie_algebras.virasoro.VirasoroAlgebra(*R*)

Bases: *InfinitelyGeneratedLieAlgebra*, *IndexedGenerators*

The Virasoro algebra.

This is the Lie algebra with basis $\{d_i\}_{i \in \mathbb{Z}} \cup \{c\}$ and subject to the relations

$$[d_i, d_j] = (i - j)d_{i+j} + \frac{1}{12}(i^3 - i)\delta_{i,-j}c$$

and

$$[d_i, c] = 0.$$

(Here, it is assumed that the base ring *R* has 2 invertible.)

This is the universal central extension $\tilde{\mathfrak{d}}$ of the Lie algebra $\mathfrak{d}$ of *regular vector fields* on $\mathbb{C}^\times$.

EXAMPLES:

```
sage: d = lie_algebras.VirasoroAlgebra(QQ)
```

```
>>> from sage.all import *
>>> d = lie_algebras.VirasoroAlgebra(QQ)
```

REFERENCES:

- [Wikipedia article Virasoro_algebra](#)

class Element

Bases: *LieAlgebraElement*

basis()

Return a basis of self.

EXAMPLES:

```
sage: d = lie_algebras.VirasoroAlgebra(QQ)
sage: B = d.basis(); B
Lazy family (basis map(i))_{i in Disjoint union of
                                     Family ({'c'}, Integer Ring)}
sage: B['c']
c
sage: B[3]
d[3]
sage: B[-15]
d[-15]
```

```
>>> from sage.all import *
>>> d = lie_algebras.VirasoroAlgebra(QQ)
>>> B = d.basis(); B
```

(continues on next page)

(continued from previous page)

```

Lazy family (basis map(i))_{i in Disjoint union of
                                Family ({'c'}, Integer Ring)}

>>> B['c']
c
>>> B[Integer(3)]
d[3]
>>> B[-Integer(15)]
d[-15]

```

bracket_on_basis(*i, j*)

Return the bracket of basis elements indexed by *x* and *y* where *x* < *y*.

(This particular implementation actually does not require *x* < *y*.)

EXAMPLES:

```

sage: d = lie_algebras.VirasoroAlgebra(QQ)
sage: d.bracket_on_basis('c', 2)
0
sage: d.bracket_on_basis(2, -2)
4*d[0] + 1/2*c

```

```

>>> from sage.all import *
>>> d = lie_algebras.VirasoroAlgebra(QQ)
>>> d.bracket_on_basis('c', Integer(2))
0
>>> d.bracket_on_basis(Integer(2), -Integer(2))
4*d[0] + 1/2*c

```

c()

The central element *c* in self.

EXAMPLES:

```

sage: d = lie_algebras.VirasoroAlgebra(QQ)
sage: d.c()
c

```

```

>>> from sage.all import *
>>> d = lie_algebras.VirasoroAlgebra(QQ)
>>> d.c()
c

```

chargeless_representation(*a, b*)

Return the chargeless representation of self with parameters *a* and *b*.

See also

ChargelessRepresentation

EXAMPLES:

```
sage: L = lie_algebras.VirasoroAlgebra(QQ)
sage: L.chargeless_representation(3, 2)
Chargeless representation (3, 2) of
The Virasoro algebra over Rational Field
```

```
>>> from sage.all import *
>>> L = lie_algebras.VirasoroAlgebra(QQ)
>>> L.chargeless_representation(Integer(3), Integer(2))
Chargeless representation (3, 2) of
The Virasoro algebra over Rational Field
```

d(i)

Return the element d_i in self.

EXAMPLES:

```
sage: L = lie_algebras.VirasoroAlgebra(QQ)
sage: L.d(2)
d[2]
```

```
>>> from sage.all import *
>>> L = lie_algebras.VirasoroAlgebra(QQ)
>>> L.d(Integer(2))
d[2]
```

degree_on_basis(i)

Return the degree of the basis element indexed by i , which is i and 0 for 'c'.

EXAMPLES:

```
sage: d = lie_algebras.VirasoroAlgebra(QQ)
sage: d.degree_on_basis(2)
2
sage: d.c().degree()
0
sage: (d.c() + d.basis()[0]).is_homogeneous()
True
```

```
>>> from sage.all import *
>>> d = lie_algebras.VirasoroAlgebra(QQ)
>>> d.degree_on_basis(Integer(2))
2
>>> d.c().degree()
0
>>> (d.c() + d.basis()[Integer(0)]).is_homogeneous()
True
```

lie_algebra_generators()

Return the generators of self as a Lie algebra.

EXAMPLES:

```
sage: d = lie_algebras.VirasoroAlgebra(QQ)
sage: d.lie_algebra_generators()
Lazy family (generator map(i))_{i in Integer Ring}
```

```
>>> from sage.all import *
>>> d = lie_algebras.VirasoroAlgebra(QQ)
>>> d.lie_algebra_generators()
Lazy family (generator map(i))_{i in Integer Ring}
```

some_elements()

Return some elements of self.

EXAMPLES:

```
sage: d = lie_algebras.VirasoroAlgebra(QQ)
sage: d.some_elements()
[d[0], d[2], d[-2], c, d[-1] + d[0] - 1/2*d[1] + c]
```

```
>>> from sage.all import *
>>> d = lie_algebras.VirasoroAlgebra(QQ)
>>> d.some_elements()
[d[0], d[2], d[-2], c, d[-1] + d[0] - 1/2*d[1] + c]
```

verma_module(c, h)

Return the Verma module with central charge c and conformal (or highest) weight h .

See also

[*VermaModule*](#)

EXAMPLES:

```
sage: L = lie_algebras.VirasoroAlgebra(QQ)
sage: L.verma_module(3, 2)
Verma module with charge 3 and conformal weight 2 of
The Virasoro algebra over Rational Field
```

```
>>> from sage.all import *
>>> L = lie_algebras.VirasoroAlgebra(QQ)
>>> L.verma_module(Integer(3), Integer(2))
Verma module with charge 3 and conformal weight 2 of
The Virasoro algebra over Rational Field
```

class sage.algebras.lie_algebras.virasoro.**WittLieAlgebra_charp**(R, p)

Bases: [*FinitelyGeneratedLieAlgebra*](#), [*IndexedGenerators*](#)

The p -Witt Lie algebra over a ring R in which $p \cdot 1_R = 0$.

Let R be a ring and p be a positive integer such that $p \cdot 1_R = 0$. The p -Witt Lie algebra over R is the Lie algebra with basis $\{d_0, d_1, \dots, d_{p-1}\}$ and subject to the relations

$$[d_i, d_j] = (i - j)d_{i+j},$$

where the $i + j$ on the right hand side is identified with its remainder modulo p .

See also[*LieAlgebraRegularVectorFields*](#)**class Element**Bases: [*LieAlgebraElement*](#)**bracket_on_basis**(*i, j*)Return the bracket of basis elements indexed by *x* and *y* where $x < y$.(This particular implementation actually does not require $x < y$.)

EXAMPLES:

```
sage: L = lie_algebras.pwitt(Zmod(5), 5)
sage: L.bracket_on_basis(2, 3)
4*d[0]
sage: L.bracket_on_basis(3, 2)
d[0]
sage: L.bracket_on_basis(2, 2)
0
sage: L.bracket_on_basis(1, 3)
3*d[4]
```

```
>>> from sage.all import *
>>> L = lie_algebras.pwitt(Zmod(Integer(5)), Integer(5))
>>> L.bracket_on_basis(Integer(2), Integer(3))
4*d[0]
>>> L.bracket_on_basis(Integer(3), Integer(2))
d[0]
>>> L.bracket_on_basis(Integer(2), Integer(2))
0
>>> L.bracket_on_basis(Integer(1), Integer(3))
3*d[4]
```

degree_on_basis(*i*)Return the degree of the basis element indexed by *i*, which is $i \bmod p$.

EXAMPLES:

```
sage: L = lie_algebras.pwitt(Zmod(5), 5)
sage: L.degree_on_basis(7)
2
sage: L.degree_on_basis(2).parent()
Ring of integers modulo 5
```

```
>>> from sage.all import *
>>> L = lie_algebras.pwitt(Zmod(Integer(5)), Integer(5))
>>> L.degree_on_basis(Integer(7))
2
>>> L.degree_on_basis(Integer(2)).parent()
Ring of integers modulo 5
```

lie_algebra_generators()

Return the generators of `self` as a Lie algebra.

EXAMPLES:

```
sage: L = lie_algebras.pwitt(Zmod(5), 5)
sage: L.lie_algebra_generators()
Finite family {0: d[0], 1: d[1], 2: d[2], 3: d[3], 4: d[4]}
```

```
>>> from sage.all import *
>>> L = lie_algebras.pwitt(Zmod(Integer(5)), Integer(5))
>>> L.lie_algebra_generators()
Finite family {0: d[0], 1: d[1], 2: d[2], 3: d[3], 4: d[4]}
```

some_elements()

Return some elements of `self`.

EXAMPLES:

```
sage: L = lie_algebras.pwitt(Zmod(5), 5)
sage: L.some_elements()
[d[0], d[2], d[3], d[0] + 2*d[1] + d[4]]
```

```
>>> from sage.all import *
>>> L = lie_algebras.pwitt(Zmod(Integer(5)), Integer(5))
>>> L.some_elements()
[d[0], d[2], d[3], d[0] + 2*d[1] + d[4]]
```

9.3 Lie Conformal Algebras

9.3.1 The main classes to work with Lie conformal algebras

Lie Conformal Algebra

Let R be a commutative ring, a *super Lie conformal algebra* [Kac1997] over R (also known as a *vertex Lie algebra*) is an $R[T]$ super module L together with a $\mathbb{Z}/2\mathbb{Z}$ -graded R -bilinear operation (called the λ -bracket) $L \otimes L \rightarrow L[\lambda]$ (polynomials in λ with coefficients in L), $a \otimes b \mapsto [a_\lambda b]$ satisfying

1. Sesquilinearity:

$$[Ta_\lambda b] = -\lambda[a_\lambda b], \quad [a_\lambda Tb] = (\lambda + T)[a_\lambda b].$$

2. Skew-Symmetry:

$$[a_\lambda b] = -(-1)^{p(a)p(b)}[b_{-\lambda-T}a],$$

where $p(a)$ is 0 if a is *even* and 1 if a is *odd*. The bracket in the RHS is computed as follows. First we evaluate $[b_\mu a]$ with the formal parameter μ to the *left*, then replace each appearance of the formal variable μ by $-\lambda - T$. Finally apply T to the coefficients in L .

3. Jacobi identity:

$$[a_\lambda [b_\mu c]] = [[a_{\lambda+\mu} b]_\mu c] + (-1)^{p(a)p(b)}[b_\mu [a_\lambda c]],$$

which is understood as an equality in $L[\lambda, \mu]$.

T is usually called the *translation operation* or the *derivative*. For an element $a \in L$ we will say that Ta is the *derivative of a* . We define the *n -th products* $a_{(n)}b$ for $a, b \in L$ by

$$[a_\lambda b] = \sum_{n \geq 0} \frac{\lambda^n}{n!} a_{(n)}b.$$

A Lie conformal algebra is called *H-Graded* [DSK2006] if there exists a decomposition $L = \bigoplus L_n$ such that the λ -bracket becomes graded of degree -1 , that is:

$$a_{(n)}b \in L_{p+q-n-1} \quad a \in L_p, b \in L_q, n \geq 0.$$

In particular this implies that the action of T increases degree by 1.

Note

In the literature arbitrary gradings are allowed. In this implementation we only support nonnegative rational gradings.

EXAMPLES:

1. The **Virasoro** Lie conformal algebra Vir over a ring R where 12 is invertible has two generators L, C as an $R[T]$ -module. It is the direct sum of a free module of rank 1 generated by L , and a free rank one R module generated by C satisfying $TC = 0$. C is central (the λ -bracket of C with any other vector vanishes). The remaining λ -bracket is given by

$$[L_\lambda L] = TL + 2\lambda L + \frac{\lambda^3}{12}C.$$

2. The **affine** or current Lie conformal algebra $L(\mathfrak{g})$ associated to a finite dimensional Lie algebra $\mathfrak{g}$ with non-degenerate, invariant R -bilinear form $(,)$ is given as a central extension of the free $R[T]$ module generated by $\mathfrak{g}$ by a central element K . The λ -bracket of generators is given by

$$[a_\lambda b] = [a, b] + \lambda(a, b)K, \quad a, b \in \mathfrak{g}$$

3. The **Weyl** Lie conformal algebra, or $\beta - \gamma$ system is given as the central extension of a free $R[T]$ module with two generators β and γ , by a central element K . The only non-trivial brackets among generators are

$$[\beta_\lambda \gamma] = -[\gamma_\lambda \beta] = K$$

4. The **Neveu-Schwarz** super Lie conformal algebra is a super Lie conformal algebra which is an extension of the Virasoro Lie conformal algebra. It consists of a Virasoro generator L as in example 1 above and an *odd* generator G . The remaining brackets are given by:

$$[L_\lambda G] = \left(T + \frac{3}{2}\lambda\right)G \quad [G_\lambda G] = 2L + \frac{\lambda^2}{3}C$$

See also

`sage.algebras.lie_conformal_algebras.examples`

The base class for all Lie conformal algebras is `LieConformalAlgebra`. All subclasses are called through its method `__classcall_private__`. This class provides no functionality besides calling the appropriate constructor.

We provide some convenience classes to define named Lie conformal algebras. See `sage.algebras.lie_conformal_algebras.examples`.

EXAMPLES:

- We construct the Virasoro Lie conformal algebra, its universal enveloping vertex algebra and lift some elements:

```
sage: Vir = lie_conformal_algebras.Virasoro(QQ)
sage: Vir.inject_variables()
Defining L, C
sage: L.bracket(L)
{0: TL, 1: 2*L, 3: 1/2*C}
```

```
>>> from sage.all import *
>>> Vir = lie_conformal_algebras.Virasoro(QQ)
>>> Vir.inject_variables()
Defining L, C
>>> L.bracket(L)
{0: TL, 1: 2*L, 3: 1/2*C}
```

- We construct the Current algebra for $\mathfrak{sl}_2$:

```
sage: R = lie_conformal_algebras.Affine(QQ, 'A1', names = ('e', 'h', 'f'))
sage: R.gens()
(e, h, f, K)
sage: R.inject_variables()
Defining e, h, f, K
sage: e.bracket(f.T())
{0: Th, 1: h, 2: 2*K}
sage: e.T(3)
6*T^(3)e
```

```
>>> from sage.all import *
>>> R = lie_conformal_algebras.Affine(QQ, 'A1', names = ('e', 'h', 'f'))
>>> R.gens()
(e, h, f, K)
>>> R.inject_variables()
Defining e, h, f, K
>>> e.bracket(f.T())
{0: Th, 1: h, 2: 2*K}
>>> e.T(Integer(3))
6*T^(3)e
```

- We construct the $\beta - \gamma$ system by directly giving the λ -brackets of the generators:

```
sage: betagamma_dict = {('b', 'a'):{0:{('K', 0):1}}}
sage: V = LieConformalAlgebra(QQ, betagamma_dict, names=('a', 'b'), weights=(1,0),
↪central_elements=('K',))
sage: V.category()
Category of H-graded finitely generated Lie conformal algebras with basis over_
↪Rational Field
sage: V.inject_variables()
Defining a, b, K
sage: a.bracket(b)
{0: -K}
```

```
>>> from sage.all import *
>>> betagamma_dict = {('b', 'a'):{Integer(0):{('K', Integer(0)):Integer(1)}}}
```

(continues on next page)

(continued from previous page)

```

>>> V = LieConformalAlgebra(QQ, betagamma_dict, names=('a', 'b'),
↳ weights=(Integer(1), Integer(0)), central_elements=('K',))
>>> V.category()
Category of H-graded finitely generated Lie conformal algebras with basis over
↳ Rational Field
>>> V.inject_variables()
Defining a, b, K
>>> a.bracket(b)
{0: -K}

```

AUTHORS:

- Reimundo Heluani (2019-08-09): Initial implementation.

class sage.algebras.lie_conformal_algebras.lie_conformal_algebra.**LieConformalAlgebra**

Bases: UniqueRepresentation, Parent

Lie Conformal Algebras base class and factory.

INPUT:

- **R** – a commutative ring (default: None); the base ring of this Lie conformal algebra. Behaviour is undefined if it is not a field of characteristic zero.
- **arg0** – dictionary (default: None); a dictionary containing the λ brackets of the generators of this Lie conformal algebra. The keys of this dictionary are pairs of either names or indices of the generators and the values are themselves dictionaries. For a pair of generators 'a' and 'b', the value of `arg0[('a', 'b')]` is a dictionary whose keys are positive integer numbers and the corresponding value for the key j is a dictionary itself representing the j -th product $a_{(j)}b$. Thus, for a positive integer number j , the value of `arg0[('a', 'b')][j]` is a dictionary whose entries are pairs ('c', n) where 'c' is the name of a generator and n is a positive number. The value for this key is the coefficient of $\frac{T^n}{n!}c$ in $a_{(j)}b$. For example the `arg0` for the *Virasoro* Lie conformal algebra is:

```
{('L', 'L'):{0:{('L', 1):1}, 1:{('L', 0):2}, 3:{('C', 0):1/2}}}
```

Do not include central elements as keys in this dictionary. Also, if the key ('a', 'b') is present, there is no need to include ('b', 'a') as it is defined by skew-symmetry. Any missing pair (besides the ones defined by skew-symmetry) is assumed to have vanishing λ -bracket.

- **names** – tuple of strings (default: None); the list of names for generators of this Lie conformal algebra. Do not include central elements in this list.
- **central_elements** – tuple of strings (default: None); a list of names for central elements of this Lie conformal algebra
- **index_set** – enumerated set (default: None); an indexing set for the generators of this Lie conformal algebra. Do not include central elements in this list.
- **weights** – tuple of nonnegative rational numbers (default: None); a list of degrees for this Lie conformal algebra. The returned Lie conformal algebra is H-Graded. This tuple needs to have the same cardinality as `index_set` or `names`. Central elements are assumed to have weight 0.
- **parity** – tuple of 0 or 1 (default: tuple of 0); if this is a super Lie conformal algebra, this tuple specifies the parity of each of the non-central generators of this Lie conformal algebra. Central elements are assumed to be even. Notice that if this tuple is present, the category of this Lie conformal algebra is set to be a subcategory of `LieConformalAlgebras(R).Super()`, even if all generators are even.
- **category** – the category that this Lie conformal algebra belongs to

In addition we accept the following keywords:

- `graded` – boolean (default: `False`); if `True`, the returned algebra is H-Graded. If `weights` is not specified, all non-central generators are assigned degree 1. This keyword is ignored if `weights` is specified
- `super` – boolean (default: `False`); if `True`, the returned algebra is a super Lie conformal algebra even if all generators are even. If `parity` is not specified, all generators are assigned even parity. This keyword is ignored if `parity` is specified.

Note

Any remaining keyword is currently passed to `CombinatorialFreeModule`.

EXAMPLES:

We construct the $\beta - \gamma$ system or Weyl Lie conformal algebra:

```
sage: betagamma_dict = {'b', 'a':{0:{('K', 0):1}}}
sage: V = LieConformalAlgebra(QQbar, betagamma_dict, names=('a', 'b'), weights=(1,
↪0), central_elements=('K',))
sage: V.category()
Category of H-graded finitely generated Lie conformal algebras with basis over_
↪Algebraic Field
sage: V.inject_variables()
Defining a, b, K
sage: a.bracket(b)
{0: -K}
```

```
>>> from sage.all import *
>>> betagamma_dict = {'b', 'a':{Integer(0):{('K', Integer(0)):Integer(1)}}}
>>> V = LieConformalAlgebra(QQbar, betagamma_dict, names=('a', 'b'),
↪weights=(Integer(1), Integer(0)), central_elements=('K',))
>>> V.category()
Category of H-graded finitely generated Lie conformal algebras with basis over_
↪Algebraic Field
>>> V.inject_variables()
Defining a, b, K
>>> a.bracket(b)
{0: -K}
```

We construct the current algebra for $\mathfrak{sl}_2$:

```
sage: sl2dict = {('e', 'f'):{0:{('h', 0):1}, 1:{('K', 0):1}}, ('e', 'h'):{0:{('e', 0):-
↪2}}, ('f', 'h'):{0:{('f', 0):2}}, ('h', 'h'):{1:{('K', 0):2}}}
sage: V = LieConformalAlgebra(QQ, sl2dict, names=('e', 'h', 'f'), central_
↪elements=('K',), graded=True)
sage: V.inject_variables()
Defining e, h, f, K
sage: e.bracket(f)
{0: h, 1: K}
sage: h.bracket(e)
{0: 2*e}
sage: e.bracket(f.T())
{0: Th, 1: h, 2: 2*K}
```

(continues on next page)

(continued from previous page)

```

sage: V.category()
Category of H-graded finitely generated Lie conformal algebras with basis over
↳ Rational Field
sage: e.degree()
1

>>> from sage.all import *
>>> sl2dict = {'e','f':{'h',Integer(0):Integer(1)}, Integer(1):{('K',Integer(0):Integer(1)}}, ('e','h':{'e',Integer(0):-Integer(2)}}, ('f','h':{'f',Integer(0):Integer(2)}}, ('h','h':{'Integer(1):{('K', Integer(0):Integer(2))}}
>>> V = LieConformalAlgebra(QQ, sl2dict, names=('e', 'h', 'f'), central_elements=
↳ ('K',), graded=True)
>>> V.inject_variables()
Defining e, h, f, K
>>> e.bracket(f)
{0: h, 1: K}
>>> h.bracket(e)
{0: 2*e}
>>> e.bracket(f.T())
{0: Th, 1: h, 2: 2*K}
>>> V.category()
Category of H-graded finitely generated Lie conformal algebras with basis over
↳ Rational Field
>>> e.degree()
1

```

Todo

This class checks that the provided dictionary is consistent with skew-symmetry. It does not check that it is consistent with the Jacobi identity.

See also

`sage.algebras.lie_conformal_algebras.graded_lie_conformal_algebra`

Examples of Lie Conformal Algebras

We implement the following examples of Lie conformal algebras:

- *Abelian Lie conformal algebra*
- *Affine Lie conformal algebra*
- *Bosonic Ghosts*
- *Fermionic Ghosts*
- *Free Bosons*
- *Free Fermions*
- *N=2 super Lie Conformal algebra*

- *Neveu-Schwarz super Lie conformal algebra*
- *Virasoro Lie conformal algebra*
- *Weyl Lie conformal algebra*

AUTHORS:

- Reimundo Heluani (2020-06-15): Initial implementation.

Lie Conformal Algebra Element

AUTHORS:

- Reimundo Heluani (2019-08-09): Initial implementation.

class sage.algebras.lie_conformal_algebras.lie_conformal_algebra_element.
LCAStructureCoefficientsElement

Bases: *LCAStructureCoefficientsElement*

An element of a Lie conformal algebra given by structure coefficients.

class sage.algebras.lie_conformal_algebras.lie_conformal_algebra_element.
LCAStructureCoefficientsElement

Bases: *IndexedFreeModuleElement*

The element class of a Lie conformal algebra with a preferred set of generators.

T ($n=1$)

The n -th derivative of this element.

INPUT:

- n – nonnegative integer (default: 1); how many times to apply T to this element

We use the *divided powers* notation $T^{(j)} = \frac{T^j}{j!}$.

EXAMPLES:

```
sage: Vir = lie_conformal_algebras.Virasoro(QQ)
sage: Vir.inject_variables()
Defining L, C
sage: L.T()
TL
sage: L.T(3)
6*T^(3)L
sage: C.T()
0

sage: R = lie_conformal_algebras.NeveuSchwarz(QQbar); R.inject_variables()
Defining L, G, C
sage: (L + 2*G.T() + 4*C).T(2)
2*T^(2)L + 12*T^(3)G
```

```
>>> from sage.all import *
>>> Vir = lie_conformal_algebras.Virasoro(QQ)
>>> Vir.inject_variables()
Defining L, C
>>> L.T()
```

(continues on next page)

(continued from previous page)

```

TL
>>> L.T(Integer(3))
6*T^(3)L
>>> C.T()
0

>>> R = lie_conformal_algebras.NeveuSchwarz(QQbar); R.inject_variables()
Defining L, G, C
>>> (L + Integer(2)*G.T() + Integer(4)*C).T(Integer(2))
2*T^(2)L + 12*T^(3)G

```

is_monomial()

Whether this element is a monomial.

EXAMPLES:

```

sage: Vir = lie_conformal_algebras.Virasoro(QQ); L = Vir.0
sage: (L + L.T()).is_monomial()
False
sage: L.T().is_monomial()
True

```

```

>>> from sage.all import *
>>> Vir = lie_conformal_algebras.Virasoro(QQ); L = Vir.gen(0)
>>> (L + L.T()).is_monomial()
False
>>> L.T().is_monomial()
True

```

See also

The Category of Lie Conformal Algebras

9.3.2 Implemented examples of Lie Conformal Algebras

Abelian Lie Conformal Algebra

For a commutative ring R and a free R -module M . The *Abelian Lie conformal algebra* generated by M is the free $R[T]$ module generated by M with vanishing λ -brackets.

AUTHORS:

- Reimundo Heluani (2020-06-15): Initial implementation.

```
class sage.algebras.lie_conformal_algebras.abelian_lie_conformal_algebra.AbelianLieConformalAlgebra (
```

Bases: *GradedLieConformalAlgebra*

The Abelian Lie conformal algebra.

INPUT:

- `R` – a commutative ring; the base ring of this Lie conformal algebra
- `ngens` – positive integer (default: 1); the number of generators of this Lie conformal algebra
- `weights` – list of positive rational numbers (default: 1 for each generator); the weights of the generators. The resulting Lie conformal algebra is H -graded.
- `parity` – `None` or a list of 0 or 1 (default: `None`); the parity of the generators. If not `None` the resulting Lie Conformal algebra is a Super Lie conformal algebra
- `names` – tuple of strings or `None` (default: `None`); the list of names of the generators of this algebra.
- `index_set` – an enumerated set or `None` (default: `None`); a set indexing the generators of this Lie conformal algebra

OUTPUT:

The Abelian Lie conformal algebra with generators $a_i, i = 1, \dots, n$ and vanishing λ -brackets, where n is `ngens`.

EXAMPLES:

```
sage: R = lie_conformal_algebras.Abelian(QQ,2); R
The Abelian Lie conformal algebra with generators (a0, a1) over Rational Field
sage: R.inject_variables()
Defining a0, a1
sage: a0.bracket(a1.T(2))
{}
```

```
>>> from sage.all import *
>>> R = lie_conformal_algebras.Abelian(QQ,Integer(2)); R
The Abelian Lie conformal algebra with generators (a0, a1) over Rational Field
>>> R.inject_variables()
Defining a0, a1
>>> a0.bracket(a1.T(Integer(2)))
{}
```

Todo

implement its own class to speed up arithmetics in this case.

Affine Lie Conformal Algebra

The affine Kac-Moody Lie conformal algebra associated to the finite dimensional simple Lie algebra $\mathfrak{g}$. For a commutative ring R , it is the $R[T]$ -module freely generated by $\mathfrak{g}$ plus a central element K satisfying $TK = 0$. The non-vanishing λ -brackets are given by

$$[a_\lambda b] = [a, b] + \lambda(a, b)K,$$

where $a, b \in \mathfrak{g}$ and (a, b) is the normalized form of $\mathfrak{g}$ so that its longest root has square-norm 2.

AUTHORS:

- Reimundo Heluani (2019-08-09): Initial implementation.


```

class sage.algebras.lie_conformal_algebras.affine_lie_conformal_algebra.AffineLieConformalAlgebra (R,
ct,
nan
pre
fix=
bra

```

Bases: *GradedLieConformalAlgebra*

The current or affine Kac-Moody Lie conformal algebra.

INPUT:

- `R` – a commutative Ring; the base ring for this Lie conformal algebra
- `ct` – a `str` or a `CartanType`; the Cartan Type for the corresponding finite dimensional Lie algebra. It must correspond to a simple finite dimensional Lie algebra.
- `names` – list of `str` or `None` (default: `None`); alternative names for the generators. If `None` the generators are labeled by the corresponding root and coroot vectors.
- `prefix` – a `str`; parameter passed to `IndexedGenerators`
- `bracket` – a `str`; parameter passed to `IndexedGenerators`

EXAMPLES:

```

sage: R = lie_conformal_algebras.Affine(QQ, 'A1')
sage: R
The affine Lie conformal algebra of type ['A', 1] over Rational Field
sage: R.an_element()
B[alpha[1]] + B[alphacheck[1]] + B[-alpha[1]] + B['K']

sage: R = lie_conformal_algebras.Affine(QQ, 'A1', names = ('e', 'h', 'f'))
sage: R.inject_variables()
Defining e, h, f, K
sage: Family(e.bracket(f.T(3)))
Finite family {0: 6*T^(3)h, 1: 6*T^(2)h, 2: 6*Th, 3: 6*h, 4: 24*K}

sage: V = lie_conformal_algebras.Affine(QQ, CartanType(["A", 2, 1]))
Traceback (most recent call last):
...
ValueError: only affine algebras of simple finite dimensional Lie algebras are
↳ implemented

```

```

>>> from sage.all import *
>>> R = lie_conformal_algebras.Affine(QQ, 'A1')
>>> R
The affine Lie conformal algebra of type ['A', 1] over Rational Field
>>> R.an_element()
B[alpha[1]] + B[alphacheck[1]] + B[-alpha[1]] + B['K']

>>> R = lie_conformal_algebras.Affine(QQ, 'A1', names = ('e', 'h', 'f'))
>>> R.inject_variables()
Defining e, h, f, K
>>> Family(e.bracket(f.T(Integer(3))))
Finite family {0: 6*T^(3)h, 1: 6*T^(2)h, 2: 6*Th, 3: 6*h, 4: 24*K}

```

(continues on next page)

(continued from previous page)

```
>>> V = lie_conformal_algebras.Affine(QQ, CartanType(["A", Integer(2), Integer(1)]))
Traceback (most recent call last):
...
ValueError: only affine algebras of simple finite dimensional Lie algebras are_
↪ implemented
```

OUTPUT:

The Affine Lie conformal algebra associated with the finite dimensional simple Lie algebra of Cartan type `ct`.

cartan_type()

The Cartan type of this Lie conformal algebra.

EXAMPLES:

```
sage: R = lie_conformal_algebras.Affine(QQ, 'B3')
sage: R
The affine Lie conformal algebra of type ['B', 3] over Rational Field
sage: R.cartan_type()
['B', 3]
```

```
>>> from sage.all import *
>>> R = lie_conformal_algebras.Affine(QQ, 'B3')
>>> R
The affine Lie conformal algebra of type ['B', 3] over Rational Field
>>> R.cartan_type()
['B', 3]
```

Bosonic Ghosts Lie Conformal Algebra

The *Bosonic-ghosts* or $\beta - \gamma$ -system Lie conformal algebra with $2n$ generators is the H-graded Lie conformal algebra generated by $\beta_i, \gamma_i, i = 1, \dots, n$ and a central element K , with non-vanishing λ -brackets:

$$[\beta_i \lambda \gamma_j] = \delta_{ij} K.$$

The generators β_i have degree 1 while the generators γ_i have degree 0.

AUTHORS:

- Reimundo Heluani (2020-06-15): Initial implementation.

class sage.algebras.lie_conformal_algebras.bosonic_ghosts_lie_conformal_algebra.**BosonicGhostsLieConformalAlgebra**

Bases: *GradedLieConformalAlgebra*

The Bosonic ghosts or $\beta - \gamma$ -system Lie conformal algebra.

INPUT:

- `R` – a commutative ring
- `ngens` – an even positive Integer (default: 2); the number of non-central generators of this Lie conformal algebra.
- `names` – list of `str`; alternative names for the generators

- `index_set` – an enumerated set; an indexing set for the generators

OUTPUT:

The Bosonic Ghosts Lie conformal algebra with generators $\beta_i, \gamma_i, i = 1, \dots, n$ and K , where $2n$ is `ngens`.

EXAMPLES:

```
sage: R = lie_conformal_algebras.BosonicGhosts(QQ); R
The Bosonic ghosts Lie conformal algebra with generators (beta, gamma, K) over
↳Rational Field
sage: R.inject_variables(); beta.bracket(gamma)
Defining beta, gamma, K
{0: K}
sage: beta.degree()
1
sage: gamma.degree()
0

sage: R = lie_conformal_algebras.BosonicGhosts(QQbar, ngens = 4, names = 'abcd'); R
↳R
The Bosonic ghosts Lie conformal algebra with generators (a, b, c, d, K) over
↳Algebraic Field
sage: R.structure_coefficients()
Finite family {('a', 'c'): ((0, K),), ('b', 'd'): ((0, K),), ('c', 'a'): ((0, -
↳K),), ('d', 'b'): ((0, -K),)}
```

```
>>> from sage.all import *
>>> R = lie_conformal_algebras.BosonicGhosts(QQ); R
The Bosonic ghosts Lie conformal algebra with generators (beta, gamma, K) over
↳Rational Field
>>> R.inject_variables(); beta.bracket(gamma)
Defining beta, gamma, K
{0: K}
>>> beta.degree()
1
>>> gamma.degree()
0

>>> R = lie_conformal_algebras.BosonicGhosts(QQbar, ngens = Integer(4), names =
↳'abcd'); R
The Bosonic ghosts Lie conformal algebra with generators (a, b, c, d, K) over
↳Algebraic Field
>>> R.structure_coefficients()
Finite family {('a', 'c'): ((0, K),), ('b', 'd'): ((0, K),), ('c', 'a'): ((0, -
↳K),), ('d', 'b'): ((0, -K),)}
```

Fermionic Ghosts Super Lie Conformal Algebra

The *Fermionic-ghosts* or *b-c* system super Lie conformal algebra with $2n$ generators is the $\mathbb{H}$ -graded super Lie conformal algebra generated by odd vectors $b_i, c_i, i = 1, \dots, n$ and a central element K , with non-vanishing λ -brackets:

$$[b_i \lambda c_j] = \delta_{ij} K.$$

The generators b_i have degree 1 while the generators c_i have degree 0.

AUTHORS:

- Reimundo Heluani (2020-06-03): Initial implementation.

class sage.algebras.lie_conformal_algebras.fermionic_ghosts_lie_conformal_algebra.FermionicGhostsLie

Bases: *GradedLieConformalAlgebra*

The Fermionic ghosts or *bc*-system super Lie conformal algebra.

INPUT:

- *R* – a commutative ring; the base ring of this Lie conformal algebra
- *ngens* – an even positive Integer (default: 2); the number of non-central generators of this Lie conformal algebra
- *names* – tuple of strings; alternative names for the generators
- *index_set* – an enumerated set; alternative indexing set for the generators

OUTPUT:

The Fermionic Ghosts super Lie conformal algebra with generators $b_i, c_i, i = 1, \dots, n$ and K where $2n$ is *ngens*.

EXAMPLES:

```
sage: R = lie_conformal_algebras.FermionicGhosts(QQ); R
The Fermionic ghosts Lie conformal algebra with generators (b, c, K) over
↳Rational Field
sage: R.inject_variables()
Defining b, c, K
sage: b.bracket(c) == c.bracket(b)
True
sage: b.degree()
1
sage: c.degree()
0
sage: R.category()
Category of H-graded super finitely generated Lie conformal algebras with basis
↳over Rational Field

sage: R = lie_conformal_algebras.FermionicGhosts(QQbar, ngens=4, names = 'abcd'); R
The Fermionic ghosts Lie conformal algebra with generators (a, b, c, d, K) over
↳Algebraic Field
sage: R.structure_coefficients()
Finite family {'a', 'c': ((0, K),), ('b', 'd'): ((0, K),), ('c', 'a'): ((0,
↳K),), ('d', 'b'): ((0, K),)}
```

```
>>> from sage.all import *
>>> R = lie_conformal_algebras.FermionicGhosts(QQ); R
The Fermionic ghosts Lie conformal algebra with generators (b, c, K) over
↳Rational Field
>>> R.inject_variables()
Defining b, c, K
```

(continues on next page)

(continued from previous page)

```

>>> b.bracket(c) == c.bracket(b)
True
>>> b.degree()
1
>>> c.degree()
0
>>> R.category()
Category of H-graded super finitely generated Lie conformal algebras with basis
↳ over Rational Field

>>> R = lie_conformal_algebras.FermionicGhosts(QQbar, ngens=Integer(4), names =
↳ 'abcd');R
The Fermionic ghosts Lie conformal algebra with generators (a, b, c, d, K) over
↳ Algebraic Field
>>> R.structure_coefficients()
Finite family {('a', 'c'): ((0, K),), ('b', 'd'): ((0, K),), ('c', 'a'): ((0,
↳ K),), ('d', 'b'): ((0, K),)}

```

Free Bosons Lie Conformal Algebra

Given an R -module M with a symmetric, bilinear pairing $(\cdot, \cdot) : M \otimes_R M \rightarrow R$. The *Free Bosons* Lie conformal algebra associated to this datum is the free $R[T]$ -module generated by M plus a central vector K satisfying $TK = 0$. The remaining λ -brackets are given by:

$$[v_\lambda w] = \lambda(v, w)K,$$

where $v, w \in M$.

This is an H-graded Lie conformal algebra where every generator $v \in M$ has degree 1.

AUTHORS:

- Reimundo Heluani (2019-08-09): Initial implementation.

class sage.algebras.lie_conformal_algebras.free_bosons_lie_conformal_algebra.**FreeBosonsLieConformalA**

Bases: *GradedLieConformalAlgebra*

The Free Bosons Lie conformal algebra.

INPUT:

- R – a commutative ring
- ngens – a positive Integer (default: 1); the number of non-central generators of this Lie conformal algebra.
- gram_matrix – a symmetric square matrix with coefficients in R (default: `identity_matrix(ngens)`); the Gram matrix of the inner product
- names – tuple of strings; alternative names for the generators
- index_set – an enumerated set; alternative indexing set for the generators

OUTPUT:

The Free Bosons Lie conformal algebra with generators

$\alpha_i, i = 1, \dots, n$ and λ -brackets

$$[\alpha_i \lambda \alpha_j] = \lambda M_{ij} K,$$

where n is the number of generators `ngens` and M is the `gram_matrix`. This Lie conformal algebra is H -graded where every generator has conformal weight 1.

EXAMPLES:

```
sage: R = lie_conformal_algebras.FreeBosons(AA); R
The free Bosons Lie conformal algebra with generators (alpha, K) over Algebraic_
↪Real Field
sage: R.inject_variables()
Defining alpha, K
sage: alpha.bracket(alpha)
{1: K}
sage: M = identity_matrix(QQ,2); R = lie_conformal_algebras.FreeBosons(QQ,gram_
↪matrix=M, names='alpha,beta'); R
The free Bosons Lie conformal algebra with generators (alpha, beta, K) over_
↪Rational Field
sage: R.inject_variables(); alpha.bracket(beta)
Defining alpha, beta, K
{}
sage: alpha.bracket(alpha)
{1: K}
sage: R = lie_conformal_algebras.FreeBosons(QQbar, ngens=3); R
The free Bosons Lie conformal algebra with generators (alpha0, alpha1, alpha2, K)_
↪over Algebraic Field
```

```
>>> from sage.all import *
>>> R = lie_conformal_algebras.FreeBosons(AA); R
The free Bosons Lie conformal algebra with generators (alpha, K) over Algebraic_
↪Real Field
>>> R.inject_variables()
Defining alpha, K
>>> alpha.bracket(alpha)
{1: K}
>>> M = identity_matrix(QQ,Integer(2)); R = lie_conformal_algebras.FreeBosons(QQ,
↪gram_matrix=M, names='alpha,beta'); R
The free Bosons Lie conformal algebra with generators (alpha, beta, K) over_
↪Rational Field
>>> R.inject_variables(); alpha.bracket(beta)
Defining alpha, beta, K
{}
>>> alpha.bracket(alpha)
{1: K}
>>> R = lie_conformal_algebras.FreeBosons(QQbar, ngens=Integer(3)); R
The free Bosons Lie conformal algebra with generators (alpha0, alpha1, alpha2, K)_
↪over Algebraic Field
```

gram_matrix()

The Gram matrix that specifies the λ -brackets of the generators.

EXAMPLES:

```
sage: R = lie_conformal_algebras.FreeBosons(QQ,ngens=2);
sage: R.gram_matrix()
[1 0]
[0 1]
```

```
>>> from sage.all import *
>>> R = lie_conformal_algebras.FreeBosons(QQ,ngens=Integer(2));
>>> R.gram_matrix()
[1 0]
[0 1]
```

Free Fermions Super Lie Conformal Algebra.

Given an R -module M with a skew-symmetric, bilinear pairing $\langle \cdot, \cdot \rangle : M \otimes_R M \rightarrow R$. The *Free Fermions* super Lie conformal algebra associated to this datum is the free $R[T]$ -super module generated by ΠM (a purely odd copy of M) plus a central vector K satisfying $TK = 0$. The remaining λ -brackets are given by:

$$[v_\lambda w] = \langle v, w \rangle K,$$

where $v, w \in M$.

This is an H -graded Lie conformal algebra where every generator $v \in M$ has degree $1/2$.

AUTHORS:

- Reimundo Heluani (2020-06-03): Initial implementation.

```
class sage.algebras.lie_conformal_algebras.free_fermions_lie_conformal_algebra.FreeFermionsLieConformal
```

Bases: *GradedLieConformalAlgebra*

The Free Fermions Super Lie conformal algebra.

INPUT:

- R – a commutative ring
- $ngens$ – a positive Integer (default: 1); the number of non-central generators of this Lie conformal algebra
- $gram\_matrix$ – a symmetric square matrix with coefficients in R (default: `identity_matrix(ngens)`); the Gram matrix of the inner product

OUTPUT:

The Free Fermions Lie conformal algebra with generators

$\psi_i, i = 1, \dots, n$ and λ -brackets

$$[\psi_i \lambda \psi_j] = M_{ij} K,$$

where n is the number of generators $ngens$ and M is the `gram_matrix`. This super Lie conformal algebra is H -graded where every generator has degree $1/2$.

EXAMPLES:

```

sage: R = lie_conformal_algebras.FreeFermions(QQbar); R
The free Fermions super Lie conformal algebra with generators (psi, K) over
↳Algebraic Field
sage: R.inject_variables()
Defining psi, K
sage: psi.bracket(psi)
{0: K}

sage: R = lie_conformal_algebras.FreeFermions(QQbar, gram_matrix=Matrix([[0, 1], [1,
↳0]])); R
The free Fermions super Lie conformal algebra with generators (psi_0, psi_1, K)
↳over Algebraic Field
sage: R.inject_variables()
Defining psi_0, psi_1, K
sage: psi_0.bracket(psi_1)
{0: K}
sage: psi_0.degree()
1/2
sage: R.category()
Category of H-graded super finitely generated Lie conformal algebras with basis
↳over Algebraic Field

```

```

>>> from sage.all import *
>>> R = lie_conformal_algebras.FreeFermions(QQbar); R
The free Fermions super Lie conformal algebra with generators (psi, K) over
↳Algebraic Field
>>> R.inject_variables()
Defining psi, K
>>> psi.bracket(psi)
{0: K}

>>> R = lie_conformal_algebras.FreeFermions(QQbar, gram_matrix=Matrix([[Integer(0),
↳Integer(1)], [Integer(1), Integer(0)]])); R
The free Fermions super Lie conformal algebra with generators (psi_0, psi_1, K)
↳over Algebraic Field
>>> R.inject_variables()
Defining psi_0, psi_1, K
>>> psi_0.bracket(psi_1)
{0: K}
>>> psi_0.degree()
1/2
>>> R.category()
Category of H-graded super finitely generated Lie conformal algebras with basis
↳over Algebraic Field

```

gram_matrix()

The Gram matrix that specifies the λ -brackets of the generators.

EXAMPLES:

```

sage: R = lie_conformal_algebras.FreeFermions(QQ, ngens=2);
sage: R.gram_matrix()
[1 0]

```

(continues on next page)

(continued from previous page)

```
[0 1]
```

```
>>> from sage.all import *
>>> R = lie_conformal_algebras.FreeFermions(QQ,ngens=Integer(2));
>>> R.gram_matrix()
[1 0]
[0 1]
```

N=2 Super Lie Conformal Algebra

The $N = 2$ super Lie conformal algebra is an extension of the Virasoro Lie conformal algebra (with generators L, C) by an even generator J which is primary of conformal weight 1 and two odd generators G_1, G_2 which are primary of conformal weight $3/2$. The remaining λ -brackets are given by:

$$\begin{aligned} [J_\lambda J] &= \frac{\lambda}{3} C, \\ [J_\lambda G_1] &= G_1, \\ [J_\lambda G_2] &= -G_2, \\ [G_{1\lambda} G_1] &= [G_{2\lambda} G_2] = 0, \\ [G_{1\lambda} G_2] &= L + \frac{1}{2} T J + \lambda J + \frac{\lambda^2}{6} C. \end{aligned}$$

AUTHORS:

- Reimundo Heluani (2020-06-03): Initial implementation.

class sage.algebras.lie_conformal_algebras.n2_lie_conformal_algebra.N2LieConformalAlgebra(R)

Bases: *GradedLieConformalAlgebra*

The N=2 super Lie conformal algebra.

INPUT:

- R – a commutative ring; the base ring of this super Lie conformal algebra

EXAMPLES:

```
sage: x = polygen(ZZ, 'x')
sage: F.<x> = NumberField(x^2 - 2)
sage: R = lie_conformal_algebras.N2(F); R
The N=2 super Lie conformal algebra over Number Field in x with defining_
↳ polynomial x^2 - 2
sage: R.inject_variables()
Defining L, J, G1, G2, C
sage: G1.bracket(G2)
{0: L + 1/2*TJ, 1: J, 2: 1/3*C}
sage: G2.bracket(G1)
{0: L - 1/2*TJ, 1: -J, 2: 1/3*C}
sage: G1.degree()
3/2
sage: J.degree()
1
```

```

>>> from sage.all import *
>>> x = polygen(ZZ, 'x')
>>> F = NumberField(x**Integer(2) - Integer(2), names=('x',)); (x,) = F._first_
↳ngens(1)
>>> R = lie_conformal_algebras.N2(F); R
The N=2 super Lie conformal algebra over Number Field in x with defining_
↳polynomial x^2 - 2
>>> R.inject_variables()
Defining L, J, G1, G2, C
>>> G1.bracket(G2)
{0: L + 1/2*TJ, 1: J, 2: 1/3*C}
>>> G2.bracket(G1)
{0: L - 1/2*TJ, 1: -J, 2: 1/3*C}
>>> G1.degree()
3/2
>>> J.degree()
1

```

The topological twist is a Virasoro vector with central charge 0:

```

sage: L2 = L - 1/2*J.T()
sage: L2.bracket(L2) == {0: L2.T(), 1: 2*L2}
True

```

```

>>> from sage.all import *
>>> L2 = L - Integer(1)/Integer(2)*J.T()
>>> L2.bracket(L2) == {Integer(0): L2.T(), Integer(1): Integer(2)*L2}
True

```

The sum of the fermions is a generator of the Neveu-Schwarz Lie conformal algebra:

```

sage: G = (G1 + G2)
sage: G.bracket(G)
{0: 2*L, 2: 2/3*C}

```

```

>>> from sage.all import *
>>> G = (G1 + G2)
>>> G.bracket(G)
{0: 2*L, 2: 2/3*C}

```

Neveu-Schwarz Super Lie Conformal Algebra

The $N = 1$ or *Neveu-Schwarz* super Lie conformal algebra is a super extension of the Virasoro Lie conformal algebra with generators L and C by an odd primary generator G of conformal weight $3/2$. The remaining λ -bracket is given by:

$$[G_\lambda G] = 2L + \frac{\lambda^2}{3}C.$$

AUTHORS:

- Reimundo Heluani (2020-06-03): Initial implementation.

```
class sage.algebras.lie_conformal_algebras.neveu_schwarz_lie_conformal_algebra.NeveuSchwarzLieConformal
```

Bases: *GradedLieConformalAlgebra*

The Neveu-Schwarz super Lie conformal algebra.

INPUT:

- R – a commutative Ring; the base ring of this Lie conformal algebra

EXAMPLES:

```
sage: R = lie_conformal_algebras.NeuSchwarz(AA); R
The Neveu-Schwarz super Lie conformal algebra over Algebraic Real Field
sage: R.structure_coefficients()
Finite family {('G', 'G'): ((0, 2*L), (2, 2/3*C)), ('G', 'L'): ((0, 1/2*TG), (1, ↪
↪ 3/2*G)), ('L', 'G'): ((0, TG), (1, 3/2*G)), ('L', 'L'): ((0, TL), (1, 2*L), ↪
↪ (3, 1/2*C))}
sage: R.inject_variables()
Defining L, G, C
sage: G.nproduct(G, 0)
2*L
sage: G.degree()
3/2
```

```
>>> from sage.all import *
>>> R = lie_conformal_algebras.NeuSchwarz(AA); R
The Neveu-Schwarz super Lie conformal algebra over Algebraic Real Field
>>> R.structure_coefficients()
Finite family {('G', 'G'): ((0, 2*L), (2, 2/3*C)), ('G', 'L'): ((0, 1/2*TG), (1, ↪
↪ 3/2*G)), ('L', 'G'): ((0, TG), (1, 3/2*G)), ('L', 'L'): ((0, TL), (1, 2*L), ↪
↪ (3, 1/2*C))}
>>> R.inject_variables()
Defining L, G, C
>>> G.nproduct(G, Integer(0))
2*L
>>> G.degree()
3/2
```

Virasoro Lie Conformal Algebra

The Virasoro Lie conformal algebra is generated by L and a central element C . The λ -brackets are given by:

$$[L_{\lambda}L] = TL + 2\lambda L + \frac{\lambda^3}{12}C.$$

It is an H-graded Lie conformal algebra with L of degree 2.

AUTHORS:

- Reimundo Heluani (2019-08-09): Initial implementation.

```
class sage.algebras.lie_conformal_algebras.virasoro_lie_conformal_algebra.VirasoroLieConformalAlgebra
```

Bases: *GradedLieConformalAlgebra*

The Virasoro Lie Conformal algebra over R .

INPUT:

- R – a commutative ring; behaviour is undefined if R is not a Field of characteristic zero

EXAMPLES:

```

sage: Vir = lie_conformal_algebras.Virasoro(QQ)
sage: Vir.category()
Category of H-graded finitely generated Lie conformal algebras with basis over
↳Rational Field
sage: Vir.inject_variables()
Defining L, C
sage: L.bracket(L)
{0: TL, 1: 2*L, 3: 1/2*C}

```

```

>>> from sage.all import *
>>> Vir = lie_conformal_algebras.Virasoro(QQ)
>>> Vir.category()
Category of H-graded finitely generated Lie conformal algebras with basis over
↳Rational Field
>>> Vir.inject_variables()
Defining L, C
>>> L.bracket(L)
{0: TL, 1: 2*L, 3: 1/2*C}

```

Weyl Lie Conformal Algebra

Given a commutative ring R , a free R -module M and a non-degenerate, skew-symmetric, bilinear pairing $\langle \cdot, \cdot \rangle : M \otimes_R M \rightarrow R$. The *Weyl Lie conformal algebra* associated to this datum is the free $R[T]$ -module generated by M plus a central vector K . The non-vanishing λ -brackets are given by:

$$[v_\lambda w] = \langle v, w \rangle K.$$

This is not an H-graded Lie conformal algebra. The choice of a Lagrangian decomposition $M = L \oplus L^*$ determines an H-graded structure. For this H-graded Lie conformal algebra see the [Bosonic Ghosts Lie conformal algebra](#)

AUTHORS:

- Reimundo Heluani (2019-08-09): Initial implementation.

```

class sage.algebras.lie_conformal_algebras.weyl_lie_conformal_algebra.WeylLieConformalAlgebra(R,
ngens=N
gram_ma
trix=Non
names=N
in-
dex_set=

```

Bases: *LieConformalAlgebraWithStructureCoefficients*

The Weyl Lie conformal algebra.

INPUT:

- R – a commutative ring; the base ring of this Lie conformal algebra
- $ngens$ – an even positive Integer (default: 2); the number of non-central generators of this Lie conformal algebra
- $gram\_matrix$ – a matrix (default: None); a non-singular skew-symmetric square matrix with coefficients in R
- $names$ – list or tuple of strings; alternative names for the generators
- $index\_set$ – an enumerated set; alternative indexing set for the generators

OUTPUT:

The Weyl Lie conformal algebra with generators

$\alpha_i, i = 1, \dots, n_{gens}$ and λ -brackets

$$[\alpha_i \lambda \alpha_j] = M_{ij} K,$$

where M is the `gram_matrix` above.

Note

The returned Lie conformal algebra is not H -graded. For a related H -graded Lie conformal algebra see [*BosonicGhostsLieConformalAlgebra*](#).

EXAMPLES:

```
sage: lie_conformal_algebras.Weyl(QQ)
The Weyl Lie conformal algebra with generators (alpha0, alpha1, K) over Rational_
↳Field
sage: R = lie_conformal_algebras.Weyl(QQbar, gram_matrix=Matrix(QQ, [[0,1],[-1,
↳0]]), names = ('a', 'b'))
sage: R.inject_variables()
Defining a, b, K
sage: a.bracket(b)
{0: K}
sage: b.bracket(a)
{0: -K}

sage: R = lie_conformal_algebras.Weyl(QQbar, ngens=4)
sage: R.gram_matrix()
[ 0  0| 1  0]
[ 0  0| 0  1]
[-----+-----]
[-1  0| 0  0]
[ 0 -1| 0  0]
sage: R.inject_variables()
Defining alpha0, alpha1, alpha2, alpha3, K
sage: alpha0.bracket(alpha2)
{0: K}

sage: R = lie_conformal_algebras.Weyl(QQ); R.category()
Category of finitely generated Lie conformal algebras with basis over Rational_
↳Field
sage: R in LieConformalAlgebras(QQ).Graded()
False
sage: R.inject_variables()
Defining alpha0, alpha1, K
sage: alpha0.degree()
Traceback (most recent call last):
...
AttributeError: 'WeylLieConformalAlgebra_with_category.element_class' object has_
↳no attribute 'degree'...
```

```

>>> from sage.all import *
>>> lie_conformal_algebras.Weyl(QQ)
The Weyl Lie conformal algebra with generators (alpha0, alpha1, K) over Rational_
↳Field
>>> R = lie_conformal_algebras.Weyl(QQbar, gram_matrix=Matrix(QQ, [[Integer(0),
↳Integer(1)], [-Integer(1), Integer(0)]]), names = ('a', 'b'))
>>> R.inject_variables()
Defining a, b, K
>>> a.bracket(b)
{0: K}
>>> b.bracket(a)
{0: -K}

>>> R = lie_conformal_algebras.Weyl(QQbar, ngens=Integer(4))
>>> R.gram_matrix()
[ 0  0| 1  0]
[ 0  0| 0  1]
[-----+-----]
[-1  0| 0  0]
[ 0 -1| 0  0]
>>> R.inject_variables()
Defining alpha0, alpha1, alpha2, alpha3, K
>>> alpha0.bracket(alpha2)
{0: K}

>>> R = lie_conformal_algebras.Weyl(QQ); R.category()
Category of finitely generated Lie conformal algebras with basis over Rational_
↳Field
>>> R in LieConformalAlgebras(QQ).Graded()
False
>>> R.inject_variables()
Defining alpha0, alpha1, K
>>> alpha0.degree()
Traceback (most recent call last):
...
AttributeError: 'WeylLieConformalAlgebra_with_category.element_class' object has_
↳no attribute 'degree'...

```

gram_matrix()

The Gram matrix that specifies the λ -brackets of the generators.

EXAMPLES:

```

sage: R = lie_conformal_algebras.Weyl(QQbar, ngens=4)
sage: R.gram_matrix()
[ 0  0| 1  0]
[ 0  0| 0  1]
[-----+-----]
[-1  0| 0  0]
[ 0 -1| 0  0]

```

```

>>> from sage.all import *
>>> R = lie_conformal_algebras.Weyl(QQbar, ngens=Integer(4))

```

(continues on next page)

(continued from previous page)

```
>>> R.gram_matrix()
[ 0  0| 1  0]
[ 0  0| 0  1]
[-----+-----]
[-1  0| 0  0]
[ 0 -1| 0  0]
```

9.3.3 See also

Finitely and Freely Generated Lie Conformal Algebras.

AUTHORS:

- Reimundo Heluani (2019-08-09): Initial implementation.

class `sage.algebras.lie_conformal_algebras.finitely_freely_generated_lca.FinitelyFreelyGeneratedLCA` (`A`)

Bases: `FreelyGeneratedLieConformalAlgebra`

Abstract base class for finitely generated Lie conformal algebras.

This class provides minimal functionality, simply sets the number of generators.

central_elements()

The central elements of this Lie conformal algebra.

EXAMPLES:

```
sage: R = lie_conformal_algebras.NeveuSchwarz(QQ); R.central_elements()
(C,)
```

```
>>> from sage.all import *
>>> R = lie_conformal_algebras.NeveuSchwarz(QQ); R.central_elements()
(C,)
```

gens()

The generators for this Lie conformal algebra.

OUTPUT:

This method returns a tuple with the (finite) generators of this Lie conformal algebra.

EXAMPLES:

```
sage: Vir = lie_conformal_algebras.Virasoro(QQ);
sage: Vir.gens()
(L, C)
```

```
>>> from sage.all import *
>>> Vir = lie_conformal_algebras.Virasoro(QQ);
>>> Vir.gens()
(L, C)
```

See also

`lie_conformal_algebra_generators`

ngens()

The number of generators of this Lie conformal algebra.

EXAMPLES:

```
sage: Vir = lie_conformal_algebras.Virasoro(QQ); Vir.ngens()
2
sage: V = lie_conformal_algebras.Affine(QQ, 'A1'); V.ngens()
4
```

```
>>> from sage.all import *
>>> Vir = lie_conformal_algebras.Virasoro(QQ); Vir.ngens()
2
>>> V = lie_conformal_algebras.Affine(QQ, 'A1'); V.ngens()
4
```

Freely Generated Lie Conformal Algebras

AUTHORS:

- Reimundo Heluani (2019-08-09): Initial implementation


```
class sage.algebras.lie_conformal_algebras.freely_generated_lie_conformal_algebra.FreelyGeneratedLie
```

Bases: *LieConformalAlgebraWithBasis*

Base class for a central extension of a freely generated Lie conformal algebra.

This class provides minimal functionality, it sets up the family of Lie conformal algebra generators.

Note

We now only accept direct sums of free modules plus some central generators C_i such that $TC_i = 0$.

central_elements()

The central generators of this Lie conformal algebra.

EXAMPLES:

```
sage: Vir = lie_conformal_algebras.Virasoro(QQ)
sage: Vir.central_elements()
(C,)
sage: V = lie_conformal_algebras.Affine(QQ, 'A1')
sage: V.central_elements()
(B['K'],)
```

```
>>> from sage.all import *
>>> Vir = lie_conformal_algebras.Virasoro(QQ)
>>> Vir.central_elements()
(C,)
>>> V = lie_conformal_algebras.Affine(QQ, 'A1')
>>> V.central_elements()
(B['K'],)
```

lie_conformal_algebra_generators()

The generators of this Lie conformal algebra.

OUTPUT: a (possibly infinite) family of generators (as an $R[T]$ -module) of this Lie conformal algebra.

EXAMPLES:

```

sage: Vir = lie_conformal_algebras.Virasoro(QQ)
sage: Vir.lie_conformal_algebra_generators()
(L, C)
sage: V = lie_conformal_algebras.Affine(QQ, 'A1')
sage: V.lie_conformal_algebra_generators()
(B[alpha[1]], B[alphacheck[1]], B[-alpha[1]], B['K'])

```

```

>>> from sage.all import *
>>> Vir = lie_conformal_algebras.Virasoro(QQ)
>>> Vir.lie_conformal_algebra_generators()
(L, C)
>>> V = lie_conformal_algebras.Affine(QQ, 'A1')
>>> V.lie_conformal_algebra_generators()
(B[alpha[1]], B[alphacheck[1]], B[-alpha[1]], B['K'])

```

Graded Lie Conformal Algebras

A (super) Lie conformal algebra V is called H -graded if there exists a decomposition $V = \oplus_n V_n$ such that the λ -bracket is graded of degree -1 , that is for homogeneous elements $a \in V_p, b \in V_q$ with λ -brackets:

$$[a_\lambda b] = \sum \frac{\lambda^n}{n!} c_n,$$

we have $c_n \in V_{p+q-n-1}$. This situation arises typically when V has a vector $L \in V$ that generates the Virasoro Lie conformal algebra. Such that for every $a \in V$ we have

$$[L_\lambda a] = Ta + \lambda \Delta_a a + O(\lambda^2).$$

In this situation V is graded by the eigenvalues Δ_a of $L_{(1)}$, the (1)-th product with L . When the higher order terms $O(\lambda^2)$ vanish we say that a is a *primary vector of conformal weight* or degree Δ_a .

Note

Although arbitrary gradings are allowed, many of the constructions we implement in these classes work only for positive rational gradings.

AUTHORS:

- Reimundo Heluani (2019-08-09): Initial implementation.

```
class sage.algebras.lie_conformal_algebras.graded_lie_conformal_algebra.GradedLieConformalAlgebra(R,
```

s_c
eff,
in-
dex
cen
tral
e-
men
cat-
e-
gor
pre
fix=
nan
la-
tex
par
ity=
wei
**/

Bases: *LieConformalAlgebraWithStructureCoefficients*

An H-Graded Lie conformal algebra.

INPUT:

- `R` – a commutative ring (default: `None`); the base ring of this Lie conformal algebra. Behaviour is undefined if it is not a field of characteristic zero
- `s_coeff` – dictionary (default: `None`); as in the input of *LieConformalAlgebra*
- `names` – tuple of strings (default: `None`); as in the input of *LieConformalAlgebra*
- `central_elements` – tuple of strings (default: `None`); as in the input of *LieConformalAlgebra*
- `index_set` – enumerated set (default: `None`); as in the input of *LieConformalAlgebra*
- `weights` – tuple of nonnegative rational numbers (default: tuple of 1); a list of degrees for this Lie conformal algebra. This tuple needs to have the same cardinality as `index_set` or `names`. Central elements are assumed to have weight 0.
- `category` – the category that this Lie conformal algebra belongs to
- `parity` – tuple of 0 or 1 (default: tuple of 0); a tuple specifying the parity of each non-central generator

EXAMPLES:

```
sage: bosondict = {('a', 'a'):{1:{('K', 0):1}}}
sage: R = LieConformalAlgebra(QQ, bosondict, names=('a',), central_elements=('K',),
↪ weights=(1,))
sage: R.inject_variables()
Defining a, K
sage: a.T(3).degree()
4
sage: K.degree()
0
sage: R.category()
```

(continues on next page)

(continued from previous page)

```
Category of H-graded finitely generated Lie conformal algebras with basis over_
↪Rational Field
```

```
>>> from sage.all import *
>>> bosondict = {'a','a':{Integer(1):{'K',Integer(0)}:Integer(1)}}
>>> R = LieConformalAlgebra(QQ,bosondict,names=('a',),central_elements=('K',),
↪weights=(Integer(1),))
>>> R.inject_variables()
Defining a, K
>>> a.T(Integer(3)).degree()
4
>>> K.degree()
0
>>> R.category()
Category of H-graded finitely generated Lie conformal algebras with basis over_
↪Rational Field
```

Lie Conformal Algebras With Basis

AUTHORS:

- Reimundo Heluani (2019-08-09): Initial implementation

```
class sage.algebras.lie_conformal_algebras.lie_conformal_algebra_with_basis.LieConformalAlgebraWithB
```

Bases: `CombinatorialFreeModule`

Abstract base class for a Lie conformal algebra with a preferred basis.

This class provides no functionality, it simply passes the arguments to `CombinatorialFreeModule`.

EXAMPLES:

```
sage: R = lie_conformal_algebras.Virasoro(QQbar);R
The Virasoro Lie conformal algebra over Algebraic Field
```

```
>>> from sage.all import *
>>> R = lie_conformal_algebras.Virasoro(QQbar);R
The Virasoro Lie conformal algebra over Algebraic Field
```

Lie Conformal Algebras With Structure Coefficients

AUTHORS:

- Reimundo Heluani (2019-08-09): Initial implementation.

```
class sage.algebras.lie_conformal_algebras.lie_conformal_algebra_with_structure_coefs.LieConformalAl
```

Bases: *FinitelyFreelyGeneratedLCA*

A Lie conformal algebra with a set of specified structure coefficients.

INPUT:

- `R` – a ring (default: `None`); the base ring of this Lie conformal algebra. Behaviour is undefined if it is not a field of characteristic zero.
- `s_coeff` – dictionary (default: `None`); a dictionary containing the λ brackets of the generators of this Lie conformal algebra. The family encodes a dictionary whose keys are pairs of either names or indices of the generators and the values are themselves dictionaries. For a pair of generators a and b , the value of `s_coeff[('a', 'b')]` is a dictionary whose keys are positive integer numbers and the corresponding value for the key j is a dictionary itself representing the j -th product $a_{(j)}b$. Thus, for a positive integer number j , the value of `s_coeff[('a', 'b')][j]` is a dictionary whose entries are pairs `('c', n)` where `'c'` is the name of a generator and n is a positive number. The value for this key is the coefficient of $\frac{T^n}{n!}c$ in $a_{(j)}b$. For example the `s_coeff` for the *Virasoro* Lie conformal algebra is:

```
{('L', 'L'):{0:{('L', 1):1}, 1:{('L', 0):2}, 3:{('C', 0):1/2}}}
```

Do not include central elements in this dictionary. Also, if the key `('a', 'b')` is present, there is no need to include `('b', 'a')` as it is defined by skew-symmetry. Any missing pair (besides the ones defined by skew-symmetry) is assumed to have vanishing λ -bracket.

- `names` – tuple of strings (default: `None`); the list of names for generators of this Lie conformal algebra. Do not include central elements in this list.
- `central_elements` – tuple of strings (default: `None`); a list of names for central elements of this Lie conformal algebra

- `index_set` – enumerated set (default: `None`); an indexing set for the generators of this Lie conformal algebra. Do not include central elements in this list.
- `parity` – tuple of 0 or 1 (default: tuple of 0); a tuple specifying the parity of each non-central generator

EXAMPLES:

- We construct the $\beta - \gamma$ system by directly giving the λ -brackets of the generators:

```
sage: betagamma_dict = {'b','a':{0:{('K',0):1}}}
sage: V = LieConformalAlgebra(QQ, betagamma_dict, names=('a','b'),
....:      weights=(1,0), central_elements=('K',))
sage: V.category()
Category of H-graded finitely generated Lie conformal algebras
with basis over Rational Field
sage: V.inject_variables()
Defining a, b, K
sage: a.bracket(b)
{0: -K}
```

```
>>> from sage.all import *
>>> betagamma_dict = {'b','a':{Integer(0):{('K',Integer(0)):Integer(1)}}}
>>> V = LieConformalAlgebra(QQ, betagamma_dict, names=('a','b'),
...      weights=(Integer(1),Integer(0)), central_elements=('K',))
>>> V.category()
Category of H-graded finitely generated Lie conformal algebras
with basis over Rational Field
>>> V.inject_variables()
Defining a, b, K
>>> a.bracket(b)
{0: -K}
```

- We construct the centerless Virasoro Lie conformal algebra:

```
sage: virdict = {('L','L'):{0:{('L',1):1}, 1:{('L',0): 2}}}
sage: R = LieConformalAlgebra(QQbar, virdict, names='L')
sage: R.inject_variables()
Defining L
sage: L.bracket(L)
{0: TL, 1: 2*L}
```

```
>>> from sage.all import *
>>> virdict = {('L','L'):{Integer(0):{('L',Integer(1)):Integer(1)},
↪Integer(1):{('L',Integer(0)): Integer(2)}}}
>>> R = LieConformalAlgebra(QQbar, virdict, names='L')
>>> R.inject_variables()
Defining L
>>> L.bracket(L)
{0: TL, 1: 2*L}
```

- The construction checks that skew-symmetry is violated:

```
sage: wrongdict = {('L','L'):{0:{('L',1):2}, 1:{('L',0): 2}}}
sage: LieConformalAlgebra(QQbar, wrongdict, names='L')
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: two distinct values given for one and the same bracket. Skew-
↪symmetry is not satisfied?
```

```
>>> from sage.all import *
>>> wrongdict = {('L', 'L'):{Integer(0):{('L', Integer(1)):Integer(2)},
↪Integer(1):{('L', Integer(0)): Integer(2)}}}
>>> LieConformalAlgebra(QQbar, wrongdict, names='L')
Traceback (most recent call last):
...
ValueError: two distinct values given for one and the same bracket. Skew-
↪symmetry is not satisfied?
```

structure_coefficients()

The structure coefficients of this Lie conformal algebra.

EXAMPLES:

```
sage: Vir = lie_conformal_algebras.Virasoro(AA)
sage: Vir.structure_coefficients()
Finite family {('L', 'L'): ((0, TL), (1, 2*L), (3, 1/2*C))}

sage: lie_conformal_algebras.NeuSchwarz(QQ).structure_coefficients()
Finite family {('G', 'G'): ((0, 2*L), (2, 2/3*C)),
('G', 'L'): ((0, 1/2*TG), (1, 3/2*G)),
('L', 'G'): ((0, TG), (1, 3/2*G)),
('L', 'L'): ((0, TL), (1, 2*L), (3, 1/2*C))}
```

```
>>> from sage.all import *
>>> Vir = lie_conformal_algebras.Virasoro(AA)
>>> Vir.structure_coefficients()
Finite family {('L', 'L'): ((0, TL), (1, 2*L), (3, 1/2*C))}

>>> lie_conformal_algebras.NeuSchwarz(QQ).structure_coefficients()
Finite family {('G', 'G'): ((0, 2*L), (2, 2/3*C)),
('G', 'L'): ((0, 1/2*TG), (1, 3/2*G)),
('L', 'G'): ((0, TG), (1, 3/2*G)),
('L', 'L'): ((0, TL), (1, 2*L), (3, 1/2*C))}
```

9.4 Jordan Algebras

AUTHORS:

- Travis Scrimshaw (2014-04-02): initial version
- Travis Scrimshaw (2023-05-09): added the 27 dimensional exceptional Jordan algebra

class sage.algebras.jordan_algebra.**ExceptionalJordanAlgebra**(*O*)

Bases: *JordanAlgebra*

The exceptional 27 dimensional Jordan algebra as self-adjoint 3×3 matrix over an octonion algebra.

Let *O* be the *OctonionAlgebra* over a commutative ring *R* of characteristic not equal to 2. The *exceptional*

Jordan algebra $\mathfrak{h}_3(\mathbf{O})$ is a 27 dimensional free R -module spanned by the matrices

$$\begin{bmatrix} \alpha & x & y \\ x^* & \beta & z \\ y^* & z^* & \gamma \end{bmatrix}$$

for $\alpha, \beta, \gamma \in R$ and $x, y, z \in \mathbf{O}$, with multiplication given by the usual symmetrizer operation $X \circ Y = \frac{1}{2}(XY + YX)$.

These are also known as *Albert algebras* due to the work of Abraham Adrian Albert on these algebras over $\mathbf{R}$.

EXAMPLES:

We construct an exceptional Jordan algebra over $\mathbf{Q}$ and perform some basic computations:

```
sage: O = OctonionAlgebra(QQ)
sage: J = JordanAlgebra(O)
sage: gens = J.gens()
sage: gens[1]
[0 0 0]
[0 1 0]
[0 0 0]
sage: gens[3]
[0 1 0]
[1 0 0]
[0 0 0]
sage: gens[1] * gens[3]
[ 0 1/2 0]
[1/2 0 0]
[ 0 0 0]
```

```
>>> from sage.all import *
>>> O = OctonionAlgebra(QQ)
>>> J = JordanAlgebra(O)
>>> gens = J.gens()
>>> gens[Integer(1)]
[0 0 0]
[0 1 0]
[0 0 0]
>>> gens[Integer(3)]
[0 1 0]
[1 0 0]
[0 0 0]
>>> gens[Integer(1)] * gens[Integer(3)]
[ 0 1/2 0]
[1/2 0 0]
[ 0 0 0]
```

The Lie algebra of derivations of the exceptional Jordan algebra is isomorphic to the simple Lie algebra of type F_4 . We verify that the derivation module has the correct dimension:

```
sage: len(J.derivations_basis()) # long time
52
sage: LieAlgebra(QQ, cartan_type='F4').dimension()
52
```



```
>>> from sage.all import *
>>> len(J.derivations_basis()) # long time
52
>>> LieAlgebra(QQ, cartan_type='F4').dimension()
52
```

REFERENCES:

- [Wikipedia article Albert algebra](#)
- [Wikipedia article Jordan algebra#Examples](#)
- [Wikipedia article Hurwitz's theorem \(composition algebras\)#Applications to Jordan algebras](#)
- <https://math.ucr.edu/home/baez/octonions/octonions.pdf>

class `Element` (*parent, data*)

Bases: `AlgebraElement`

An element of an exceptional Jordan algebra.

monomial_coefficients (*copy=True*)

Return a dictionary whose keys are indices of basis elements in the support of `self` and whose values are the corresponding coefficients.

INPUT:

- `copy` – ignored

EXAMPLES:

```
sage: O = OctonionAlgebra(QQ)
sage: J = JordanAlgebra(O)
sage: elt = sum(~QQ(ind) * b for ind, b in enumerate(J.basis()[::6],
↳start=1)); elt
[
      1          1/2*k  1/3*i + 1/4*lk]
[
    -1/2*k          0          1/5*li]
[-1/3*i - 1/4*lk    -1/5*li          0]
sage: elt.monomial_coefficients()
{0: 1, 6: 1/2, 12: 1/3, 18: 1/4, 24: 1/5}
```

```
>>> from sage.all import *
>>> O = OctonionAlgebra(QQ)
>>> J = JordanAlgebra(O)
>>> elt = sum(~QQ(ind) * b for ind, b in enumerate(J.
↳basis()[::Integer(6)], start=Integer(1))); elt
[
      1          1/2*k  1/3*i + 1/4*lk]
[
    -1/2*k          0          1/5*li]
[-1/3*i - 1/4*lk    -1/5*li          0]
>>> elt.monomial_coefficients()
{0: 1, 6: 1/2, 12: 1/3, 18: 1/4, 24: 1/5}
```

algebra_generators ()

Return a basis of `self`.

EXAMPLES:

```

sage: O = OctonionAlgebra(QQ)
sage: J = JordanAlgebra(O)
sage: B = J.basis()
sage: B[::6]
([1 0 0]
 [0 0 0]
 [0 0 0],
 [ 0 k 0]
 [-k 0 0]
 [ 0 0 0],
 [ 0 0 i]
 [ 0 0 0]
 [-i 0 0],
 [ 0 0 1k]
 [ 0 0 0]
 [-1k 0 0],
 [ 0 0 0]
 [ 0 0 1i]
 [ 0 -1i 0])
sage: len(B)
27

```

```

>>> from sage.all import *
>>> O = OctonionAlgebra(QQ)
>>> J = JordanAlgebra(O)
>>> B = J.basis()
>>> B[::Integer(6)]
([1 0 0]
 [0 0 0]
 [0 0 0],
 [ 0 k 0]
 [-k 0 0]
 [ 0 0 0],
 [ 0 0 i]
 [ 0 0 0]
 [-i 0 0],
 [ 0 0 1k]
 [ 0 0 0]
 [-1k 0 0],
 [ 0 0 0]
 [ 0 0 1i]
 [ 0 -1i 0])
>>> len(B)
27

```

basis()

Return a basis of `self`.

EXAMPLES:

```

sage: O = OctonionAlgebra(QQ)
sage: J = JordanAlgebra(O)
sage: B = J.basis()

```

(continues on next page)

(continued from previous page)

```

sage: B[::6]
([1 0 0]
 [0 0 0]
 [0 0 0],
 [ 0 k 0]
 [-k 0 0]
 [ 0 0 0],
 [ 0 0 i]
 [ 0 0 0]
 [-i 0 0],
 [ 0 0 1k]
 [ 0 0 0]
 [-1k 0 0],
 [ 0 0 0]
 [ 0 0 1i]
 [ 0 -1i 0])
sage: len(B)
27

```

```

>>> from sage.all import *
>>> O = OctonionAlgebra(QQ)
>>> J = JordanAlgebra(O)
>>> B = J.basis()
>>> B[::Integer(6)]
([1 0 0]
 [0 0 0]
 [0 0 0],
 [ 0 k 0]
 [-k 0 0]
 [ 0 0 0],
 [ 0 0 i]
 [ 0 0 0]
 [-i 0 0],
 [ 0 0 1k]
 [ 0 0 0]
 [-1k 0 0],
 [ 0 0 0]
 [ 0 0 1i]
 [ 0 -1i 0])
>>> len(B)
27

```

gens()

Return the generators of self.

EXAMPLES:

```

sage: O = OctonionAlgebra(QQ)
sage: J = JordanAlgebra(O)
sage: G = J.gens()
sage: G[0]
[1 0 0]

```

(continues on next page)

(continued from previous page)

```

[0 0 0]
[0 0 0]
sage: G[5]
[ 0  j  0]
[-j  0  0]
[ 0  0  0]
sage: G[22]
[ 0  0  0]
[ 0  0  k]
[ 0 -k  0]

```

```

>>> from sage.all import *
>>> O = OctonionAlgebra(QQ)
>>> J = JordanAlgebra(O)
>>> G = J.gens()
>>> G[Integer(0)]
[1 0 0]
[0 0 0]
[0 0 0]
>>> G[Integer(5)]
[ 0  j  0]
[-j  0  0]
[ 0  0  0]
>>> G[Integer(22)]
[ 0  0  0]
[ 0  0  k]
[ 0 -k  0]

```

one()

Return multiplicative identity.

EXAMPLES:

```

sage: O = OctonionAlgebra(QQ)
sage: J = JordanAlgebra(O)
sage: J.one()
[1 0 0]
[0 1 0]
[0 0 1]
sage: all(J.one() * b == b for b in J.basis())
True

```

```

>>> from sage.all import *
>>> O = OctonionAlgebra(QQ)
>>> J = JordanAlgebra(O)
>>> J.one()
[1 0 0]
[0 1 0]
[0 0 1]
>>> all(J.one() * b == b for b in J.basis())
True

```

some_elements()

(continued from previous page)

```

[ 0  j  0]
[-j  0  0]
[ 0  0  0],
[ 0  0  1j]
[ 0  0  0]
[-1j  0  0],
[ 0  0  0]
[ 0  1 -1j]
[ 0  1j  0],
[      1      0  j - 1i]
[      0      1      0]
[-j + 1i      0      1],
[      1  j + 1k      1]
[-j - 1k      0  i + 1j]
[     -1 -i - 1j      0],
[ 1  0 -k]
[ 0  0  j]
[ k -j  0]]

```

```

>>> from sage.all import *
>>> O = OctonionAlgebra(QQ)
>>> J = JordanAlgebra(O)
>>> J.some_elements()
[[6/5  0  0]
 [ 0 6/5  0]
 [ 0  0 6/5],
 [1 0 0]
 [0 1 0]
 [0 0 1],
 [0 0 0]
 [0 0 0]
 [0 0 0],
 [0 0 0]
 [0 1 0]
 [0 0 0],
 [ 0  j  0]
 [-j  0  0]
 [ 0  0  0],
 [ 0  0  1j]
 [ 0  0  0]
 [-1j  0  0],
 [      0      0      0]
 [      0      1  1/2*1j]
 [      0 -1/2*1j      0],
 [      1      0  j + 2*1i]
 [      0      1      0]
 [-j - 2*1i      0      1],
 [      1  j + 1k      1]
 [-j - 1k      0  i + 1j]
 [     -1 -i - 1j      0],
 [      1  3/2*1  2*k]
 [-3/2*1      0  5/2*j]

```

(continues on next page)

(continued from previous page)

```

[ -2*k -5/2*j      0]]

>>> O = OctonionAlgebra(GF(Integer(3)))
>>> J = JordanAlgebra(O)
>>> J.some_elements()
[[-1  0  0]
 [ 0 -1  0]
 [ 0  0 -1],
 [1 0 0]
 [0 1 0]
 [0 0 1],
 [0 0 0]
 [0 0 0]
 [0 0 0],
 [0 0 0]
 [0 1 0]
 [0 0 0],
 [ 0  j  0]
 [-j  0  0]
 [ 0  0  0],
 [ 0  0  1j]
 [ 0  0  0]
 [-1j  0  0],
 [ 0  0  0]
 [ 0  1 -1j]
 [ 0  1j  0],
 [      1      0  j - 1i]
 [      0      1      0]
 [-j + 1i      0      1],
 [      1  j + 1k      1]
 [-j - 1k      0  i + 1j]
 [      -1 -i - 1j      0],
 [ 1  0 -k]
 [ 0  0  j]
 [ k -j  0]]

```

zero()

Return the additive identity.

EXAMPLES:

```

sage: O = OctonionAlgebra(QQ)
sage: J = JordanAlgebra(O)
sage: J.zero()
[0 0 0]
[0 0 0]
[0 0 0]

```

```

>>> from sage.all import *
>>> O = OctonionAlgebra(QQ)
>>> J = JordanAlgebra(O)
>>> J.zero()

```

(continues on next page)

(continued from previous page)

```
[0 0 0]
[0 0 0]
[0 0 0]
```

class sage.algebras.jordan_algebra.**JordanAlgebra**

Bases: UniqueRepresentation, Parent

A Jordan algebra.

A *Jordan algebra* is a magmatic algebra (over a commutative ring R) whose multiplication satisfies the following axioms:

- $xy = yx$, and
- $(xy)(xx) = x(y(xx))$ (the Jordan identity).

See [Ja1971], [Ch2012], and [McC1978], for example.

These axioms imply that a Jordan algebra is power-associative and the following generalization of Jordan's identity holds [Al1947]: $(x^m y)x^n = x^m(yx^n)$ for all $m, n \in \mathbf{Z}_{>0}$.

Let A be an associative algebra over a ring R in which 2 is invertible. We construct a Jordan algebra A^+ with ground set A by defining the multiplication as

$$x \circ y = \frac{xy + yx}{2}.$$

Often the multiplication is written as $x \circ y$ to avoid confusion with the product in the associative algebra A . We note that if A is commutative then this reduces to the usual multiplication in A .

Jordan algebras constructed in this fashion, or their subalgebras, are called *special*. All other Jordan algebras are called *exceptional*.

Jordan algebras can also be constructed from a module M over R with a symmetric bilinear form $(\cdot, \cdot) : M \times M \rightarrow R$. We begin with the module $M^* = R \oplus M$ and define multiplication in M^* by

$$(\alpha + x) \circ (\beta + y) = \underbrace{\alpha\beta + (x, y)}_{\in R} + \underbrace{\beta x + \alpha y}_{\in M},$$

where $\alpha, \beta \in R$ and $x, y \in M$.

INPUT:

Can be either an associative algebra A or a symmetric bilinear form given as a matrix (possibly followed by, or preceded by, a base ring argument).

EXAMPLES:

We let the base algebra A be the free algebra on 3 generators:

```
sage: F.<x,y,z> = FreeAlgebra(QQ)
sage: J = JordanAlgebra(F); J
Jordan algebra of Free Algebra on 3 generators (x, y, z) over Rational Field
sage: a,b,c = map(J, F.gens())
sage: a*b
1/2*x*y + 1/2*y*x
sage: b*a
1/2*x*y + 1/2*y*x
```



```

>>> from sage.all import *
>>> F = FreeAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z) = F._first_ngens(3)
>>> J = JordanAlgebra(F); J
Jordan algebra of Free Algebra on 3 generators (x, y, z) over Rational Field
>>> a,b,c = map(J, F.gens())
>>> a*b
1/2*x*y + 1/2*y*x
>>> b*a
1/2*x*y + 1/2*y*x

```

Jordan algebras are typically non-associative:

```

sage: (a*b)*c
1/4*x*y*z + 1/4*y*x*z + 1/4*z*x*y + 1/4*z*y*x
sage: a*(b*c)
1/4*x*y*z + 1/4*x*z*y + 1/4*y*z*x + 1/4*z*y*x

```

```

>>> from sage.all import *
>>> (a*b)*c
1/4*x*y*z + 1/4*y*x*z + 1/4*z*x*y + 1/4*z*y*x
>>> a*(b*c)
1/4*x*y*z + 1/4*x*z*y + 1/4*y*z*x + 1/4*z*y*x

```

We check the Jordan identity:

```

sage: (a*b)*(a*a) == a*(b*(a*a))
True
sage: x = a + c
sage: y = b - 2*a
sage: (x*y)*(x*x) == x*(y*(x*x))
True

```

```

>>> from sage.all import *
>>> (a*b)*(a*a) == a*(b*(a*a))
True
>>> x = a + c
>>> y = b - Integer(2)*a
>>> (x*y)*(x*x) == x*(y*(x*x))
True

```

Next we construct a Jordan algebra from a symmetric bilinear form:

```

sage: m = matrix([[-2,3],[3,4]])
sage: J.<a,b,c> = JordanAlgebra(m); J
Jordan algebra over Integer Ring given by the symmetric bilinear form:
[-2  3]
[ 3  4]
sage: a
1 + (0, 0)
sage: b
0 + (1, 0)
sage: x = 3*a - 2*b + c; x
3 + (-2, 1)

```

```

>>> from sage.all import *
>>> m = matrix([[Integer(2), Integer(3)], [Integer(3), Integer(4)]])
>>> J = JordanAlgebra(m, names=('a', 'b', 'c')); (a, b, c) = J._first_ngens(3);
↪ J
Jordan algebra over Integer Ring given by the symmetric bilinear form:
[-2  3]
[ 3  4]
>>> a
1 + (0, 0)
>>> b
0 + (1, 0)
>>> x = Integer(3)*a - Integer(2)*b + c; x
3 + (-2, 1)

```

We again show that Jordan algebras are usually non-associative:

```

sage: (x*b)*b
-6 + (7, 0)
sage: x*(b*b)
-6 + (4, -2)

```

```

>>> from sage.all import *
>>> (x*b)*b
-6 + (7, 0)
>>> x*(b*b)
-6 + (4, -2)

```

We verify the Jordan identity:

```

sage: y = -a + 4*b - c
sage: (x*y)*(x*x) == x*(y*(x*x))
True

```

```

>>> from sage.all import *
>>> y = -a + Integer(4)*b - c
>>> (x*y)*(x*x) == x*(y*(x*x))
True

```

The base ring, while normally inferred from the matrix, can also be explicitly specified:

```

sage: J.<a,b,c> = JordanAlgebra(m, QQ); J
Jordan algebra over Rational Field given by the symmetric bilinear form:
[-2  3]
[ 3  4]
sage: J.<a,b,c> = JordanAlgebra(QQ, m); J # either order work
Jordan algebra over Rational Field given by the symmetric bilinear form:
[-2  3]
[ 3  4]

```

```

>>> from sage.all import *
>>> J = JordanAlgebra(m, QQ, names=('a', 'b', 'c')); (a, b, c) = J._first_
↪ ngens(3); J

```

(continues on next page)

(continued from previous page)

```

Jordan algebra over Rational Field given by the symmetric bilinear form:
[-2  3]
[ 3  4]
>>> J = JordanAlgebra(QQ, m, names=('a', 'b', 'c',)); (a, b, c,) = J._first_
↳ngens(3); J # either order work
Jordan algebra over Rational Field given by the symmetric bilinear form:
[-2  3]
[ 3  4]

```

REFERENCES:

- [Wikipedia article Jordan_algebra](#)
- [Ja1971]
- [Ch2012]
- [McC1978]
- [Al1947]

class sage.algebras.jordan_algebra.**JordanAlgebraSymmetricBilinear**(*R*, *form*, *names=None*)

Bases: *JordanAlgebra*

A Jordan algebra given by a symmetric bilinear form *m*.

class **Element**(*parent*, *s*, *v*)

Bases: *AlgebraElement*

An element of a Jordan algebra defined by a symmetric bilinear form.

bar()

Return the result of the bar involution of *self*.

The bar involution $\bar{}$ is the *R*-linear endomorphism of M^* defined by $\bar{1} = 1$ and $\bar{x} = -x$ for $x \in M$.

EXAMPLES:

```

sage: m = matrix([[0,1],[1,1]])
sage: J.<a,b,c> = JordanAlgebra(m)
sage: x = 4*a - b + 3*c
sage: x.bar()
4 + (1, -3)

```

```

>>> from sage.all import *
>>> m = matrix([[Integer(0), Integer(1)], [Integer(1), Integer(1)]])
>>> J = JordanAlgebra(m, names=('a', 'b', 'c',)); (a, b, c,) = J._first_
↳ngens(3)
>>> x = Integer(4)*a - b + Integer(3)*c
>>> x.bar()
4 + (1, -3)

```

We check that it is an algebra morphism:

```

sage: y = 2*a + 2*b - c
sage: x.bar() * y.bar() == (x*y).bar()
True

```

```
>>> from sage.all import *
>>> y = Integer(2)*a + Integer(2)*b - c
>>> x.bar() * y.bar() == (x*y).bar()
True
```

monomial_coefficients (*copy=True*)

Return a dictionary whose keys are indices of basis elements in the support of `self` and whose values are the corresponding coefficients.

INPUT:

- `copy` – ignored

EXAMPLES:

```
sage: m = matrix([[0,1],[1,1]])
sage: J.<a,b,c> = JordanAlgebra(m)
sage: elt = a + 2*b - c
sage: elt.monomial_coefficients()
{0: 1, 1: 2, 2: -1}
```

```
>>> from sage.all import *
>>> m = matrix([[Integer(0),Integer(1)],[Integer(1),Integer(1)]])
>>> J = JordanAlgebra(m, names=('a', 'b', 'c',)); (a, b, c,) = J._first_
↳ngens(3)
>>> elt = a + Integer(2)*b - c
>>> elt.monomial_coefficients()
{0: 1, 1: 2, 2: -1}
```

norm()

Return the norm of `self`.

The norm of an element $\alpha + x \in M^*$ is given by $n(\alpha + x) = \alpha^2 - (x, x)$.

EXAMPLES:

```
sage: m = matrix([[0,1],[1,1]])
sage: J.<a,b,c> = JordanAlgebra(m)
sage: x = 4*a - b + 3*c; x
4 + (-1, 3)
sage: x.norm()
13
```

```
>>> from sage.all import *
>>> m = matrix([[Integer(0),Integer(1)],[Integer(1),Integer(1)]])
>>> J = JordanAlgebra(m, names=('a', 'b', 'c',)); (a, b, c,) = J._first_
↳ngens(3)
>>> x = Integer(4)*a - b + Integer(3)*c; x
4 + (-1, 3)
>>> x.norm()
13
```

trace()

Return the trace of `self`.

The trace of an element $\alpha + x \in M^*$ is given by $t(\alpha + x) = 2\alpha$.

EXAMPLES:

```
sage: m = matrix([[0,1],[1,1]])
sage: J.<a,b,c> = JordanAlgebra(m)
sage: x = 4*a - b + 3*c
sage: x.trace()
8
```

```
>>> from sage.all import *
>>> m = matrix([[Integer(0),Integer(1)],[Integer(1),Integer(1)]])
>>> J = JordanAlgebra(m, names=('a', 'b', 'c')); (a, b, c) = J._first_
↳ngens(3)
>>> x = Integer(4)*a - b + Integer(3)*c
>>> x.trace()
8
```

algebra_generators()

Return a basis of `self`.

The basis returned begins with the unity of R and continues with the standard basis of M .

EXAMPLES:

```
sage: m = matrix([[0,1],[1,1]])
sage: J = JordanAlgebra(m)
sage: J.basis()
Family (1 + (0, 0), 0 + (1, 0), 0 + (0, 1))
```

```
>>> from sage.all import *
>>> m = matrix([[Integer(0),Integer(1)],[Integer(1),Integer(1)]])
>>> J = JordanAlgebra(m)
>>> J.basis()
Family (1 + (0, 0), 0 + (1, 0), 0 + (0, 1))
```

basis()

Return a basis of `self`.

The basis returned begins with the unity of R and continues with the standard basis of M .

EXAMPLES:

```
sage: m = matrix([[0,1],[1,1]])
sage: J = JordanAlgebra(m)
sage: J.basis()
Family (1 + (0, 0), 0 + (1, 0), 0 + (0, 1))
```

```
>>> from sage.all import *
>>> m = matrix([[Integer(0),Integer(1)],[Integer(1),Integer(1)]])
>>> J = JordanAlgebra(m)
>>> J.basis()
Family (1 + (0, 0), 0 + (1, 0), 0 + (0, 1))
```

gens()

Return the generators of `self`.

EXAMPLES:

```
sage: m = matrix([[0,1],[1,1]])
sage: J = JordanAlgebra(m)
sage: J.gens()
(1 + (0, 0), 0 + (1, 0), 0 + (0, 1))
```

```
>>> from sage.all import *
>>> m = matrix([[Integer(0),Integer(1)],[Integer(1),Integer(1)]])
>>> J = JordanAlgebra(m)
>>> J.gens()
(1 + (0, 0), 0 + (1, 0), 0 + (0, 1))
```

one()

Return the element 1 if it exists.

EXAMPLES:

```
sage: m = matrix([[0,1],[1,1]])
sage: J = JordanAlgebra(m)
sage: J.one()
1 + (0, 0)
```

```
>>> from sage.all import *
>>> m = matrix([[Integer(0),Integer(1)],[Integer(1),Integer(1)]])
>>> J = JordanAlgebra(m)
>>> J.one()
1 + (0, 0)
```

zero()

Return the element 0.

EXAMPLES:

```
sage: m = matrix([[0,1],[1,1]])
sage: J = JordanAlgebra(m)
sage: J.zero()
0 + (0, 0)
```

```
>>> from sage.all import *
>>> m = matrix([[Integer(0),Integer(1)],[Integer(1),Integer(1)]])
>>> J = JordanAlgebra(m)
>>> J.zero()
0 + (0, 0)
```

class sage.algebras.jordan_algebra.**SpecialJordanAlgebra** (*A*, *names=None*)

Bases: *JordanAlgebra*

A (special) Jordan algebra A^+ from an associative algebra A .

class **Element** (*parent*, *x*)

Bases: *AlgebraElement*

An element of a special Jordan algebra.

monomial_coefficients (*copy=True*)

Return a dictionary whose keys are indices of basis elements in the support of *self* and whose values are the corresponding coefficients.

INPUT:

- *copy* – boolean (default: *True*); if *self* is internally represented by a dictionary *d*, then make a copy of *d*; if *False*, then this can cause undesired behavior by mutating *d*

EXAMPLES:

```
sage: F.<x,y,z> = FreeAlgebra(QQ)
sage: J = JordanAlgebra(F)
sage: a,b,c = map(J, F.gens())
sage: elt = a + 2*b - c
sage: elt.monomial_coefficients()
{x: 1, y: 2, z: -1}
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z) = F._first_
↳ngens(3)
>>> J = JordanAlgebra(F)
>>> a,b,c = map(J, F.gens())
>>> elt = a + Integer(2)*b - c
>>> elt.monomial_coefficients()
{x: 1, y: 2, z: -1}
```

algebra_generators ()

Return the basis of *self*.

EXAMPLES:

```
sage: F.<x,y,z> = FreeAlgebra(QQ)
sage: J = JordanAlgebra(F)
sage: J.basis()
Lazy family (Term map(i))_{i in Free monoid on 3 generators (x, y, z)}
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z) = F._first_
↳ngens(3)
>>> J = JordanAlgebra(F)
>>> J.basis()
Lazy family (Term map(i))_{i in Free monoid on 3 generators (x, y, z)}
```

basis ()

Return the basis of *self*.

EXAMPLES:

```
sage: F.<x,y,z> = FreeAlgebra(QQ)
sage: J = JordanAlgebra(F)
sage: J.basis()
Lazy family (Term map(i))_{i in Free monoid on 3 generators (x, y, z)}
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z) = F._first_
```

(continues on next page)

(continued from previous page)

```

↪ngens(3)
>>> J = JordanAlgebra(F)
>>> J.basis()
Lazy family (Term map(i))_{i in Free monoid on 3 generators (x, y, z)}

```

gens()

Return the generators of self.

EXAMPLES:

```

sage: cat = Algebras(QQ).WithBasis().FiniteDimensional()
sage: C = CombinatorialFreeModule(QQ, ['x', 'y', 'z'], category=cat)
sage: J = JordanAlgebra(C)
sage: J.gens()
(B['x'], B['y'], B['z'])

sage: F.<x,y,z> = FreeAlgebra(QQ)
sage: J = JordanAlgebra(F)
sage: J.gens()
Traceback (most recent call last):
...
NotImplementedError: infinite set

```

```

>>> from sage.all import *
>>> cat = Algebras(QQ).WithBasis().FiniteDimensional()
>>> C = CombinatorialFreeModule(QQ, ['x', 'y', 'z'], category=cat)
>>> J = JordanAlgebra(C)
>>> J.gens()
(B['x'], B['y'], B['z'])

>>> F = FreeAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z) = F._first_
↪ngens(3)
>>> J = JordanAlgebra(F)
>>> J.gens()
Traceback (most recent call last):
...
NotImplementedError: infinite set

```

one()

Return the element 1 if it exists.

EXAMPLES:

```

sage: F.<x,y,z> = FreeAlgebra(QQ)
sage: J = JordanAlgebra(F)
sage: J.one()
1

```

```

>>> from sage.all import *
>>> F = FreeAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z) = F._first_
↪ngens(3)
>>> J = JordanAlgebra(F)

```

(continues on next page)

(continued from previous page)

```
>>> J.one()
1
```

zero()

Return the element 0.

EXAMPLES:

```
sage: F.<x,y,z> = FreeAlgebra(QQ)
sage: J = JordanAlgebra(F)
sage: J.zero()
0
```

```
>>> from sage.all import *
>>> F = FreeAlgebra(QQ, names=('x', 'y', 'z')); (x, y, z) = F._first_
↳ngens(3)
>>> J = JordanAlgebra(F)
>>> J.zero()
0
```

9.5 Free Dendriform Algebras

AUTHORS:

Frédéric Chapoton (2017)

class sage.combinat.free_dendriform_algebra.DendriformFunctor(*vars*)

Bases: ConstructionFunctor

A constructor for dendriform algebras.

EXAMPLES:

```
sage: P = algebras.FreeDendriform(ZZ, 'x,y')
sage: x,y = P.gens()
sage: F = P.construction()[0]; F
Dendriform[x,y]

sage: A = GF(5)['a,b']
sage: a, b = A.gens()
sage: F(A)
Free Dendriform algebra on 2 generators ['x', 'y']
over Multivariate Polynomial Ring in a, b over Finite Field of size 5

sage: f = A.hom([a+b,a-b],A)
sage: F(f)
Generic endomorphism of Free Dendriform algebra on 2 generators ['x', 'y']
over Multivariate Polynomial Ring in a, b over Finite Field of size 5

sage: F(f)(a * F(A)(x))
(a+b)*B[x[., .]]
```

```

>>> from sage.all import *
>>> P = algebras.FreeDendriform(ZZ, 'x,y')
>>> x,y = P.gens()
>>> F = P.construction()[Integer(0)]; F
Dendriform[x,y]

>>> A = GF(Integer(5))['a,b']
>>> a, b = A.gens()
>>> F(A)
Free Dendriform algebra on 2 generators ['x', 'y']
over Multivariate Polynomial Ring in a, b over Finite Field of size 5

>>> f = A.hom([a+b,a-b],A)
>>> F(f)
Generic endomorphism of Free Dendriform algebra on 2 generators ['x', 'y']
over Multivariate Polynomial Ring in a, b over Finite Field of size 5

>>> F(f)(a * F(A)(x))
(a+b)*B[x[., .]]

```

merge(*other*)

Merge self with another construction functor, or return None.

EXAMPLES:

```

sage: F = sage.combinat.free_dendriform_algebra.DendriformFunctor(['x','y'])
sage: G = sage.combinat.free_dendriform_algebra.DendriformFunctor(['t'])
sage: F.merge(G)
Dendriform[x,y,t]
sage: F.merge(F)
Dendriform[x,y]

```

```

>>> from sage.all import *
>>> F = sage.combinat.free_dendriform_algebra.DendriformFunctor(['x','y'])
>>> G = sage.combinat.free_dendriform_algebra.DendriformFunctor(['t'])
>>> F.merge(G)
Dendriform[x,y,t]
>>> F.merge(F)
Dendriform[x,y]

```

Now some actual use cases:

```

sage: R = algebras.FreeDendriform(ZZ, 'x,y,z')
sage: x,y,z = R.gens()
sage: 1/2 * x
1/2*B[x[., .]]
sage: parent(1/2 * x)
Free Dendriform algebra on 3 generators ['x', 'y', 'z'] over Rational Field

sage: S = algebras.FreeDendriform(QQ, 'zt')
sage: z,t = S.gens()
sage: x + t
B[t[., .]] + B[x[., .]]

```

(continues on next page)

(continued from previous page)

```
sage: parent(x + t)
Free Dendriform algebra on 4 generators ['z', 't', 'x', 'y'] over Rational_
↳Field
```

```
>>> from sage.all import *
>>> R = algebras.FreeDendriform(ZZ, 'x,y,z')
>>> x,y,z = R.gens()
>>> Integer(1)/Integer(2) * x
1/2*B[x[., .]]
>>> parent(Integer(1)/Integer(2) * x)
Free Dendriform algebra on 3 generators ['x', 'y', 'z'] over Rational Field

>>> S = algebras.FreeDendriform(QQ, 'zt')
>>> z,t = S.gens()
>>> x + t
B[t[., .]] + B[x[., .]]
>>> parent(x + t)
Free Dendriform algebra on 4 generators ['z', 't', 'x', 'y'] over Rational_
↳Field
```

rank = 9

```
class sage.combinat.free_dendriform_algebra.FreeDendriformAlgebra(R, names=None)
```

Bases: `CombinatorialFreeModule`

The free dendriform algebra.

Dendriform algebras are associative algebras, where the associative product $*$ is decomposed as a sum of two binary operations

$$x * y = x \succ y + x \prec y$$

that satisfy the axioms:

$$(x \succ y) \prec z = x \succ (y \prec z),$$

$$(x \prec y) \prec z = x \prec (y * z).$$

$$(x * y) \succ z = x \succ (y \succ z).$$

The free Dendriform algebra on a given set E has an explicit description using (planar) binary trees, just as the free associative algebra can be described using words. The underlying vector space has a basis indexed by finite binary trees endowed with a map from their vertices to E . In this basis, the associative product of two (decorated) binary trees $S * T$ is the sum over all possible ways of identifying (glueing) the rightmost path in S and the leftmost path in T .

The decomposition of the associative product as the sum of two binary operations $\succ$ and $\prec$ is made by separating the terms according to the origin of the root vertex. For $x \succ y$, one keeps the terms where the root vertex comes from y , whereas for $x \prec y$ one keeps the terms where the root vertex comes from x .

The free dendriform algebra can also be considered as the free algebra over the Dendriform operad.

Note

The usual binary operator $*$ is used for the associative product.

EXAMPLES:

```
sage: F = algebras.FreeDendriform(ZZ, 'xyz')
sage: x,y,z = F.gens()
sage: (x * y) * z
B[x[., y[., z[., .]]]] + B[x[., z[y[., .], .]]] + B[y[x[., .], z[., .]]]
+ B[z[x[., y[., .]], .]] + B[z[y[x[., .], .], .]]
```

```
>>> from sage.all import *
>>> F = algebras.FreeDendriform(ZZ, 'xyz')
>>> x,y,z = F.gens()
>>> (x * y) * z
B[x[., y[., z[., .]]]] + B[x[., z[y[., .], .]]] + B[y[x[., .], z[., .]]]
+ B[z[x[., y[., .]], .]] + B[z[y[x[., .], .], .]]
```

The free dendriform algebra is associative:

```
sage: x * (y * z) == (x * y) * z
True
```

```
>>> from sage.all import *
>>> x * (y * z) == (x * y) * z
True
```

The associative product decomposes in two parts:

```
sage: x * y == F.prec(x, y) + F.succ(x, y)
True
```

```
>>> from sage.all import *
>>> x * y == F.prec(x, y) + F.succ(x, y)
True
```

The axioms hold:

```
sage: F.prec(F.succ(x, y), z) == F.succ(x, F.prec(y, z))
True
sage: F.prec(F.prec(x, y), z) == F.prec(x, y * z)
True
sage: F.succ(x * y, z) == F.succ(x, F.succ(y, z))
True
```

```
>>> from sage.all import *
>>> F.prec(F.succ(x, y), z) == F.succ(x, F.prec(y, z))
True
>>> F.prec(F.prec(x, y), z) == F.prec(x, y * z)
True
>>> F.succ(x * y, z) == F.succ(x, F.succ(y, z))
True
```

When there is only one generator, unlabelled trees are used instead:

```

sage: F1 = algebras.FreeDendriform(QQ)
sage: w = F1.gen(0); w
B[[., .]]
sage: w * w * w
B[[., [., [., .]]]] + B[[., [[., .], .]]] + B[[[., .], [., .]]]
+ B[[[., [., .]], .]] + B[[[[., .], .], .]]

```

```

>>> from sage.all import *
>>> F1 = algebras.FreeDendriform(QQ)
>>> w = F1.gen(Integer(0)); w
B[[., .]]
>>> w * w * w
B[[., [., [., .]]]] + B[[., [[., .], .]]] + B[[[., .], [., .]]]
+ B[[[., [., .]], .]] + B[[[[., .], .], .]]

```

The set E can be infinite:

```

sage: F = algebras.FreeDendriform(QQ, ZZ)
sage: w = F.gen(1); w
B[1[., .]]
sage: x = F.gen(2); x
B[-1[., .]]
sage: w*x
B[-1[1[., .], .]] + B[1[., -1[., .]]]

```

```

>>> from sage.all import *
>>> F = algebras.FreeDendriform(QQ, ZZ)
>>> w = F.gen(Integer(1)); w
B[1[., .]]
>>> x = F.gen(Integer(2)); x
B[-1[., .]]
>>> w*x
B[-1[1[., .], .]] + B[1[., -1[., .]]]

```

REFERENCES:

- [LR1998]

`algebra_generators()`

Return the generators of this algebra.

These are the binary trees with just one vertex.

EXAMPLES:

```

sage: A = algebras.FreeDendriform(ZZ, 'fgh'); A
Free Dendriform algebra on 3 generators ['f', 'g', 'h']
over Integer Ring
sage: list(A.algebra_generators())
[B[f[., .]], B[g[., .]], B[h[., .]]]

sage: A = algebras.FreeDendriform(QQ, ['x1', 'x2'])
sage: list(A.algebra_generators())
[B[x1[., .]], B[x2[., .]]]

```

```

>>> from sage.all import *
>>> A = algebras.FreeDendriform(ZZ, 'fgh'); A
Free Dendriform algebra on 3 generators ['f', 'g', 'h']
over Integer Ring
>>> list(A.algebra_generators())
[B[f[., .]], B[g[., .]], B[h[., .]]]

>>> A = algebras.FreeDendriform(QQ, ['x1', 'x2'])
>>> list(A.algebra_generators())
[B[x1[., .]], B[x2[., .]]]

```

an_element()

Return an element of self.

EXAMPLES:

```

sage: A = algebras.FreeDendriform(QQ, 'xy')
sage: A.an_element()
B[x[., .]] + 2*B[x[., x[., .]]] + 2*B[x[x[., .], .]]

```

```

>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ, 'xy')
>>> A.an_element()
B[x[., .]] + 2*B[x[., x[., .]]] + 2*B[x[x[., .], .]]

```

change_ring(R)

Return the free dendriform algebra in the same variables over R .

INPUT:

- R – a ring

EXAMPLES:

```

sage: A = algebras.FreeDendriform(ZZ, 'fgh')
sage: A.change_ring(QQ)
Free Dendriform algebra on 3 generators ['f', 'g', 'h'] over
Rational Field

```

```

>>> from sage.all import *
>>> A = algebras.FreeDendriform(ZZ, 'fgh')
>>> A.change_ring(QQ)
Free Dendriform algebra on 3 generators ['f', 'g', 'h'] over
Rational Field

```

construction()

Return a pair (F, R) , where F is a *DendriformFunctor* and R is a ring, such that $F(R)$ returns self.

EXAMPLES:

```

sage: P = algebras.FreeDendriform(ZZ, 'x,y')
sage: x,y = P.gens()
sage: F, R = P.construction()
sage: F

```

(continues on next page)

(continued from previous page)

```

Dendriform[x,y]
sage: R
Integer Ring
sage: F(ZZ) is P
True
sage: F(QQ)
Free Dendriform algebra on 2 generators ['x', 'y'] over Rational Field

```

```

>>> from sage.all import *
>>> P = algebras.FreeDendriform(ZZ, 'x,y')
>>> x,y = P.gens()
>>> F, R = P.construction()
>>> F
Dendriform[x,y]
>>> R
Integer Ring
>>> F(ZZ) is P
True
>>> F(QQ)
Free Dendriform algebra on 2 generators ['x', 'y'] over Rational Field

```

coproduct_on_basis(x)

Return the coproduct of a binary tree.

EXAMPLES:

```

sage: A = algebras.FreeDendriform(QQ)
sage: x = A.gen(0)
sage: ascii_art(A.coproduct(A.one())) # indirect doctest
1 # 1

sage: ascii_art(A.coproduct(x)) # indirect doctest
1 # B + B # 1
   o   o

sage: A = algebras.FreeDendriform(QQ, 'xyz')
sage: x, y, z = A.gens()
sage: w = A.under(z,A.over(x,y))
sage: A.coproduct(z)
B[.] # B[z[., .]] + B[z[., .]] # B[.]
sage: A.coproduct(w)
B[.] # B[x[z[., .], y[., .]]] + B[x[., .]] # B[z[., y[., .]]] +
B[x[., .]] # B[y[z[., .], .]] + B[x[., y[., .]]] # B[z[., .]] +
B[x[z[., .], .]] # B[y[., .]] + B[x[z[., .], y[., .]]] # B[.]

```

```

>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ)
>>> x = A.gen(Integer(0))
>>> ascii_art(A.coproduct(A.one())) # indirect doctest
1 # 1

>>> ascii_art(A.coproduct(x)) # indirect doctest

```

(continues on next page)

(continued from previous page)

```

1 # B + B # 1
   o   o

>>> A = algebras.FreeDendriform(QQ, 'xyz')
>>> x, y, z = A.gens()
>>> w = A.under(z, A.over(x, y))
>>> A.coproduct(z)
B[.] # B[z[., .]] + B[z[., .]] # B[.]
>>> A.coproduct(w)
B[.] # B[x[z[., .], y[., .]]] + B[x[., .]] # B[z[., y[., .]]] +
B[x[., .]] # B[y[z[., .], .]] + B[x[., y[., .]]] # B[z[., .]] +
B[x[z[., .], .]] # B[y[., .]] + B[x[z[., .], y[., .]]] # B[.]

```

degree_on_basis(*t*)

Return the degree of a binary tree in the free Dendriform algebra.

This is the number of vertices.

EXAMPLES:

```

sage: A = algebras.FreeDendriform(QQ, '@')
sage: RT = A.basis().keys()
sage: u = RT([], '@')
sage: A.degree_on_basis(u.over(u))
2

```

```

>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ, '@')
>>> RT = A.basis().keys()
>>> u = RT([], '@')
>>> A.degree_on_basis(u.over(u))
2

```

gen(*i*)

Return the *i*-th generator of the algebra.

INPUT:

- *i* – integer

EXAMPLES:

```

sage: F = algebras.FreeDendriform(ZZ, 'xyz')
sage: F.gen(0)
B[x[., .]]

sage: F.gen(4)
Traceback (most recent call last):
...
IndexError: argument i (= 4) must be between 0 and 2

```

```

>>> from sage.all import *
>>> F = algebras.FreeDendriform(ZZ, 'xyz')
>>> F.gen(Integer(0))

```

(continues on next page)

(continued from previous page)

```

B[x[., .]]

>>> F.gen(Integer(4))
Traceback (most recent call last):
...
IndexError: argument i (= 4) must be between 0 and 2

```

gens()

Return the generators of `self` (as an algebra).

EXAMPLES:

```

sage: A = algebras.FreeDendriform(ZZ, 'fgh')
sage: A.gens()
(B[f[., .]], B[g[., .]], B[h[., .]])

```

```

>>> from sage.all import *
>>> A = algebras.FreeDendriform(ZZ, 'fgh')
>>> A.gens()
(B[f[., .]], B[g[., .]], B[h[., .]])

```

one_basis()

Return the index of the unit.

EXAMPLES:

```

sage: A = algebras.FreeDendriform(QQ, '@')
sage: A.one_basis()
.
sage: A = algebras.FreeDendriform(QQ, 'xy')
sage: A.one_basis()
.

```

```

>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ, '@')
>>> A.one_basis()
.
>>> A = algebras.FreeDendriform(QQ, 'xy')
>>> A.one_basis()
.

```

over()

Return the over product.

The over product x/y is the binary tree obtained by grafting the root of y at the rightmost leaf of x .

The usual symbol for this operation is $/$.

See also

`product()`, `succ()`, `prec()`, `under()`

EXAMPLES:

```
sage: A = algebras.FreeDendriform(QQ)
sage: RT = A.basis().keys()
sage: x = A.gen(0)
sage: A.over(x, x)
B[[., [., .]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ)
>>> RT = A.basis().keys()
>>> x = A.gen(Integer(0))
>>> A.over(x, x)
B[[., [., .]]]
```

prec()

Return the $\prec$ dendriform product.

This is the sum over all possible ways to identify the rightmost path in x and the leftmost path in y , with the additional condition that the root vertex of the result comes from x .

The usual symbol for this operation is $\prec$.

See also

`product()`, `succ()`, `over()`, `under()`

EXAMPLES:

```
sage: A = algebras.FreeDendriform(QQ)
sage: RT = A.basis().keys()
sage: x = A.gen(0)
sage: A.prec(x, x)
B[[., [., .]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ)
>>> RT = A.basis().keys()
>>> x = A.gen(Integer(0))
>>> A.prec(x, x)
B[[., [., .]]]
```

prec_product_on_basis(x, y)

Return the $\prec$ dendriform product of two trees.

This is the sum over all possible ways of identifying the rightmost path in x and the leftmost path in y , with the additional condition that the root vertex of the result comes from x .

The usual symbol for this operation is $\prec$.

See also

- `product_on_basis()`, `succ_product_on_basis()`

EXAMPLES:

```
sage: A = algebras.FreeDendriform(QQ)
sage: RT = A.basis().keys()
sage: x = RT([])
sage: A.prec_product_on_basis(x, x)
B[[., [., .]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ)
>>> RT = A.basis().keys()
>>> x = RT([])
>>> A.prec_product_on_basis(x, x)
B[[., [., .]]]
```

product_on_basis(x, y)

Return the $*$ associative dendriform product of two trees.

This is the sum over all possible ways of identifying the rightmost path in x and the leftmost path in y . Every term corresponds to a shuffle of the vertices on the rightmost path in x and the vertices on the leftmost path in y .

See also

- `succ_product_on_basis()`, `prec_product_on_basis()`

EXAMPLES:

```
sage: A = algebras.FreeDendriform(QQ)
sage: RT = A.basis().keys()
sage: x = RT([])
sage: A.product_on_basis(x, x)
B[[., [., .]]] + B[[[., .], .]]
```

```
>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ)
>>> RT = A.basis().keys()
>>> x = RT([])
>>> A.product_on_basis(x, x)
B[[., [., .]]] + B[[[., .], .]]
```

some_elements()

Return some elements of the free dendriform algebra.

EXAMPLES:

```
sage: A = algebras.FreeDendriform(QQ)
sage: A.some_elements()
[B[.],
 B[[., .]],
 B[[., [., .]]] + B[[[., .], .]],
 B[.] + B[[., [., .]]] + B[[[., .], .]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ)
>>> A.some_elements()
[B[.],
 B[[.], .]],
 B[[.], [.], .]] + B[[[.], .], .]],
 B[.] + B[[.], [.], .]] + B[[[.], .], .]]]
```

With several generators:

```
sage: A = algebras.FreeDendriform(QQ, 'xy')
sage: A.some_elements()
[B[.],
 B[x[.], .]],
 B[x[.], x[.], .]] + B[x[x[.], .], .]],
 B[.] + B[x[.], x[.], .]] + B[x[x[.], .], .]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ, 'xy')
>>> A.some_elements()
[B[.],
 B[x[.], .]],
 B[x[.], x[.], .]] + B[x[x[.], .], .]],
 B[.] + B[x[.], x[.], .]] + B[x[x[.], .], .]]]
```

succ()

Return the $\succ$ dendriform product.

This is the sum over all possible ways of identifying the rightmost path in x and the leftmost path in y , with the additional condition that the root vertex of the result comes from y .

The usual symbol for this operation is $\succ$.

See also

`product()`, `prec()`, `over()`, `under()`

EXAMPLES:

```
sage: A = algebras.FreeDendriform(QQ)
sage: RT = A.basis().keys()
sage: x = A.gen(0)
sage: A.succ(x, x)
B[[[.], .], .]]
```

```
>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ)
>>> RT = A.basis().keys()
>>> x = A.gen(Integer(0))
>>> A.succ(x, x)
B[[[.], .], .]]
```

succ_product_on_basis (x, y)

Return the $\succ$ dendriform product of two trees.

This is the sum over all possible ways to identify the rightmost path in x and the leftmost path in y , with the additional condition that the root vertex of the result comes from y .

The usual symbol for this operation is $\succ$.

See also

- `product_on_basis()`, `prec_product_on_basis()`

EXAMPLES:

```
sage: A = algebras.FreeDendriform(QQ)
sage: RT = A.basis().keys()
sage: x = RT([])
sage: A.succ_product_on_basis(x, x)
B[[[., .], .]]
```

```
>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ)
>>> RT = A.basis().keys()
>>> x = RT([])
>>> A.succ_product_on_basis(x, x)
B[[[., .], .]]
```

under ()

Return the under product.

The over product $x \setminus y$ is the binary tree obtained by grafting the root of x at the leftmost leaf of y .

The usual symbol for this operation is $\setminus$.

See also

`product()`, `succ()`, `prec()`, `over()`

EXAMPLES:

```
sage: A = algebras.FreeDendriform(QQ)
sage: RT = A.basis().keys()
sage: x = A.gen(0)
sage: A.under(x, x)
B[[[., .], .]]
```

```
>>> from sage.all import *
>>> A = algebras.FreeDendriform(QQ)
>>> RT = A.basis().keys()
>>> x = A.gen(Integer(0))
>>> A.under(x, x)
B[[[., .], .]]
```

`variable_names()`

Return the names of the variables.

EXAMPLES:

```
sage: R = algebras.FreeDendriform(QQ, 'xy')
sage: R.variable_names()
{'x', 'y'}
```

```
>>> from sage.all import *
>>> R = algebras.FreeDendriform(QQ, 'xy')
>>> R.variable_names()
{'x', 'y'}
```

9.6 Free Pre-Lie Algebras

AUTHORS:

- Florent Hivert, Frédéric Chapoton (2011)

`class sage.combinat.free_prelie_algebra.FreePreLieAlgebra(R, names=None)`

Bases: `CombinatorialFreeModule`

The free pre-Lie algebra.

Pre-Lie algebras are non-associative algebras, where the product $*$ satisfies

$$(x * y) * z - x * (y * z) = (x * z) * y - x * (z * y).$$

We use here the convention where the associator

$$(x, y, z) := (x * y) * z - x * (y * z)$$

is symmetric in its two rightmost arguments. This is sometimes called a right pre-Lie algebra.

They have appeared in numerical analysis and deformation theory.

The free Pre-Lie algebra on a given set E has an explicit description using rooted trees, just as the free associative algebra can be described using words. The underlying vector space has a basis indexed by finite rooted trees endowed with a map from their vertices to E . In this basis, the product of two (decorated) rooted trees $S * T$ is the sum over vertices of S of the rooted tree obtained by adding one edge from the root of T to the given vertex of S . The root of these trees is taken to be the root of S . The free pre-Lie algebra can also be considered as the free algebra over the PreLie operad.

Warning

The usual binary operator $*$ can be used for the pre-Lie product. Beware that it but must be parenthesized properly, as the pre-Lie product is not associative. By default, a multiple product will be taken with left parentheses.

EXAMPLES:

```
sage: F = algebras.FreePreLie(ZZ, 'xyz')
sage: x, y, z = F.gens()
sage: (x * y) * z
```

(continues on next page)

(continued from previous page)

```
B[x[y[z[]]]] + B[x[y[], z[]]]
sage: (x * y) * z - x * (y * z) == (x * z) * y - x * (z * y)
True
```

```
>>> from sage.all import *
>>> F = algebras.FreePreLie(ZZ, 'xyz')
>>> x,y,z = F.gens()
>>> (x * y) * z
B[x[y[z[]]]] + B[x[y[], z[]]]
>>> (x * y) * z - x * (y * z) == (x * z) * y - x * (z * y)
True
```

The free pre-Lie algebra is non-associative:

```
sage: x * (y * z) == (x * y) * z
False
```

```
>>> from sage.all import *
>>> x * (y * z) == (x * y) * z
False
```

The default product is with left parentheses:

```
sage: x * y * z == (x * y) * z
True
sage: x * y * z * x == ((x * y) * z) * x
True
```

```
>>> from sage.all import *
>>> x * y * z == (x * y) * z
True
>>> x * y * z * x == ((x * y) * z) * x
True
```

The NAP product as defined in [Liv2006] is also implemented on the same vector space:

```
sage: N = F.nap_product
sage: N(x*y,z*z)
B[x[y[], z[z[]]]]
```

```
>>> from sage.all import *
>>> N = F.nap_product
>>> N(x*y,z*z)
B[x[y[], z[z[]]]]
```

When None is given as input, unlabelled trees are used instead:

```
sage: F1 = algebras.FreePreLie(QQ, None)
sage: w = F1.gen(0); w
B[[]]
sage: w * w * w * w
B[[[[[[]]]]]] + B[[[[]], []]] + 3*B[[[]], [[]]] + B[[[]], [], []]
```

```
>>> from sage.all import *
>>> F1 = algebras.FreePreLie(QQ, None)
>>> w = F1.gen(Integer(0)); w
B[[]]
>>> w * w * w * w
B[[[[[[]]]]]] + B[[[[]], []]] + 3*B[[[]], [[]]] + B[[[]], [], []]
```

However, it is equally possible to use labelled trees instead:

```
sage: F1 = algebras.FreePreLie(QQ, 'q')
sage: w = F1.gen(0); w
B[q[]]
sage: w * w * w * w
B[q[q[q[q[]]]]] + B[q[q[q[]], q[]]] + 3*B[q[q[]], q[q[]]] + B[q[q[]], q[], q[]]
```

```
>>> from sage.all import *
>>> F1 = algebras.FreePreLie(QQ, 'q')
>>> w = F1.gen(Integer(0)); w
B[q[]]
>>> w * w * w * w
B[q[q[q[q[]]]]] + B[q[q[q[]], q[]]] + 3*B[q[q[]], q[q[]]] + B[q[q[]], q[], q[]]
```

The set E can be infinite:

```
sage: F = algebras.FreePreLie(QQ, ZZ)
sage: w = F.gen(1); w
B[1[]]
sage: x = F.gen(2); x
B[-1[]]
sage: y = F.gen(3); y
B[2[]]
sage: w*x
B[1[-1[]]]
sage: (w*x)*y
B[1[-1[2[]]]] + B[1[-1[], 2[]]]
sage: w*(x*y)
B[1[-1[2[]]]]
```

```
>>> from sage.all import *
>>> F = algebras.FreePreLie(QQ, ZZ)
>>> w = F.gen(Integer(1)); w
B[1[]]
>>> x = F.gen(Integer(2)); x
B[-1[]]
>>> y = F.gen(Integer(3)); y
B[2[]]
>>> w*x
B[1[-1[]]]
>>> (w*x)*y
B[1[-1[2[]]]] + B[1[-1[], 2[]]]
>>> w*(x*y)
B[1[-1[2[]]]]
```


Elements of a free pre-Lie algebra can be lifted to the universal enveloping algebra of the associated Lie algebra. The universal enveloping algebra is the Grossman-Larson Hopf algebra:

```
sage: F = algebras.FreePreLie(QQ, 'abc')
sage: a,b,c = F.gens()
sage: (a*b+b*c).lift()
B[#[a[b[]]]] + B[#[b[c[]]]]
```

```
>>> from sage.all import *
>>> F = algebras.FreePreLie(QQ, 'abc')
>>> a,b,c = F.gens()
>>> (a*b+b*c).lift()
B[#[a[b[]]]] + B[#[b[c[]]]]
```

Note

Variables names can be `None`, a list of strings, a string or an integer. When `None` is given, unlabelled rooted trees are used. When a single string is given, each letter is taken as a variable. See `sage.combinat.words.alphabet.build_alphabet()`.

Warning

Beware that the underlying combinatorial free module is based either on `RootedTrees` or on `Labelled-RootedTrees`, with no restriction on the labellings. This means that all code calling the `basis()` method would not give meaningful results, since `basis()` returns many “chaff” elements that do not belong to the algebra.

REFERENCES:

- [ChLi]
- [Liv2006]

class Element

Bases: `IndexedFreeModuleElement`

lift()

Lift element to the Grossman-Larson algebra.

EXAMPLES:

```
sage: F = algebras.FreePreLie(QQ, 'abc')
sage: elt = F.an_element().lift(); elt
B[#[a[a[a[a[]]]]]] + B[#[a[a[], a[a[]]]]]
sage: parent(elt)
Grossman-Larson Hopf algebra on 3 generators ['a', 'b', 'c']
over Rational Field
```

```
>>> from sage.all import *
>>> F = algebras.FreePreLie(QQ, 'abc')
>>> elt = F.an_element().lift(); elt
B[#[a[a[a[a[]]]]]] + B[#[a[a[], a[a[]]]]]
```

(continues on next page)

(continued from previous page)

```
>>> parent(elt)
Grossman-Larson Hopf algebra on 3 generators ['a', 'b', 'c']
over Rational Field
```

valuation()

Return the valuation of self.

EXAMPLES:

```
sage: a = algebras.FreePreLie(QQ).gen(0)
sage: a.valuation()
1
sage: (a*a).valuation()
2

sage: a, b = algebras.FreePreLie(QQ, 'ab').gens()
sage: (a+b).valuation()
1
sage: (a*b).valuation()
2
sage: (a*b+a).valuation()
1
```

```
>>> from sage.all import *
>>> a = algebras.FreePreLie(QQ).gen(Integer(0))
>>> a.valuation()
1
>>> (a*a).valuation()
2

>>> a, b = algebras.FreePreLie(QQ, 'ab').gens()
>>> (a+b).valuation()
1
>>> (a*b).valuation()
2
>>> (a*b+a).valuation()
1
```

algebra_generators()

Return the generators of this algebra.

These are the rooted trees with just one vertex.

EXAMPLES:

```
sage: A = algebras.FreePreLie(ZZ, 'fgh'); A
Free PreLie algebra on 3 generators ['f', 'g', 'h']
over Integer Ring
sage: list(A.algebra_generators())
[B[f[]], B[g[]], B[h[]]]

sage: A = algebras.FreePreLie(QQ, ['x1', 'x2'])
sage: list(A.algebra_generators())
[B[x1[]], B[x2[]]]
```

```

>>> from sage.all import *
>>> A = algebras.FreePreLie(ZZ, 'fgh'); A
Free PreLie algebra on 3 generators ['f', 'g', 'h']
over Integer Ring
>>> list(A.algebra_generators())
[B[f[]], B[g[]], B[h[]]]

>>> A = algebras.FreePreLie(QQ, ['x1', 'x2'])
>>> list(A.algebra_generators())
[B[x1[]], B[x2[]]]

```

an_element()

Return an element of self.

EXAMPLES:

```

sage: A = algebras.FreePreLie(QQ, 'xy')
sage: A.an_element()
B[x[x[x[x[]]]]] + B[x[x[]], x[x[]]]

```

```

>>> from sage.all import *
>>> A = algebras.FreePreLie(QQ, 'xy')
>>> A.an_element()
B[x[x[x[x[x[]]]]]] + B[x[x[]], x[x[]]]

```

bracket_on_basis(x, y)

Return the Lie bracket of two trees.

This is the commutator $[x, y] = x * y - y * x$ of the pre-Lie product.

See also

[`pre\_Lie\_product\_on\_basis\(\)`](#)

EXAMPLES:

```

sage: A = algebras.FreePreLie(QQ, None)
sage: RT = A.basis().keys()
sage: x = RT([RT([])])
sage: y = RT([x])
sage: A.bracket_on_basis(x, y)
-B[[[]], [[]]] + B[[[]], [[]]] - B[[[]], [[]]]

```

```

>>> from sage.all import *
>>> A = algebras.FreePreLie(QQ, None)
>>> RT = A.basis().keys()
>>> x = RT([RT([])])
>>> y = RT([x])
>>> A.bracket_on_basis(x, y)
-B[[[]], [[]]] + B[[[]], [[]]] - B[[[]], [[]]]

```

change_ring(R)

Return the free pre-Lie algebra in the same variables over R .

INPUT:

- R – a ring

EXAMPLES:

```
sage: A = algebras.FreePreLie(ZZ, 'fgh')
sage: A.change_ring(QQ)
Free PreLie algebra on 3 generators ['f', 'g', 'h'] over
Rational Field
```

```
>>> from sage.all import *
>>> A = algebras.FreePreLie(ZZ, 'fgh')
>>> A.change_ring(QQ)
Free PreLie algebra on 3 generators ['f', 'g', 'h'] over
Rational Field
```

construction()

Return a pair (F, R) , where F is a *PreLieFunctor* and R is a ring, such that $F(R)$ returns self.

EXAMPLES:

```
sage: P = algebras.FreePreLie(ZZ, 'x,y')
sage: x,y = P.gens()
sage: F, R = P.construction()
sage: F
PreLie[x,y]
sage: R
Integer Ring
sage: F(ZZ) is P
True
sage: F(QQ)
Free PreLie algebra on 2 generators ['x', 'y'] over Rational Field
```

```
>>> from sage.all import *
>>> P = algebras.FreePreLie(ZZ, 'x,y')
>>> x,y = P.gens()
>>> F, R = P.construction()
>>> F
PreLie[x,y]
>>> R
Integer Ring
>>> F(ZZ) is P
True
>>> F(QQ)
Free PreLie algebra on 2 generators ['x', 'y'] over Rational Field
```

corolla(x, y, n, N)

Return the corolla obtained with x as root and y as leaves.

INPUT:

- x, y – two elements
- n – integer; width of the corolla
- N – integer; truncation order (up to order N included)

OUTPUT:

the sum over all possible ways to graft n copies of y on top of x (with at most N vertices in total)

This operation can be defined by induction starting from the pre-Lie product.

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ)
sage: a = A.gen(0)
sage: b = A.corolla(a,a,1,4); b
B[[[]]]
sage: A.corolla(b,b,2,7)
B[[[[[]], [[]]]] + 2*B[[[[]], [[]]]] + B[[[]], [[]], [[]]]

sage: A = algebras.FreePreLie(QQ, 'o')
sage: a = A.gen(0)
sage: b = A.corolla(a,a,1,4)

sage: A = algebras.FreePreLie(QQ, 'ab')
sage: a, b = A.gens()
sage: A.corolla(a,b,1,4)
B[a[b[]]]
sage: A.corolla(b,a,3,4)
B[b[a[], a[], a[]]]

sage: A.corolla(a+b,a+b,2,4)
B[a[a[], a[]] + 2*B[a[a[], b[]]] + B[a[b[], b[]]] + B[b[a[], a[]]] +
2*B[b[a[], b[]]] + B[b[b[], b[]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreePreLie(QQ)
>>> a = A.gen(Integer(0))
>>> b = A.corolla(a,a,Integer(1),Integer(4)); b
B[[[]]]
>>> A.corolla(b,b,Integer(2),Integer(7))
B[[[[[]], [[]]]] + 2*B[[[[]], [[]]]] + B[[[]], [[]], [[]]]

>>> A = algebras.FreePreLie(QQ, 'o')
>>> a = A.gen(Integer(0))
>>> b = A.corolla(a,a,Integer(1),Integer(4))

>>> A = algebras.FreePreLie(QQ, 'ab')
>>> a, b = A.gens()
>>> A.corolla(a,b,Integer(1),Integer(4))
B[a[b[]]]
>>> A.corolla(b,a,Integer(3),Integer(4))
B[b[a[], a[], a[]]]

>>> A.corolla(a+b,a+b,Integer(2),Integer(4))
B[a[a[], a[]] + 2*B[a[a[], b[]]] + B[a[b[], b[]]] + B[b[a[], a[]]] +
2*B[b[a[], b[]]] + B[b[b[], b[]]]
```

degree_on_basis(t)

Return the degree of a rooted tree in the free Pre-Lie algebra.

This is the number of vertices.

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, None)
sage: RT = A.basis().keys()
sage: A.degree_on_basis(RT([RT([])]))
2
```

```
>>> from sage.all import *
>>> A = algebras.FreePreLie(QQ, None)
>>> RT = A.basis().keys()
>>> A.degree_on_basis(RT([RT([])]))
2
```

gen(*i*)

Return the *i*-th generator of the algebra.

INPUT:

- *i* – integer

EXAMPLES:

```
sage: F = algebras.FreePreLie(ZZ, 'xyz')
sage: F.gen(0)
B[x[]]

sage: F.gen(4)
Traceback (most recent call last):
...
IndexError: argument i (= 4) must be between 0 and 2
```

```
>>> from sage.all import *
>>> F = algebras.FreePreLie(ZZ, 'xyz')
>>> F.gen(Integer(0))
B[x[]]

>>> F.gen(Integer(4))
Traceback (most recent call last):
...
IndexError: argument i (= 4) must be between 0 and 2
```

gens()

Return the generators of *self* (as an algebra).

EXAMPLES:

```
sage: A = algebras.FreePreLie(ZZ, 'fgh')
sage: A.gens()
(B[f[]], B[g[]], B[h[]])
```

```
>>> from sage.all import *
>>> A = algebras.FreePreLie(ZZ, 'fgh')
>>> A.gens()
(B[f[]], B[g[]], B[h[]])
```

group_product ($x, y, n, N=10$)

Return the truncated group product of x and y .

This is a weighted sum of all corollas with up to n leaves, with x as root and y as leaves.

The result is computed up to order N (included).

When considered with infinitely many terms and infinite precision, this is an analogue of the Baker-Campbell-Hausdorff formula: it defines an associative product on the completed free pre-Lie algebra.

INPUT:

- x, y – two elements
- n – integer; the maximal width of corollas
- N – integer (default: 10); truncation order

EXAMPLES:

In the free pre-Lie algebra with one generator:

```
sage: PL = algebras.FreePreLie(QQ)
sage: a = PL.gen(0)
sage: PL.group_product(a, a, 3, 3)
B[[]] + B[[[]]] + 1/2*B[[[]], []]
```

```
>>> from sage.all import *
>>> PL = algebras.FreePreLie(QQ)
>>> a = PL.gen(Integer(0))
>>> PL.group_product(a, a, Integer(3), Integer(3))
B[[]] + B[[[]]] + 1/2*B[[[]], []]
```

In the free pre-Lie algebra with several generators:

```
sage: PL = algebras.FreePreLie(QQ, '@O')
sage: a, b = PL.gens()
sage: PL.group_product(a, b, 3, 3)
B[@[]] + B[@[O[]]] + 1/2*B[@[O[]], O[]]
sage: PL.group_product(a, b, 3, 10)
B[@[]] + B[@[O[]]] + 1/2*B[@[O[]], O[]] + 1/6*B[@[O[]], O[], O[]]
```

```
>>> from sage.all import *
>>> PL = algebras.FreePreLie(QQ, '@O')
>>> a, b = PL.gens()
>>> PL.group_product(a, b, Integer(3), Integer(3))
B[@[]] + B[@[O[]]] + 1/2*B[@[O[]], O[]]
>>> PL.group_product(a, b, Integer(3), Integer(10))
B[@[]] + B[@[O[]]] + 1/2*B[@[O[]], O[]] + 1/6*B[@[O[]], O[], O[]]
```

nap_product ()

Return the NAP product.

See also

`nap_product_on_basis()`

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, None)
sage: RT = A.basis().keys()
sage: x = A(RT([RT([])]))
sage: A.nap_product(x, x)
B[[[]], [[]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreePreLie(QQ, None)
>>> RT = A.basis().keys()
>>> x = A(RT([RT([])]))
>>> A.nap_product(x, x)
B[[[]], [[]]]
```

nap_product_on_basis(x, y)

Return the NAP product of two trees.

This is the grafting of the root of y over the root of x . The root of the resulting tree is the root of x .**See also***nap_product()*

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, None)
sage: RT = A.basis().keys()
sage: x = RT([RT([])])
sage: A.nap_product_on_basis(x, x)
B[[[]], [[]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreePreLie(QQ, None)
>>> RT = A.basis().keys()
>>> x = RT([RT([])])
>>> A.nap_product_on_basis(x, x)
B[[[]], [[]]]
```

pre_Lie_product()

Return the pre-Lie product.

See also*pre_Lie_product_on_basis()*

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, None)
sage: RT = A.basis().keys()
sage: x = A(RT([RT([])]))
```

(continues on next page)

(continued from previous page)

```
sage: A.pre_Lie_product(x, x)
B[[[[[]]]]] + B[[[]], [[]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreePreLie(QQ, None)
>>> RT = A.basis().keys()
>>> x = A(RT([RT([])])
>>> A.pre_Lie_product(x, x)
B[[[[[]]]]] + B[[[]], [[]]]
```

pre_Lie_product_on_basis(x, y)

Return the pre-Lie product of two trees.

This is the sum over all graftings of the root of y over a vertex of x . The root of the resulting trees is the root of x .

See also

[`pre\_Lie\_product\(\)`](#)

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, None)
sage: RT = A.basis().keys()
sage: x = RT([RT([])])
sage: A.product_on_basis(x, x)
B[[[[[]]]]] + B[[[]], [[]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreePreLie(QQ, None)
>>> RT = A.basis().keys()
>>> x = RT([RT([])])
>>> A.product_on_basis(x, x)
B[[[[[]]]]] + B[[[]], [[]]]
```

product_on_basis(x, y)

Return the pre-Lie product of two trees.

This is the sum over all graftings of the root of y over a vertex of x . The root of the resulting trees is the root of x .

See also

[`pre\_Lie\_product\(\)`](#)

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, None)
sage: RT = A.basis().keys()
sage: x = RT([RT([])])
```

(continues on next page)

(continued from previous page)

```
sage: A.product_on_basis(x, x)
B[[[[]]]] + B[[[]], [[]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreePreLie(QQ, None)
>>> RT = A.basis().keys()
>>> x = RT([RT([])])
>>> A.product_on_basis(x, x)
B[[[[]]]] + B[[[]], [[]]]
```

some_elements()

Return some elements of the free pre-Lie algebra.

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, None)
sage: A.some_elements()
[B[[[]], B[[[]]], B[[[[]]]]] + B[[[]], [[]]], B[[[[]]]] + B[[[]], []], ↵
↵B[[[[]]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreePreLie(QQ, None)
>>> A.some_elements()
[B[[[]], B[[[]]], B[[[[]]]]] + B[[[]], [[]]], B[[[[]]]] + B[[[]], []], ↵
↵B[[[[]]]]
```

With several generators:

```
sage: A = algebras.FreePreLie(QQ, 'xy')
sage: A.some_elements()
[B[x[]],
 B[x[x[]]],
 B[x[x[x[x[]]]] + B[x[x[], x[x[]]]],
 B[x[x[x[]]]] + B[x[x[], x[]]],
 B[x[x[y[]]]] + B[x[x[], y[]]]]
```

```
>>> from sage.all import *
>>> A = algebras.FreePreLie(QQ, 'xy')
>>> A.some_elements()
[B[x[]],
 B[x[x[]]],
 B[x[x[x[x[]]]] + B[x[x[], x[x[]]]],
 B[x[x[x[]]]] + B[x[x[], x[]]],
 B[x[x[y[]]]] + B[x[x[], y[]]]]
```

variable_names()

Return the names of the variables.

EXAMPLES:

```
sage: R = algebras.FreePreLie(QQ, 'xy')
sage: R.variable_names()
```

(continues on next page)

(continued from previous page)

```
{'x', 'y'}

sage: R = algebras.FreePreLie(QQ, None)
sage: R.variable_names()
{'o'}
```

```
>>> from sage.all import *
>>> R = algebras.FreePreLie(QQ, 'xy')
>>> R.variable_names()
{'x', 'y'}

>>> R = algebras.FreePreLie(QQ, None)
>>> R.variable_names()
{'o'}
```

```
class sage.combinat.free_prelie_algebra.PreLieFunctor(vars)
```

Bases: `ConstructionFunctor`

A constructor for pre-Lie algebras.

EXAMPLES:

```
sage: P = algebras.FreePreLie(ZZ, 'x,y')
sage: x,y = P.gens()
sage: F = P.construction()[0]; F
PreLie[x,y]

sage: A = GF(5)['a,b']
sage: a, b = A.gens()
sage: F(A)
Free PreLie algebra on 2 generators ['x', 'y'] over Multivariate Polynomial Ring_
↳in a, b over Finite Field of size 5

sage: f = A.hom([a+b,a-b],A)
sage: F(f)
Generic endomorphism of Free PreLie algebra on 2 generators ['x', 'y']
over Multivariate Polynomial Ring in a, b over Finite Field of size 5

sage: F(f)(a * F(A)(x))
(a+b)*B[x[]]
```

```
>>> from sage.all import *
>>> P = algebras.FreePreLie(ZZ, 'x,y')
>>> x,y = P.gens()
>>> F = P.construction()[Integer(0)]; F
PreLie[x,y]

>>> A = GF(Integer(5))['a,b']
>>> a, b = A.gens()
>>> F(A)
Free PreLie algebra on 2 generators ['x', 'y'] over Multivariate Polynomial Ring_
↳in a, b over Finite Field of size 5
```

(continues on next page)

(continued from previous page)

```

>>> f = A.hom([a+b,a-b],A)
>>> F(f)
Generic endomorphism of Free PreLie algebra on 2 generators ['x', 'y']
over Multivariate Polynomial Ring in a, b over Finite Field of size 5

>>> F(f) (a * F(A) (x))
(a+b)*B[x[]]

```

merge (*other*)

Merge self with another construction functor, or return None.

EXAMPLES:

```

sage: F = sage.combinat.free_prelie_algebra.PreLieFunctor(['x','y'])
sage: G = sage.combinat.free_prelie_algebra.PreLieFunctor(['t'])
sage: F.merge(G)
PreLie[x,y,t]
sage: F.merge(F)
PreLie[x,y]

```

```

>>> from sage.all import *
>>> F = sage.combinat.free_prelie_algebra.PreLieFunctor(['x','y'])
>>> G = sage.combinat.free_prelie_algebra.PreLieFunctor(['t'])
>>> F.merge(G)
PreLie[x,y,t]
>>> F.merge(F)
PreLie[x,y]

```

Now some actual use cases:

```

sage: R = algebras.FreePreLie(ZZ, 'xyz')
sage: x,y,z = R.gens()
sage: 1/2 * x
1/2*B[x[]]
sage: parent(1/2 * x)
Free PreLie algebra on 3 generators ['x', 'y', 'z'] over Rational Field

sage: S = algebras.FreePreLie(QQ, 'zt')
sage: z,t = S.gens()
sage: x + t
B[t[]] + B[x[]]
sage: parent(x + t)
Free PreLie algebra on 4 generators ['z', 't', 'x', 'y'] over Rational Field

```

```

>>> from sage.all import *
>>> R = algebras.FreePreLie(ZZ, 'xyz')
>>> x,y,z = R.gens()
>>> Integer(1)/Integer(2) * x
1/2*B[x[]]
>>> parent(Integer(1)/Integer(2) * x)
Free PreLie algebra on 3 generators ['x', 'y', 'z'] over Rational Field

```

(continues on next page)

(continued from previous page)

```

>>> S = algebras.FreePreLie(QQ, 'zt')
>>> z,t = S.gens()
>>> x + t
B[t[]] + B[x[]]
>>> parent(x + t)
Free PreLie algebra on 4 generators ['z', 't', 'x', 'y'] over Rational Field

```

rank = 9

`sage.combinat.free_prelie_algebra.corolla_gen(tx, list_ty, labels=True)`

Yield the terms in the corolla with given bottom tree and top trees.

These are the possible terms in the simultaneous grafting of the top trees on vertices of the bottom tree.

INPUT:

- `tx` – a tree
- `list_ty` – list of trees

EXAMPLES:

```

sage: from sage.combinat.free_prelie_algebra import corolla_gen
sage: a = algebras.FreePreLie(QQ).gen(0)
sage: ta = a.support()[0]
sage: list(corolla_gen(ta, [ta], False))
[[]]

sage: a, b = algebras.FreePreLie(QQ, 'ab').gens()
sage: ta = a.support()[0]
sage: tb = b.support()[0]
sage: ab = (a*b).support()[0]
sage: list(corolla_gen(ta, [tb]))
[a[b[]]]
sage: list(corolla_gen(tb, [ta, ta]))
[b[a[]], a[]]
sage: list(corolla_gen(ab, [ab, ta]))
[a[a[]], b[], a[b[]], a[a[b[]], b[a[]], a[a[], b[a[b[]]]],
a[b[a[], a[b[]]]]]

```

```

>>> from sage.all import *
>>> from sage.combinat.free_prelie_algebra import corolla_gen
>>> a = algebras.FreePreLie(QQ).gen(Integer(0))
>>> ta = a.support()[Integer(0)]
>>> list(corolla_gen(ta, [ta], False))
[[]]

>>> a, b = algebras.FreePreLie(QQ, 'ab').gens()
>>> ta = a.support()[Integer(0)]
>>> tb = b.support()[Integer(0)]
>>> ab = (a*b).support()[Integer(0)]
>>> list(corolla_gen(ta, [tb]))
[a[b[]]]

```

(continues on next page)

(continued from previous page)

```
>>> list(corolla_gen(tb, [ta, ta]))
[b[a[], a[]]]
>>> list(corolla_gen(ab, [ab, ta]))
[a[a[], b[], a[b[]]], a[a[b[]], b[a[]]], a[a[], b[a[b[]]]],
a[b[a[]], a[b[]]]]
```

sage.combinat.free_prelie_algebra.**tree_from_sortkey**(ch, labels=True)

Transform a list of (valence, label) into a tree and a remainder.

This is like an inverse of the sort_key method.

INPUT:

- ch – list of pairs (integer, label)
- labels – boolean (default: True); whether to use labelled trees

OUTPUT:

a pair (tree, remainder of the input)

EXAMPLES:

```
sage: from sage.combinat.free_prelie_algebra import tree_from_sortkey
sage: a = algebras.FreePreLie(QQ).gen(0)
sage: t = (a*a*a*a).support()
sage: all(tree_from_sortkey(u.sort_key(), False)[0] == u for u in t)
True

sage: a, b = algebras.FreePreLie(QQ, 'ab').gens()
sage: t = (a*b*a*b).support()
sage: all(tree_from_sortkey(u.sort_key())[0] == u for u in t)
True
```

```
>>> from sage.all import *
>>> from sage.combinat.free_prelie_algebra import tree_from_sortkey
>>> a = algebras.FreePreLie(QQ).gen(Integer(0))
>>> t = (a*a*a*a).support()
>>> all(tree_from_sortkey(u.sort_key(), False)[Integer(0)] == u for u in t)
True

>>> a, b = algebras.FreePreLie(QQ, 'ab').gens()
>>> t = (a*b*a*b).support()
>>> all(tree_from_sortkey(u.sort_key())[Integer(0)] == u for u in t)
True
```

9.7 Shuffle algebras

AUTHORS:

- Frédéric Chapoton (2013-03): Initial version
- Matthieu Deneufchatel (2013-07): Implemented dual PBW basis

```
class sage.algebras.shuffle_algebra.DualPBWBasis (R, names)
```

Bases: `CombinatorialFreeModule`

The basis dual to the Poincaré-Birkhoff-Witt basis of the free algebra.

We recursively define the dual PBW basis as the basis of the shuffle algebra given by

$$S_w = \begin{cases} w & |w| = 1, \\ xS_u & w = xu \text{ and } w \in \text{Lyn}(X), \\ \frac{S_{\ell_{i_1}}^{*\alpha_1} * \dots * S_{\ell_{i_k}}^{*\alpha_k}}{\alpha_1! \dots \alpha_k!} & w = \ell_{i_1}^{\alpha_1} \dots \ell_{i_k}^{\alpha_k} \text{ with } \ell_1 > \dots > \ell_k \in \text{Lyn}(X). \end{cases}$$

where $S * T$ denotes the shuffle product of S and T and $\text{Lyn}(X)$ is the set of Lyndon words in the alphabet X .

The definition may be found in Theorem 5.3 of [Reu1993].

INPUT:

- `R` – ring
- `names` – names of the generators (string or an alphabet)

EXAMPLES:

```
sage: S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
sage: S
The dual Poincare-Birkhoff-Witt basis of Shuffle Algebra on 2 generators ['a', 'b
↪'] over Rational Field
sage: S.one()
S[]
sage: S.one_basis()
word:
sage: T = ShuffleAlgebra(QQ, 'abcd').dual_pbw_basis(); T
The dual Poincare-Birkhoff-Witt basis of Shuffle Algebra on 4 generators ['a', 'b
↪', 'c', 'd'] over Rational Field
sage: T.algebra_generators()
(S[a], S[b], S[c], S[d])
```

```
>>> from sage.all import *
>>> S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
>>> S
The dual Poincare-Birkhoff-Witt basis of Shuffle Algebra on 2 generators ['a', 'b
↪'] over Rational Field
>>> S.one()
S[]
>>> S.one_basis()
word:
>>> T = ShuffleAlgebra(QQ, 'abcd').dual_pbw_basis(); T
The dual Poincare-Birkhoff-Witt basis of Shuffle Algebra on 4 generators ['a', 'b
↪', 'c', 'd'] over Rational Field
>>> T.algebra_generators()
(S[a], S[b], S[c], S[d])
```

```
class Element
```

Bases: `IndexedFreeModuleElement`

An element in the dual PBW basis.

expand()Expand `self` in words of the shuffle algebra.

EXAMPLES:

```
sage: S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
sage: f = S('ab') + S('bab')
sage: f.expand()
B[ab] + 2*B[abb] + B[bab]
```

```
>>> from sage.all import *
>>> S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
>>> f = S('ab') + S('bab')
>>> f.expand()
B[ab] + 2*B[abb] + B[bab]
```

algebra_generators()Return the algebra generators of `self`.

EXAMPLES:

```
sage: S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
sage: S.algebra_generators()
(S[a], S[b])
```

```
>>> from sage.all import *
>>> S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
>>> S.algebra_generators()
(S[a], S[b])
```

antipode(elt)Return the antipode of the element `elt`.

EXAMPLES:

```
sage: A = ShuffleAlgebra(QQ, 'ab')
sage: S = A.dual_pbw_basis()
sage: w = S('abaab').antipode(); w
S[abaab] - 2*S[ababa] - S[baaba]
+ 3*S[babaa] - 6*S[bbaaa]
sage: w.antipode()
S[abaab]
```

```
>>> from sage.all import *
>>> A = ShuffleAlgebra(QQ, 'ab')
>>> S = A.dual_pbw_basis()
>>> w = S('abaab').antipode(); w
S[abaab] - 2*S[ababa] - S[baaba]
+ 3*S[babaa] - 6*S[bbaaa]
>>> w.antipode()
S[abaab]
```

coproduct(elt)Return the coproduct of the element `elt`.

EXAMPLES:

```

sage: A = ShuffleAlgebra(QQ, 'ab')
sage: S = A.dual_pbw_basis()
sage: S('ab').coproduct()
S[] # S[ab] + S[a] # S[b]
+ S[ab] # S[]
sage: S('ba').coproduct()
S[] # S[ba] + S[a] # S[b]
+ S[b] # S[a] + S[ba] # S[]

```

```

>>> from sage.all import *
>>> A = ShuffleAlgebra(QQ, 'ab')
>>> S = A.dual_pbw_basis()
>>> S('ab').coproduct()
S[] # S[ab] + S[a] # S[b]
+ S[ab] # S[]
>>> S('ba').coproduct()
S[] # S[ba] + S[a] # S[b]
+ S[b] # S[a] + S[ba] # S[]

```

counit(S)

Return the counit of S.

EXAMPLES:

```

sage: F = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
sage: (3*F.gen(0)+5*F.gen(1)**2).counit()
0
sage: (4*F.one()).counit()
4

```

```

>>> from sage.all import *
>>> F = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
>>> (Integer(3)*F.gen(Integer(0))+Integer(5)*F.gen(Integer(1))**Integer(2)).
↪counit()
0
>>> (Integer(4)*F.one()).counit()
4

```

degree_on_basis(w)

Return the degree of the element w.

EXAMPLES:

```

sage: S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
sage: [S.degree_on_basis(x.leading_support()) for x in S.some_elements() if x_
↪!= 0]
[0, 1, 1, 2]

```

```

>>> from sage.all import *
>>> S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
>>> [S.degree_on_basis(x.leading_support()) for x in S.some_elements() if x !

```

(continues on next page)

(continued from previous page)

```

↪ = Integer(0)]
[0, 1, 1, 2]

```

expansion()

Return the morphism corresponding to the expansion into words of the shuffle algebra.

EXAMPLES:

```

sage: S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
sage: f = S('ab') + S('aba')
sage: S.expansion(f)
2*B[aab] + B[ab] + B[aba]

```

```

>>> from sage.all import *
>>> S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
>>> f = S('ab') + S('aba')
>>> S.expansion(f)
2*B[aab] + B[ab] + B[aba]

```

expansion_on_basis(w)

Return the expansion of S_w in words of the shuffle algebra.

INPUT:

- w – a word

EXAMPLES:

```

sage: S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
sage: S.expansion_on_basis(Word())
B[]
sage: S.expansion_on_basis(Word()).parent()
Shuffle Algebra on 2 generators ['a', 'b'] over Rational Field
sage: S.expansion_on_basis(Word('abba'))
2*B[aabb] + B[abab] + B[abba]
sage: S.expansion_on_basis(Word())
B[]
sage: S.expansion_on_basis(Word('abab'))
2*B[aabb] + B[abab]

```

```

>>> from sage.all import *
>>> S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
>>> S.expansion_on_basis(Word())
B[]
>>> S.expansion_on_basis(Word()).parent()
Shuffle Algebra on 2 generators ['a', 'b'] over Rational Field
>>> S.expansion_on_basis(Word('abba'))
2*B[aabb] + B[abab] + B[abba]
>>> S.expansion_on_basis(Word())
B[]
>>> S.expansion_on_basis(Word('abab'))
2*B[aabb] + B[abab]

```

gen(i)

Return the i -th generator of `self`.

EXAMPLES:

```
sage: S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
sage: S.gen(0)
S[a]
sage: S.gen(1)
S[b]
```

```
>>> from sage.all import *
>>> S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
>>> S.gen(Integer(0))
S[a]
>>> S.gen(Integer(1))
S[b]
```

gens()

Return the algebra generators of `self`.

EXAMPLES:

```
sage: S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
sage: S.algebra_generators()
(S[a], S[b])
```

```
>>> from sage.all import *
>>> S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
>>> S.algebra_generators()
(S[a], S[b])
```

one_basis()

Return the indexing element of the basis element 1.

EXAMPLES:

```
sage: S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
sage: S.one_basis()
word:
```

```
>>> from sage.all import *
>>> S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
>>> S.one_basis()
word:
```

product(u, v)

Return the product of two elements u and v .

EXAMPLES:

```
sage: S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
sage: a, b = S.gens()
sage: S.product(a, b)
```

(continues on next page)

(continued from previous page)

```

S[ba]
sage: S.product(b, a)
S[ba]
sage: S.product(b^2*a, a*b*a)
36*S[bbbbaaa]

```

```

>>> from sage.all import *
>>> S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
>>> a,b = S.gens()
>>> S.product(a, b)
S[ba]
>>> S.product(b, a)
S[ba]
>>> S.product(b**Integer(2)*a, a*b*a)
36*S[bbbbaaa]

```

shuffle_algebra()

Return the associated shuffle algebra of self.

EXAMPLES:

```

sage: S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
sage: S.shuffle_algebra()
Shuffle Algebra on 2 generators ['a', 'b'] over Rational Field

```

```

>>> from sage.all import *
>>> S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
>>> S.shuffle_algebra()
Shuffle Algebra on 2 generators ['a', 'b'] over Rational Field

```

some_elements()

Return some typical elements.

EXAMPLES:

```

sage: F = ShuffleAlgebra(QQ, 'xyz').dual_pbw_basis()
sage: F.some_elements()
[0, S[], S[x], S[y], S[z], S[zx]]

```

```

>>> from sage.all import *
>>> F = ShuffleAlgebra(QQ, 'xyz').dual_pbw_basis()
>>> F.some_elements()
[0, S[], S[x], S[y], S[z], S[zx]]

```

class sage.algebras.shuffle_algebra.**ShuffleAlgebra**(*R, names, prefix*)

Bases: [CombinatorialFreeModule](#)

The shuffle algebra on some generators over a base ring.

Shuffle algebras are commutative and associative algebras, with a basis indexed by words. The product of two words $w_1 \cdot w_2$ is given by the sum over the shuffle product of w_1 and w_2 .

See also

For more on shuffle products, see `shuffle_product` and `shuffle()`.

REFERENCES:

- [Wikipedia article Shuffle algebra](#)

INPUT:

- `R` – ring
- `names` – generator names (string or an alphabet)

EXAMPLES:

```
sage: F = ShuffleAlgebra(QQ, 'xyz'); F
Shuffle Algebra on 3 generators ['x', 'y', 'z'] over Rational Field

sage: mul(F.gens())
B[xyz] + B[xzy] + B[yxz] + B[yzx] + B[zxy] + B[zyx]

sage: mul([ F.gen(i) for i in range(2) ]) + mul([ F.gen(i+1) for i in range(2) ])
B[xy] + B[yx] + B[yz] + B[zy]

sage: S = ShuffleAlgebra(ZZ, 'abcabc'); S
Shuffle Algebra on 3 generators ['a', 'b', 'c'] over Integer Ring
sage: S.base_ring()
Integer Ring

sage: G = ShuffleAlgebra(S, 'mn'); G
Shuffle Algebra on 2 generators ['m', 'n'] over Shuffle Algebra on 3 generators [
↪ 'a', 'b', 'c'] over Integer Ring
sage: G.base_ring()
Shuffle Algebra on 3 generators ['a', 'b', 'c'] over Integer Ring
```

```
>>> from sage.all import *
>>> F = ShuffleAlgebra(QQ, 'xyz'); F
Shuffle Algebra on 3 generators ['x', 'y', 'z'] over Rational Field

>>> mul(F.gens())
B[xyz] + B[xzy] + B[yxz] + B[yzx] + B[zxy] + B[zyx]

>>> mul([ F.gen(i) for i in range(Integer(2)) ]) + mul([ F.gen(i+Integer(1)) for
↪ i in range(Integer(2)) ])
B[xy] + B[yx] + B[yz] + B[zy]

>>> S = ShuffleAlgebra(ZZ, 'abcabc'); S
Shuffle Algebra on 3 generators ['a', 'b', 'c'] over Integer Ring
>>> S.base_ring()
Integer Ring

>>> G = ShuffleAlgebra(S, 'mn'); G
Shuffle Algebra on 2 generators ['m', 'n'] over Shuffle Algebra on 3 generators [
↪ 'a', 'b', 'c'] over Integer Ring
```

(continues on next page)

(continued from previous page)

```
>>> G.base_ring()
Shuffle Algebra on 3 generators ['a', 'b', 'c'] over Integer Ring
```

Shuffle algebras commute with their base ring:

```
sage: K = ShuffleAlgebra(QQ, 'ab')
sage: a,b = K.gens()
sage: K.is_commutative()
True
sage: L = ShuffleAlgebra(K, 'cd')
sage: c,d = L.gens()
sage: L.is_commutative()
True
sage: s = a*b^2 * c^3; s
(12*B[abb]+12*B[bab]+12*B[bba])*B[ccc]
sage: parent(s)
Shuffle Algebra on 2 generators ['c', 'd'] over Shuffle Algebra on 2 generators [
↪ 'a', 'b'] over Rational Field
sage: c^3 * a * b^2
(12*B[abb]+12*B[bab]+12*B[bba])*B[ccc]
```

```
>>> from sage.all import *
>>> K = ShuffleAlgebra(QQ, 'ab')
>>> a,b = K.gens()
>>> K.is_commutative()
True
>>> L = ShuffleAlgebra(K, 'cd')
>>> c,d = L.gens()
>>> L.is_commutative()
True
>>> s = a*b**Integer(2) * c**Integer(3); s
(12*B[abb]+12*B[bab]+12*B[bba])*B[ccc]
>>> parent(s)
Shuffle Algebra on 2 generators ['c', 'd'] over Shuffle Algebra on 2 generators [
↪ 'a', 'b'] over Rational Field
>>> c**Integer(3) * a * b**Integer(2)
(12*B[abb]+12*B[bab]+12*B[bba])*B[ccc]
```

Shuffle algebras are commutative:

```
sage: c^3 * b * a * b == c * a * c * b^2 * c
True
```

```
>>> from sage.all import *
>>> c**Integer(3) * b * a * b == c * a * c * b**Integer(2) * c
True
```

We can also manipulate elements in the basis and coerce elements from our base field:

```
sage: F = ShuffleAlgebra(QQ, 'abc')
sage: B = F.basis()
sage: B[Word('bb')] * B[Word('ca')]
```

(continues on next page)

(continued from previous page)

```

B[bbca] + B[bcab] + B[bcba] + B[cabb]
+ B[cbab] + B[cbba]
sage: 1 - B[Word('bb')] * B[Word('ca')] / 2
B[] - 1/2*B[bbca] - 1/2*B[bcab] - 1/2*B[bcba]
- 1/2*B[cabb] - 1/2*B[cbab] - 1/2*B[cbba]

```

```

>>> from sage.all import *
>>> F = ShuffleAlgebra(QQ, 'abc')
>>> B = F.basis()
>>> B[Word('bb')] * B[Word('ca')]
B[bbca] + B[bcab] + B[bcba] + B[cabb]
+ B[cbab] + B[cbba]
>>> Integer(1) - B[Word('bb')] * B[Word('ca')] / Integer(2)
B[] - 1/2*B[bbca] - 1/2*B[bcab] - 1/2*B[bcba]
- 1/2*B[cabb] - 1/2*B[cbab] - 1/2*B[cbba]

```

algebra_generators()

Return the generators of this algebra.

EXAMPLES:

```

sage: A = ShuffleAlgebra(ZZ, 'fgh'); A
Shuffle Algebra on 3 generators ['f', 'g', 'h'] over Integer Ring
sage: A.algebra_generators()
Family (B[f], B[g], B[h])

sage: A = ShuffleAlgebra(QQ, ['x1', 'x2'])
sage: A.algebra_generators()
Family (B[x1], B[x2])

```

```

>>> from sage.all import *
>>> A = ShuffleAlgebra(ZZ, 'fgh'); A
Shuffle Algebra on 3 generators ['f', 'g', 'h'] over Integer Ring
>>> A.algebra_generators()
Family (B[f], B[g], B[h])

>>> A = ShuffleAlgebra(QQ, ['x1', 'x2'])
>>> A.algebra_generators()
Family (B[x1], B[x2])

```

antipode_on_basis(w)

Return the antipode on the basis element w.

EXAMPLES:

```

sage: A = ShuffleAlgebra(QQ, 'abc')
sage: W = A.basis().keys()
sage: A.antipode_on_basis(W("acb"))
-B[bca]

```

```

>>> from sage.all import *
>>> A = ShuffleAlgebra(QQ, 'abc')

```

(continues on next page)

(continued from previous page)

```
>>> W = A.basis().keys()
>>> A.antipode_on_basis(W("acb"))
-B[bca]
```

coproduct_on_basis(w)

Return the coproduct of the element of the basis indexed by the word w .

The coproduct is given by deconcatenation.

INPUT:

- w – a word

EXAMPLES:

```
sage: F = ShuffleAlgebra(QQ, 'ab')
sage: F.coproduct_on_basis(Word('a'))
B[] # B[a] + B[a] # B[]
sage: F.coproduct_on_basis(Word('aba'))
B[] # B[aba] + B[a] # B[ba]
+ B[ab] # B[a] + B[aba] # B[]
sage: F.coproduct_on_basis(Word())
B[] # B[]
```

```
>>> from sage.all import *
>>> F = ShuffleAlgebra(QQ, 'ab')
>>> F.coproduct_on_basis(Word('a'))
B[] # B[a] + B[a] # B[]
>>> F.coproduct_on_basis(Word('aba'))
B[] # B[aba] + B[a] # B[ba]
+ B[ab] # B[a] + B[aba] # B[]
>>> F.coproduct_on_basis(Word())
B[] # B[]
```

counit(S)

Return the counit of S .

EXAMPLES:

```
sage: F = ShuffleAlgebra(QQ, 'ab')
sage: S = F.an_element(); S
B[] + 2*B[a] + 3*B[b] + B[bab]
sage: F.counit(S)
1
```

```
>>> from sage.all import *
>>> F = ShuffleAlgebra(QQ, 'ab')
>>> S = F.an_element(); S
B[] + 2*B[a] + 3*B[b] + B[bab]
>>> F.counit(S)
1
```

degree_on_basis(w)

Return the degree of the element w .

EXAMPLES:

```
sage: A = ShuffleAlgebra(QQ, 'ab')
sage: [A.degree_on_basis(x.leading_support()) for x in A.some_elements() if x_
↪ != 0]
[0, 1, 1, 2]
```

```
>>> from sage.all import *
>>> A = ShuffleAlgebra(QQ, 'ab')
>>> [A.degree_on_basis(x.leading_support()) for x in A.some_elements() if x !
↪ = Integer(0)]
[0, 1, 1, 2]
```

dual_pbw_basis()

Return the dual PBW of self.

EXAMPLES:

```
sage: A = ShuffleAlgebra(QQ, 'ab')
sage: A.dual_pbw_basis()
The dual Poincare-Birkhoff-Witt basis of Shuffle Algebra on 2 generators ['a',
↪ 'b'] over Rational Field
```

```
>>> from sage.all import *
>>> A = ShuffleAlgebra(QQ, 'ab')
>>> A.dual_pbw_basis()
The dual Poincare-Birkhoff-Witt basis of Shuffle Algebra on 2 generators ['a',
↪ 'b'] over Rational Field
```

gen(i)

Return the i-th generator of the algebra.

INPUT:

- i – integer

EXAMPLES:

```
sage: F = ShuffleAlgebra(ZZ, 'xyz')
sage: F.gen(0)
B[x]

sage: F.gen(4)
Traceback (most recent call last):
...
IndexError: argument i (= 4) must be between 0 and 2
```

```
>>> from sage.all import *
>>> F = ShuffleAlgebra(ZZ, 'xyz')
>>> F.gen(Integer(0))
B[x]

>>> F.gen(Integer(4))
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
IndexError: argument i (= 4) must be between 0 and 2
```

gens()

Return the generators of this algebra.

EXAMPLES:

```
sage: A = ShuffleAlgebra(ZZ, 'fgh'); A
Shuffle Algebra on 3 generators ['f', 'g', 'h'] over Integer Ring
sage: A.algebra_generators()
Family (B[f], B[g], B[h])

sage: A = ShuffleAlgebra(QQ, ['x1', 'x2'])
sage: A.algebra_generators()
Family (B[x1], B[x2])
```

```
>>> from sage.all import *
>>> A = ShuffleAlgebra(ZZ, 'fgh'); A
Shuffle Algebra on 3 generators ['f', 'g', 'h'] over Integer Ring
>>> A.algebra_generators()
Family (B[f], B[g], B[h])

>>> A = ShuffleAlgebra(QQ, ['x1', 'x2'])
>>> A.algebra_generators()
Family (B[x1], B[x2])
```

one_basis()

Return the empty word, which index of 1 of this algebra, as per `AlgebrasWithBasis.ParentMethods.one_basis()`.

EXAMPLES:

```
sage: A = ShuffleAlgebra(QQ, 'a')
sage: A.one_basis()
word:
sage: A.one()
B[]
```

```
>>> from sage.all import *
>>> A = ShuffleAlgebra(QQ, 'a')
>>> A.one_basis()
word:
>>> A.one()
B[]
```

product_on_basis(w1, w2)

Return the product of basis elements `w1` and `w2`, as per `AlgebrasWithBasis.ParentMethods.product_on_basis()`.

INPUT:

- `w1, w2` – basis elements

EXAMPLES:

```

sage: A = ShuffleAlgebra(QQ, 'abc')
sage: W = A.basis().keys()
sage: A.product_on_basis(W("acb"), W("cba"))
B[acbacb] + B[acbcab] + 2*B[acbcba]
+ 2*B[accbab] + 4*B[accbba] + B[cabacb]
+ B[cabcab] + B[cabcba] + B[cacbab]
+ 2*B[cacbba] + 2*B[cbacab] + B[cbacab]
+ B[cbacba]

sage: (a,b,c) = A.algebra_generators()
sage: a * (1-b)^2 * c
2*B[abbc] - 2*B[abc] + 2*B[abcb] + B[ac]
- 2*B[acb] + 2*B[acbb] + 2*B[babc]
- 2*B[bac] + 2*B[bacb] + 2*B[bbac]
+ 2*B[bbca] - 2*B[bca] + 2*B[bcab]
+ 2*B[bcba] + B[ca] - 2*B[cab] + 2*B[cabb]
- 2*B[cba] + 2*B[cbab] + 2*B[cbba]

```

```

>>> from sage.all import *
>>> A = ShuffleAlgebra(QQ, 'abc')
>>> W = A.basis().keys()
>>> A.product_on_basis(W("acb"), W("cba"))
B[acbacb] + B[acbcab] + 2*B[acbcba]
+ 2*B[accbab] + 4*B[accbba] + B[cabacb]
+ B[cabcab] + B[cabcba] + B[cacbab]
+ 2*B[cacbba] + 2*B[cbacab] + B[cbacab]
+ B[cbacba]

>>> (a,b,c) = A.algebra_generators()
>>> a * (Integer(1)-b)**Integer(2) * c
2*B[abbc] - 2*B[abc] + 2*B[abcb] + B[ac]
- 2*B[acb] + 2*B[acbb] + 2*B[babc]
- 2*B[bac] + 2*B[bacb] + 2*B[bbac]
+ 2*B[bbca] - 2*B[bca] + 2*B[bcab]
+ 2*B[bcba] + B[ca] - 2*B[cab] + 2*B[cabb]
- 2*B[cba] + 2*B[cbab] + 2*B[cbba]

```

some_elements()

Return some typical elements.

EXAMPLES:

```

sage: F = ShuffleAlgebra(ZZ, 'xyz')
sage: F.some_elements()
[0, B[], B[x], B[y], B[z], B[xz] + B[zx]]

```

```

>>> from sage.all import *
>>> F = ShuffleAlgebra(ZZ, 'xyz')
>>> F.some_elements()
[0, B[], B[x], B[y], B[z], B[xz] + B[zx]]

```

to_dual_pbw_element(w)

Return the element w of `self` expressed in the dual PBW basis.

INPUT:

- w – an element of the shuffle algebra

EXAMPLES:

```
sage: A = ShuffleAlgebra(QQ, 'ab')
sage: f = 2 * A(Word()) + A(Word('ab')); f
2*B[] + B[ab]
sage: A.to_dual_pbw_element(f)
2*S[] + S[ab]
sage: A.to_dual_pbw_element(A.one())
S[]
sage: S = A.dual_pbw_basis()
sage: elt = S.expansion_on_basis(Word('abba')); elt
2*B[aabb] + B[abab] + B[abba]
sage: A.to_dual_pbw_element(elt)
S[abba]
sage: A.to_dual_pbw_element(2*A(Word('aabb')) + A(Word('abab')))
S[abab]
sage: S.expansion(S('abab'))
2*B[aabb] + B[abab]
```

```
>>> from sage.all import *
>>> A = ShuffleAlgebra(QQ, 'ab')
>>> f = Integer(2) * A(Word()) + A(Word('ab')); f
2*B[] + B[ab]
>>> A.to_dual_pbw_element(f)
2*S[] + S[ab]
>>> A.to_dual_pbw_element(A.one())
S[]
>>> S = A.dual_pbw_basis()
>>> elt = S.expansion_on_basis(Word('abba')); elt
2*B[aabb] + B[abab] + B[abba]
>>> A.to_dual_pbw_element(elt)
S[abba]
>>> A.to_dual_pbw_element(Integer(2)*A(Word('aabb')) + A(Word('abab')))
S[abab]
>>> S.expansion(S('abab'))
2*B[aabb] + B[abab]
```

variable_names()

Return the names of the variables.

EXAMPLES:

```
sage: R = ShuffleAlgebra(QQ, 'xy')
sage: R.variable_names()
{'x', 'y'}
```

```
>>> from sage.all import *
>>> R = ShuffleAlgebra(QQ, 'xy')
>>> R.variable_names()
{'x', 'y'}
```

9.8 Free Zinbiel Algebras

AUTHORS:

- Travis Scrimshaw (2015-09): initial version

class sage.algebras.free_zinbiel_algebra.**FreeZinbielAlgebra**($R, n, names, prefix, side$)

Bases: `CombinatorialFreeModule`

The free Zinbiel algebra on n generators.

Let R be a ring. A *Zinbiel algebra* is a non-associative algebra with multiplication $\circ$ that satisfies

$$(a \circ b) \circ c = a \circ (b \circ c) + a \circ (c \circ b).$$

Zinbiel algebras were first introduced by Loday (see [Lod1995] and [LV2012]) as the Koszul dual to Leibniz algebras (hence the name coined by Lemaire).

By default, the convention above is used. The opposite product, which satisfy the opposite axiom, can be used instead by setting the `side` parameter to `'>'` instead of the default value `'<'`.

Zinbiel algebras are divided power algebras, in that for

$$x^{\circ n} = (x \circ (x \circ \cdots \circ (x \circ x) \cdots))$$

we have

$$x^{\circ m} \circ x^{\circ n} = \binom{n+m-1}{m} x^{n+m}$$

and

$$\underbrace{((x \circ \cdots \circ x \circ (x \circ x) \cdots))}_{n+1 \text{ times}} = n! x^n.$$

Note

This implies that Zinbiel algebras are not power associative.

To every Zinbiel algebra, we can construct a corresponding commutative associative algebra by using the symmetrized product:

$$a * b = a \circ b + b \circ a.$$

The free Zinbiel algebra on n generators is isomorphic as R -modules to the reduced tensor algebra $\bar{T}(R^n)$ with the product

$$(x_0 x_1 \cdots x_p) \circ (x_{p+1} x_{p+2} \cdots x_{p+q}) = \sum_{\sigma \in S_{p,q}} x_0 (x_{\sigma(1)} x_{\sigma(2)} \cdots x_{\sigma(p+q)}),$$

where $S_{p,q}$ is the set of (p, q) -shuffles.

The free Zinbiel algebra is free as a divided power algebra. Moreover, the corresponding commutative algebra is isomorphic to the (non-unital) shuffle algebra.

INPUT:

- R – a ring

- n – (optional) the number of generators
- `names` – the generator names

Warning

Currently the basis is indexed by all finite words over the variables, including the empty word. This is a slight abuse as it is supposed to be indexed by all non-empty words.

EXAMPLES:

We create the free Zinbiel algebra and check the defining relation:

```
sage: Z.<x,y,z> = algebras.FreeZinbiel(QQ)
sage: (x*y)*z
Z[xyz] + Z[xzy]
sage: x*(y*z) + x*(z*y)
Z[xyz] + Z[xzy]
```

```
>>> from sage.all import *
>>> Z = algebras.FreeZinbiel(QQ, names=('x', 'y', 'z')); (x, y, z) = Z._first_
↳ngens(3)
>>> (x*y)*z
Z[xyz] + Z[xzy]
>>> x*(y*z) + x*(z*y)
Z[xyz] + Z[xzy]
```

We see that the Zinbiel algebra is not associative, not even power associative:

```
sage: x*(y*z)
Z[xyz]
sage: x*(x*x)
Z[xxx]
sage: (x*x)*x
2*Z[xxx]
```

```
>>> from sage.all import *
>>> x*(y*z)
Z[xyz]
>>> x*(x*x)
Z[xxx]
>>> (x*x)*x
2*Z[xxx]
```

We verify that it is a divided power algebra:

```
sage: (x*(x*x)) * (x*(x*(x*x)))
15*Z[xxxxxxxx]
sage: binomial(3+4-1, 4)
15
sage: (x*(x*(x*x))) * (x*(x*x))
20*Z[xxxxxxxx]
sage: binomial(3+4-1, 3)
```

(continues on next page)

(continued from previous page)

```

20
sage: ((x*x)*x)*x
6*Z[xxxx]
sage: (((x*x)*x)*x)*x
24*Z[xxxxx]

```

```

>>> from sage.all import *
>>> (x*(x*x)) * (x*(x*(x*x)))
15*Z[xxxxxxxx]
>>> binomial(Integer(3)+Integer(4)-Integer(1), Integer(4))
15
>>> (x*(x*(x*x))) * (x*(x*x))
20*Z[xxxxxxxx]
>>> binomial(Integer(3)+Integer(4)-Integer(1), Integer(3))
20
>>> ((x*x)*x)*x
6*Z[xxxx]
>>> (((x*x)*x)*x)*x
24*Z[xxxxx]

```

A few tests with the opposite convention for the product:

```

sage: Z.<x,y,z> = algebras.FreeZinbiel(QQ, side='>')
sage: (x*y)*z
Z[xyz]
sage: x*(y*z)
Z[xyz] + Z[yxz]

```

```

>>> from sage.all import *
>>> Z = algebras.FreeZinbiel(QQ, side='>', names=('x', 'y', 'z')); (x, y, z) = Z._first_ngens(3)
>>> (x*y)*z
Z[xyz]
>>> x*(y*z)
Z[xyz] + Z[yxz]

```

REFERENCES:

- [Wikipedia article Zinbiel_algebra](#)
- [Lod1995]
- [LV2012]

`algebra_generators()`

Return the algebra generators of `self`.

EXAMPLES:

```

sage: Z.<x,y,z> = algebras.FreeZinbiel(QQ)
sage: list(Z.algebra_generators())
[Z[x], Z[y], Z[z]]

```

```
>>> from sage.all import *
>>> Z = algebras.FreeZinbiel(QQ, names=('x', 'y', 'z',)); (x, y, z,) = Z._
↳first_ngens(3)
>>> list(Z.algebra_generators())
[Z[x], Z[y], Z[z]]
```

change_ring(*R*)

Return the free Zinbiel algebra in the same variables over *R*.

INPUT:

- *R* – a ring

The same side convention is used for the product.

EXAMPLES:

```
sage: A = algebras.FreeZinbiel(ZZ, 'f,g,h')
sage: A.change_ring(QQ)
Free Zinbiel algebra on generators (Z[f], Z[g], Z[h])
over Rational Field
```

```
>>> from sage.all import *
>>> A = algebras.FreeZinbiel(ZZ, 'f,g,h')
>>> A.change_ring(QQ)
Free Zinbiel algebra on generators (Z[f], Z[g], Z[h])
over Rational Field
```

construction()

Return a pair (*F*, *R*), where *F* is a *ZinbielFunctor* and *R* is a ring, such that *F*(*R*) returns self.

EXAMPLES:

```
sage: P = algebras.FreeZinbiel(ZZ, 'x,y')
sage: x,y = P.gens()
sage: F, R = P.construction()
sage: F
Zinbiel[x,y]
sage: R
Integer Ring
sage: F(ZZ) is P
True
sage: F(QQ)
Free Zinbiel algebra on generators (Z[x], Z[y]) over Rational Field
```

```
>>> from sage.all import *
>>> P = algebras.FreeZinbiel(ZZ, 'x,y')
>>> x,y = P.gens()
>>> F, R = P.construction()
>>> F
Zinbiel[x,y]
>>> R
Integer Ring
>>> F(ZZ) is P
```

(continues on next page)

(continued from previous page)

```
True
>>> F(QQ)
Free Zinbiel algebra on generators (Z[x], Z[y]) over Rational Field
```

coproduct_on_basis(*w*)

Return the coproduct of the element of the basis indexed by the word *w*.

The coproduct is given by deconcatenation.

INPUT:

- *w* – a word

EXAMPLES:

```
sage: F = algebras.FreeZinbiel(QQ, ['a', 'b'])
sage: F.coproduct_on_basis(Word('a'))
Z[] # Z[a] + Z[a] # Z[]
sage: F.coproduct_on_basis(Word('aba'))
Z[] # Z[aba] + Z[a] # Z[ba] + Z[ab] # Z[a] + Z[aba] # Z[]
sage: F.coproduct_on_basis(Word())
Z[] # Z[]
```

```
>>> from sage.all import *
>>> F = algebras.FreeZinbiel(QQ, ['a', 'b'])
>>> F.coproduct_on_basis(Word('a'))
Z[] # Z[a] + Z[a] # Z[]
>>> F.coproduct_on_basis(Word('aba'))
Z[] # Z[aba] + Z[a] # Z[ba] + Z[ab] # Z[a] + Z[aba] # Z[]
>>> F.coproduct_on_basis(Word())
Z[] # Z[]
```

counit(*S*)

Return the counit of *S*.

EXAMPLES:

```
sage: F = algebras.FreeZinbiel(QQ, ['a', 'b'])
sage: S = F.an_element(); S
Z[] + 2*Z[a] + 3*Z[b] + Z[bab]
sage: F.counit(S)
1
```

```
>>> from sage.all import *
>>> F = algebras.FreeZinbiel(QQ, ['a', 'b'])
>>> S = F.an_element(); S
Z[] + 2*Z[a] + 3*Z[b] + Z[bab]
>>> F.counit(S)
1
```

degree_on_basis(*t*)

Return the degree of a word in the free Zinbiel algebra.

This is the length.

EXAMPLES:

```
sage: A = algebras.FreeZinbiel(QQ, 'x,y')
sage: W = A.basis().keys()
sage: A.degree_on_basis(W('xy'))
2
```

```
>>> from sage.all import *
>>> A = algebras.FreeZinbiel(QQ, 'x,y')
>>> W = A.basis().keys()
>>> A.degree_on_basis(W('xy'))
2
```

gens()

Return the generators of self.

EXAMPLES:

```
sage: Z.<x,y,z> = algebras.FreeZinbiel(QQ)
sage: Z.gens()
(Z[x], Z[y], Z[z])
```

```
>>> from sage.all import *
>>> Z = algebras.FreeZinbiel(QQ, names=('x', 'y', 'z')); (x, y, z) = Z._
↳first_ngens(3)
>>> Z.gens()
(Z[x], Z[y], Z[z])
```

product_on_basis_left(x, y)

Return the product $<$ of the basis elements indexed by x and y .

This is one half of the shuffle product, where the first letter comes from the first letter of the first argument.

INPUT:

- x, y – two words

EXAMPLES:

```
sage: Z.<x,y,z> = algebras.FreeZinbiel(QQ)
sage: (x*y)*z # indirect doctest
Z[xyz] + Z[xzy]
```

```
>>> from sage.all import *
>>> Z = algebras.FreeZinbiel(QQ, names=('x', 'y', 'z')); (x, y, z) = Z._
↳first_ngens(3)
>>> (x*y)*z # indirect doctest
Z[xyz] + Z[xzy]
```

product_on_basis_right(x, y)

Return the product $>$ of the basis elements indexed by x and y .

This is one half of the shuffle product, where the last letter comes from the last letter of the second argument.

INPUT:

- x, y – two words

EXAMPLES:

```
sage: Z.<x,y,z> = algebras.FreeZinbiel(QQ, side='>')
sage: (x*y)*z # indirect doctest
Z[xyz]
```

```
>>> from sage.all import *
>>> Z = algebras.FreeZinbiel(QQ, side='>', names=('x', 'y', 'z')); (x, y, z,
↪) = Z._first_ngens(3)
>>> (x*y)*z # indirect doctest
Z[xyz]
```

side()

Return the choice of side for the product.

This is either '<' or '>'.

EXAMPLES:

```
sage: Z.<x,y,z> = algebras.FreeZinbiel(QQ)
sage: Z.side()
'<'
```

```
>>> from sage.all import *
>>> Z = algebras.FreeZinbiel(QQ, names=('x', 'y', 'z')); (x, y, z,) = Z._
↪first_ngens(3)
>>> Z.side()
'<'
```

class sage.algebras.free_zinbiel_algebra.**ZinbielFunctor**(*variables, side*)

Bases: [ConstructionFunctor](#)

A constructor for free Zinbiel algebras.

EXAMPLES:

```
sage: P = algebras.FreeZinbiel(ZZ, 'x,y')
sage: x,y = P.gens()
sage: F = P.construction()[0]; F
Zinbiel[x,y]

sage: A = GF(5)['a,b']
sage: a, b = A.gens()
sage: F(A)
Free Zinbiel algebra on generators (Z[x], Z[y])
over Multivariate Polynomial Ring in a, b over Finite Field of size 5

sage: f = A.hom([a+b,a-b],A)
sage: F(f)
Generic endomorphism of Free Zinbiel algebra on generators (Z[x], Z[y])
over Multivariate Polynomial Ring in a, b over Finite Field of size 5

sage: F(f)(a * F(A)(x))
(a+b)*Z[x]
```

```

>>> from sage.all import *
>>> P = algebras.FreeZinbiel(ZZ, 'x,y')
>>> x,y = P.gens()
>>> F = P.construction()[Integer(0)]; F
Zinbiel[x,y]

>>> A = GF(Integer(5))['a,b']
>>> a, b = A.gens()
>>> F(A)
Free Zinbiel algebra on generators (Z[x], Z[y])
over Multivariate Polynomial Ring in a, b over Finite Field of size 5

>>> f = A.hom([a+b,a-b],A)
>>> F(f)
Generic endomorphism of Free Zinbiel algebra on generators (Z[x], Z[y])
over Multivariate Polynomial Ring in a, b over Finite Field of size 5

>>> F(f)(a * F(A)(x))
(a+b)*Z[x]

```

merge(*other*)

Merge self with another construction functor, or return None.

EXAMPLES:

```

sage: functor = sage.algebras.free_zinbiel_algebra.ZinbielFunctor
sage: F = functor(['x','y'], '<')
sage: G = functor(['t'], '<')
sage: F.merge(G)
Zinbiel[x,y,t]
sage: F.merge(F)
Zinbiel[x,y]

```

```

>>> from sage.all import *
>>> functor = sage.algebras.free_zinbiel_algebra.ZinbielFunctor
>>> F = functor(['x','y'], '<')
>>> G = functor(['t'], '<')
>>> F.merge(G)
Zinbiel[x,y,t]
>>> F.merge(F)
Zinbiel[x,y]

```

With an infinite generating set:

```

sage: H = functor(ZZ, '<')
sage: H.merge(H) is H
True
sage: H.merge(F) is None
True
sage: F.merge(H) is None
True

```

```

>>> from sage.all import *
>>> H = functor(ZZ, '<')
>>> H.merge(H) is H
True
>>> H.merge(F) is None
True
>>> F.merge(H) is None
True

```

Now some actual use cases:

```

sage: R = algebras.FreeZinbiel(ZZ, 'x,y,z')
sage: x,y,z = R.gens()
sage: 1/2 * x
1/2*Z[x]
sage: parent(1/2 * x)
Free Zinbiel algebra on generators (Z[x], Z[y], Z[z])
over Rational Field

sage: S = algebras.FreeZinbiel(QQ, 'z,t')
sage: z,t = S.gens()
sage: x * t
Z[xt]
sage: parent(x * t)
Free Zinbiel algebra on generators (Z[z], Z[t], Z[x], Z[y])
over Rational Field

```

```

>>> from sage.all import *
>>> R = algebras.FreeZinbiel(ZZ, 'x,y,z')
>>> x,y,z = R.gens()
>>> Integer(1)/Integer(2) * x
1/2*Z[x]
>>> parent(Integer(1)/Integer(2) * x)
Free Zinbiel algebra on generators (Z[x], Z[y], Z[z])
over Rational Field

>>> S = algebras.FreeZinbiel(QQ, 'z,t')
>>> z,t = S.gens()
>>> x * t
Z[xt]
>>> parent(x * t)
Free Zinbiel algebra on generators (Z[z], Z[t], Z[x], Z[y])
over Rational Field

```

rank = 9

INDICES AND TABLES

- [Index](#)
- [Module Index](#)
- [Search Page](#)

BIBLIOGRAPHY

- [Naz96] Maxim Nazarov, Young's Orthogonal Form for Brauer's Centralizer Algebra. *Journal of Algebra* 182 (1996), 664–693.
- [GL1996] J.J. Graham and G.I. Lehrer, Cellular algebras. *Inventiones mathematicae* 123 (1996), 1–34.
- [Solomon67] Louis Solomon. *The Burnside Algebra of a Finite Group*. *Journal of Combinatorial Theory*, **2**, 1967.
[doi:10.1016/S0021-9800\(67\)80064-4](https://doi.org/10.1016/S0021-9800(67)80064-4).
- [Greene73] Curtis Greene. *On the Möbius algebra of a partially ordered set*. *Advances in Mathematics*, **10**, 1973.
[doi:10.1016/0001-8708\(73\)90106-0](https://doi.org/10.1016/0001-8708(73)90106-0).
- [Etienne98] Gwihen Etienne. *On the Möbius algebra of geometric lattices*. *European Journal of Combinatorics*, **19**, 1998.
[doi:10.1006/eujc.1998.0227](https://doi.org/10.1006/eujc.1998.0227).

PYTHON MODULE INDEX

a

sage.algebras.affine_nil_temperley_lieb, 151

sage.algebras.askey_wilson, 155

sage.algebras.associated_graded, 1007

sage.algebras.catalog, 1

sage.algebras.cellular_basis, 1011

sage.algebras.clifford_algebra, 257

sage.algebras.cluster_algebra, 295

sage.algebras.commutative_dga, 939

sage.algebras.down_up_algebra, 353

sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra, 121

sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element, 138

sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_ideal, 145

sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism, 146

sage.algebras.finite_gca, 933

sage.algebras.free_algebra, 59

sage.algebras.free_algebra_element, 80

sage.algebras.free_algebra_quotient, 106

sage.algebras.free_algebra_quotient_element, 112

sage.algebras.free_zinbiel_algebra, 1385

sage.algebras.fusion_rings.f_matrix, 391

sage.algebras.fusion_rings.fast_parallel_fmats_methods, 433

sage.algebras.fusion_rings.fast_parallel_fusion_ring_braid_repn, 434

sage.algebras.fusion_rings.fusion_double, 419

sage.algebras.fusion_rings.fusion_ring, 365

sage.algebras.fusion_rings.poly_tup_engine, 434

sage.algebras.fusion_rings.shm_managers, 440

sage.algebras.group_algebra, 469

sage.algebras.hall_algebra, 447

sage.algebras.hecke_algebras.ariki_koike_algebra, 803

sage.algebras.hecke_algebras.ariki_koike_specht_modules, 816

sage.algebras.hecke_algebras.cubic_hecke_algebra, 862

sage.algebras.hecke_algebras.cubic_hecke_base_ring, 895

sage.algebras.hecke_algebras.cubic_hecke_matrix_rep, 917

sage.algebras.iwahori_hecke_algebra, 821

sage.algebras.jordan_algebra, 1323

sage.algebras.letterplace.free_algebra_element_letterplace, 91

sage.algebras.letterplace.free_algebra_letterplace, 82

sage.algebras.letterplace.letterplace_ideal, 98

sage.algebras.lie_algebras.abelian, 1055

sage.algebras.lie_algebras.affine_lie_algebra, 1058

sage.algebras.lie_algebras.bch, 1072

sage.algebras.lie_algebras.bgg_dual_module, 1077

sage.algebras.lie_algebras.bgg_resolution, 1074

sage.algebras.lie_algebras.center_uea, 1092

sage.algebras.lie_algebras.classical_lie_algebra, 1100

sage.algebras.lie_algebras.examples, 1121

sage.algebras.lie_algebras.free_lie_algebra, 1133

sage.algebras.lie_algebras.heisenberg, 1141

sage.algebras.lie_algebras.lie_algebra, 1151

sage.algebras.lie_algebras.lie_algebra_element, 1173

sage.algebras.lie_algebras.morphism, 1183

sage.algebras.lie_algebras.nilpotent_lie_algebra, 1191

`sage.algebras.lie_algebras.onsager`, 1195
`sage.algebras.lie_algebras.poincare_birkhoff_witt`, 1211
`sage.algebras.lie_algebras.quotient`, 1221
`sage.algebras.lie_algebras.rank_two_heisenberg_virasoro`, 1228
`sage.algebras.lie_algebras.representation`, 1231
`sage.algebras.lie_algebras.structure_coefficients`, 1240
`sage.algebras.lie_algebras.subalgebra`, 1244
`sage.algebras.lie_algebras.symplectic_derivation`, 1258
`sage.algebras.lie_algebras.verma_module`, 1261
`sage.algebras.lie_algebras.virasoro`, 1279
`sage.algebras.lie_conformal_algebras.abelian_lie_conformal_algebra`, 1299
`sage.algebras.lie_conformal_algebras.affine_lie_conformal_algebra`, 1300
`sage.algebras.lie_conformal_algebras.bosonic_ghosts_lie_conformal_algebra`, 1302
`sage.algebras.lie_conformal_algebras.examples`, 1297
`sage.algebras.lie_conformal_algebras.fermionic_ghosts_lie_conformal_algebra`, 1303
`sage.algebras.lie_conformal_algebras.finitely_freely_generated_lca`, 1315
`sage.algebras.lie_conformal_algebras.free_bosons_lie_conformal_algebra`, 1305
`sage.algebras.lie_conformal_algebras.free_fermions_lie_conformal_algebra`, 1307
`sage.algebras.lie_conformal_algebras.freely_generated_lie_conformal_algebra`, 1316
`sage.algebras.lie_conformal_algebras.graded_lie_conformal_algebra`, 1318
`sage.algebras.lie_conformal_algebras.lie_conformal_algebra`, 1292
`sage.algebras.lie_conformal_algebras.lie_conformal_algebra_element`, 1298
`sage.algebras.lie_conformal_algebras.lie_conformal_algebra_with_basis`, 1320

`sage.algebras.lie_conformal_algebras.lie_conformal_algebra_with_structure_coefs`, 1321
`sage.algebras.lie_conformal_algebras.n2_lie_conformal_algebra`, 1309
`sage.algebras.lie_conformal_algebras.neveu_schwarz_lie_conformal_algebra`, 1310
`sage.algebras.lie_conformal_algebras.virasoro_lie_conformal_algebra`, 1311
`sage.algebras.lie_conformal_algebras.weyl_lie_conformal_algebra`, 1312
`sage.algebras.nil_coxeter_algebra`, 855
`sage.algebras.octonion_algebra`, 1043
`sage.algebras.orlik_solomon`, 496
`sage.algebras.orlik_terao`, 487
`sage.algebras.q_commuting_polynomials`, 1023
`sage.algebras.q_system`, 1017
`sage.algebras.quantum_clifford`, 526
`sage.algebras.quantum_groups.ace_quantum_onsager`, 3
`sage.algebras.quantum_groups.fock_space`, 13
`sage.algebras.quantum_groups.q_numbers`, 44
`sage.algebras.quantum_groups.quantum_group_gap`, 539
`sage.algebras.quantum_groups.representations`, 47
`sage.algebras.quantum_matrix_coordinate_algebra`, 570
`sage.algebras.quantum_oscillator`, 581
`sage.algebras.quatalg.quaternion_algebra`, 589
`sage.algebras.rational_cherednik_algebra`, 658
`sage.algebras.schur_algebra`, 662
`sage.algebras.shuffle_algebra`, 1370
`sage.algebras.splitting_algebra`, 1034
`sage.algebras.steenrod.steenrod_algebra`, 669
`sage.algebras.steenrod.steenrod_algebra_bases`, 729
`sage.algebras.steenrod.steenrod_algebra_misc`, 745
`sage.algebras.steenrod.steenrod_algebra_mult`, 762
`sage.algebras.tensor_algebra`, 114
`sage.algebras.weyl_algebra`, 773
`sage.algebras.yangian`, 787
`sage.algebras.yokonuma_hecke_algebra`, 857

C

`sage.combinat.descent_algebra`, 338
`sage.combinat.diagram_algebras`, 173

`sage.combinat.free_dendriform_algebra`, [1341](#)
`sage.combinat.free_prelie_algebra`, [1354](#)
`sage.combinat.grossman_larson_algebras`, [471](#)
`sage.combinat.partition_algebra`, [509](#)
`sage.combinat.posets.incidence_algebras`,
 [460](#)
`sage.combinat.posets.moebius_algebra`, [480](#)

A

- `a()` (*sage.combinat.diagram_algebras.PartitionAlgebra* method), 210
- `a_realization()` (*sage.algebras.hecke_algebras.ariki_koike_algebra.ArikiKoikeAlgebra* method), 815
- `a_realization()` (*sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra* method), 847
- `a_realization()` (*sage.algebras.lie_algebras.free_lie_algebra.FreeLieAlgebra* method), 1137
- `a_realization()` (*sage.algebras.quantum_groups.fock_space.FockSpace* method), 32
- `a_realization()` (*sage.combinat.descent_algebra.DescentAlgebra* method), 350
- `a_realization()` (*sage.combinat.posets.moebius_algebra.MoebiusAlgebra* method), 482
- `a_realization()` (*sage.combinat.posets.moebius_algebra.QuantumMoebiusAlgebra* method), 486
- `AA()` (in module *sage.algebras.steenrod.steenrod_algebra*), 678
- `abelian()` (in module *sage.algebras.lie_algebras.examples*), 1122
- `AbelianLieAlgebra` (class in *sage.algebras.lie_algebras.abelian*), 1055
- `AbelianLieAlgebra.Element` (class in *sage.algebras.lie_algebras.abelian*), 1056
- `AbelianLieConformalAlgebra` (class in *sage.algebras.lie_conformal_algebras.abelian_lie_conformal_algebra*), 1299
- `abs()` (*sage.algebras.octonion_algebra.Octonion_generic* method), 1049
- `AbsIrreducibleRep` (class in *sage.algebras.hecke_algebras.cubic_hecke_matrix_rep*), 917
- `AbstractPartitionDiagram` (class in *sage.combinat.diagram_algebras*), 173
- `AbstractPartitionDiagrams` (class in *sage.combinat.diagram_algebras*), 178
- `ACEQuantumOnsagerAlgebra` (class in *sage.algebras.quantum_groups.ace_quantum_onsager*), 3
- `additive_order()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element* method), 690
- `adem()` (in module *sage.algebras.steenrod.steenrod_algebra_mult*), 764
- `adjoint_matrix()` (*sage.algebras.lie_algebras.subalgebra.LieSubalgebra_finite_dimensional_with_basis.Element* method), 1248
- `AdjointRepresentation` (class in *sage.algebras.quantum_groups.representations*), 47
- `affine()` (*sage.algebras.lie_algebras.classical_lie_algebra.ClassicalMatrixLieAlgebra* method), 1101
- `affine()` (*sage.algebras.lie_algebras.classical_lie_algebra.LieAlgebraChevalleyBasis* method), 1106
- `affine_transformations_line()` (in module *sage.algebras.lie_algebras.examples*), 1122
- `AffineLieAlgebra` (class in *sage.algebras.lie_algebras.affine_lie_algebra*), 1058
- `AffineLieConformalAlgebra` (class in *sage.algebras.lie_conformal_algebras.affine_lie_conformal_algebra*), 1300
- `AffineNilTemperleyLiebTypeA` (class in *sage.algebras.affine_nil_temperley_lieb*), 151
- `algebra_generator()` (*sage.algebras.affine_nil_temperley_lieb.AffineNilTemperleyLiebTypeA* method), 151
- `algebra_generators()` (*sage.algebras.affine_nil_temperley_lieb.AffineNilTemperleyLiebTypeA* method), 152
- `algebra_generators()` (*sage.algebras.askey_wilson.AskeyWilsonAlgebra* method), 160
- `algebra_generators()` (*sage.algebras.associated_graded.AssociatedGradedAlgebra* method), 1009
- `algebra_generators()` (*sage.algebras.clifford_algebra.CliffordAlgebra* method), 259
- `algebra_generators()` (*sage.algebras.down_up_algebra.DownUpAlgebra* method), 356
- `algebra_generators()` (*sage.algebras.finite_gca.FiniteGCAAlgebra* method), 935
- `algebra_generators()` (*sage.algebras.free_algebra.FreeAlgebra_generic* method), 67

`algebra_generators()` (*sage.algebras.free_algebra.PBWBasisOfFreeAlgebra* method), 77
`algebra_generators()` (*sage.algebras.free_zinbiel_algebra.FreeZinbielAlgebra* method), 1387
`algebra_generators()` (*sage.algebras.hecke_algebras.ariki_koike_algebra.ArikiKoikeAlgebra.LT* method), 808
`algebra_generators()` (*sage.algebras.hecke_algebras.ariki_koike_algebra.ArikiKoikeAlgebra.T* method), 813
`algebra_generators()` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* method), 870
`algebra_generators()` (*sage.algebras.jordan_algebra.ExceptionalJordanAlgebra* method), 1325
`algebra_generators()` (*sage.algebras.jordan_algebra.JordanAlgebraSymmetricBilinear* method), 1337
`algebra_generators()` (*sage.algebras.jordan_algebra.SpecialJordanAlgebra* method), 1339
`algebra_generators()` (*sage.algebras.lie_algebras.center_uea.CenterUEA* method), 1096
`algebra_generators()` (*sage.algebras.lie_algebras.onsager.QuantumOnsagerAlgebra* method), 1206
`algebra_generators()` (*sage.algebras.lie_algebras.poincare_birkhoff_witt.PoincareBirkhoffWittBasis* method), 1213
`algebra_generators()` (*sage.algebras.orlik_solomon.OrlikSolomonAlgebra* method), 497
`algebra_generators()` (*sage.algebras.orlik_terao.OrlikTeraoAlgebra* method), 489
`algebra_generators()` (*sage.algebras.q_commuting_polynomials.qCommutingPolynomials_generic* method), 1032
`algebra_generators()` (*sage.algebras.q_system.QSystem* method), 1020
`algebra_generators()` (*sage.algebras.quantum_clifford.QuantumCliffordAlgebra* method), 530
`algebra_generators()` (*sage.algebras.quantum_groups.ace_quantum_onsager.ACEQuantumOnsagerAlgebra* method), 5
`algebra_generators()` (*sage.algebras.quantum_groups.quantum_group_gap.LowerHalfQuantumGroup* method), 547
`algebra_generators()` (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroup* method), 559
`algebra_generators()` (*sage.algebras.quantum_matrix_coordinate_algebra.QuantumGL* method), 572
`algebra_generators()` (*sage.algebras.quantum_matrix_coordinate_algebra.QuantumMatrixCoordinateAlgebra* method), 575
`algebra_generators()` (*sage.algebras.quantum_oscillator.QuantumOscillatorAlgebra* method), 585
`algebra_generators()` (*sage.algebras.rational_cherednik_algebra.RationalCherednikAlgebra* method), 659
`algebra_generators()` (*sage.algebras.shuffle_algebra.DualPBWBasis* method), 1372
`algebra_generators()` (*sage.algebras.shuffle_algebra.ShuffleAlgebra* method), 1379
`algebra_generators()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic* method), 704
`algebra_generators()` (*sage.algebras.tensor_algebra.TensorAlgebra* method), 115
`algebra_generators()` (*sage.algebras.weyl_algebra.DifferentialWeylAlgebra* method), 774
`algebra_generators()` (*sage.algebras.yangian.Yangian* method), 793
`algebra_generators()` (*sage.algebras.yokonuma_hecke_algebra.YokonumaHeckeAlgebra* method), 859
`algebra_generators()` (*sage.combinat.free_dendriform_algebra.FreeDendriformAlgebra* method), 1345
`algebra_generators()` (*sage.combinat.free_prelie_algebra.FreePreLieAlgebra* method), 1358
`AlgebraMorphism` (class in *sage.algebras.askey_wilson*), 155
`alternating_central_extension()` (*sage.algebras.lie_algebras.onsager.OnsagerAlgebra* method), 1197
`alternative_name()` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* method), 921
`ambient()` (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 308
`ambient()` (*sage.algebras.lie_algebras.affine_lie_algebra.TwistedAffineLieAlgebra* method), 1067
`ambient()` (*sage.algebras.lie_algebras.bgg_dual_module.SimpleModule* method), 1082
`ambient()` (*sage.algebras.lie_algebras.center_uea.CenterUEA* method), 1097
`ambient()` (*sage.algebras.lie_algebras.quotient.LieQuotient_finite_dimensional_with_basis* method), 1225
`ambient()` (*sage.algebras.lie_algebras.subalgebra.LieSubalgebra_finite_dimensional_with_basis* method), 1249
`ambient()` (*sage.algebras.quantum_groups.quantum_group_gap.HighestWeightSubmodule* method), 541
`ambient()` (*sage.algebras.quantum_groups.quantum_group_gap.LowerHalfQuantumGroup* method), 547

- `ambient()` (*sage.combinat.diagram_algebras.SubPartitionAlgebra* method), 238
`an_element()` (*sage.algebras.askey_wilson.AskeyWilsonAlgebra* method), 160
`an_element()` (*sage.algebras.quantum_groups.quantum_group_gap.HighestWeightModule* method), 539
`an_element()` (*sage.algebras.quantum_groups.quantum_group_gap.HighestWeightSubmodule* method), 541
`an_element()` (*sage.algebras.quantum_groups.quantum_group_gap.LowerHalfQuantumGroup* method), 547
`an_element()` (*sage.algebras.rational_cherednik_algebra.RationalCherednikAlgebra* method), 659
`an_element()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic* method), 705
`an_element()` (*sage.combinat.free_dendriform_algebra.FreeDendriformAlgebra* method), 1346
`an_element()` (*sage.combinat.free_prelie_algebra.FreePreLieAlgebra* method), 1359
`an_element()` (*sage.combinat.grossman_larson_algebras.GrossmanLarsonAlgebra* method), 473
`antipode()` (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroup* method), 559
`antipode()` (*sage.algebras.shuffle_algebra.DualPBW BASIS* method), 1372
`antipode_on_basis()` (*sage.algebras.clifford_algebra.ExteriorAlgebra* method), 274
`antipode_on_basis()` (*sage.algebras.hall_algebra.HallAlgebra* method), 451
`antipode_on_basis()` (*sage.algebras.hall_algebra.HallAlgebraMonomials* method), 457
`antipode_on_basis()` (*sage.algebras.quantum_matrix_coordinate_algebra.QuantumGL* method), 572
`antipode_on_basis()` (*sage.algebras.shuffle_algebra.ShuffleAlgebra* method), 1379
`antipode_on_basis()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic* method), 706
`antipode_on_basis()` (*sage.algebras.tensor_algebra.TensorAlgebra* method), 115
`antipode_on_basis()` (*sage.algebras.yangian.GradientYangianLoop* method), 787
`antipode_on_basis()` (*sage.combinat.grossman_larson_algebras.GrossmanLarsonAlgebra* method), 474
`aomoto_complex()` (*sage.algebras.orlik_solomon.OrlikSolomonAlgebra* method), 498
`apply_coeff_map()` (in module *sage.algebras.fusion_rings.poly_tup_engine*), 434
`approximation` (*sage.algebras.quantum_groups.fock_space.FockSpace* attribute), 32
`approximation` (*sage.algebras.quantum_groups.fock_space.FockSpaceTruncated* attribute), 44
`ariki_koike_algebra()` (*sage.algebras.hecke_algebras.ariki_koike_specht_modules.SpechtModule* method), 821
`ArikiKoikeAlgebra` (class in *sage.algebras.hecke_algebras.ariki_koike_algebra*), 803
`ArikiKoikeAlgebra.LT` (class in *sage.algebras.hecke_algebras.ariki_koike_algebra*), 807
`ArikiKoikeAlgebra.LT.Element` (class in *sage.algebras.hecke_algebras.ariki_koike_algebra*), 807
`ArikiKoikeAlgebra.T` (class in *sage.algebras.hecke_algebras.ariki_koike_algebra*), 811
`arnonA_long_mono_to_string()` (in module *sage.algebras.steenrod.steenrod_algebra_misc*), 745
`arnonA_mono_to_string()` (in module *sage.algebras.steenrod.steenrod_algebra_misc*), 746
`arnonC_basis()` (in module *sage.algebras.steenrod.steenrod_algebra_bases*), 730
`as_cdga()` (*sage.algebras.orlik_solomon.OrlikSolomonAlgebra* method), 499
`as_gca()` (*sage.algebras.orlik_solomon.OrlikSolomonAlgebra* method), 500
`as_splitting_algebra()` (*sage.algebras.hecke_algebras.cubic_hecke_base_ring.CubicHeckeExtensionRing* method), 896
`AskeyWilsonAlgebra` (class in *sage.algebras.askey_wilson*), 155
`AssociatedGradedAlgebra` (class in *sage.algebras.associated_graded*), 1007
`associative_algebra()` (*sage.algebras.lie_algebras.lie_algebra.LieAlgebraFromAssociative* method), 1167
`AssociativeFunctor` (class in *sage.algebras.free_algebra*), 61
`asymmetry_function()` (*sage.algebras.lie_algebras.classical_lie_algebra.LieAlgebraChevalleyBasis_simply_laced* method), 1110
`atomic_basis()` (in module *sage.algebras.steenrod.steenrod_algebra_bases*), 731
`atomic_basis_odd()` (in module *sage.algebras.steenrod.steenrod_algebra_bases*), 733
`attempt_number_field_computation()` (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 395
- ## B
- `b_matrix()` (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 308
`b_matrix()` (*sage.algebras.cluster_algebra.ClusterAlgebraSeed* method), 330

- `bar()` (*sage.algebras.jordan_algebra.JordanAlgebraSymmetricBilinear.Element* method), 1335
- `bar()` (*sage.algebras.quantum_groups.quantum_group_gap.LowerHalfQuantumGroup.Element* method), 544
- `bar()` (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroup.Element* method), 554
- `bar()` (*sage.algebras.quantum_matrix_coordinate_algebra.QuantumMatrixCoordinateAlgebra_abstract.Element* method), 577
- `bar_on_basis()` (*sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T* method), 840
- `base_diagram()` (*sage.combinat.diagram_algebras.AbstractPartitionDiagram* method), 174
- `base_extend()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra* method), 122
- `base_map()` (*sage.algebras.lie_algebras.morphism.LieAlgebraHomomorphism_im_gens* method), 1185
- `base_module()` (*sage.algebras.tensor_algebra.TensorAlgebra* method), 116
- `base_ring()` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* method), 870
- `BaseRingLift` (class in *sage.algebras.tensor_algebra*), 114
- `basis()` (*sage.algebras.commutative_dga.GCAlgebra* method), 985
- `basis()` (*sage.algebras.commutative_dga.GCAlgebra_multigraded* method), 994
- `basis()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra* method), 122
- `basis()` (*sage.algebras.jordan_algebra.ExceptionalJordanAlgebra* method), 1326
- `basis()` (*sage.algebras.jordan_algebra.JordanAlgebraSymmetricBilinear* method), 1337
- `basis()` (*sage.algebras.jordan_algebra.SpecialJordanAlgebra* method), 1339
- `basis()` (*sage.algebras.lie_algebras.affine_lie_algebra.AffineLieAlgebra* method), 1058
- `basis()` (*sage.algebras.lie_algebras.classical_lie_algebra.ClassicalMatrixLieAlgebra* method), 1102
- `basis()` (*sage.algebras.lie_algebras.classical_lie_algebra.e8* method), 1115
- `basis()` (*sage.algebras.lie_algebras.classical_lie_algebra.gl* method), 1116
- `basis()` (*sage.algebras.lie_algebras.classical_lie_algebra.MatrixCompactRealForm* method), 1112
- `basis()` (*sage.algebras.lie_algebras.free_lie_algebra.FreeLieBasis_abstract* method), 1139
- `basis()` (*sage.algebras.lie_algebras.heisenberg.HeisenbergAlgebra_fd* method), 1144
- `basis()` (*sage.algebras.lie_algebras.heisenberg.InfiniteHeisenbergAlgebra* method), 1150
- `basis()` (*sage.algebras.lie_algebras.onsager.OnsagerAlgebra* method), 1197
- `basis()` (*sage.algebras.lie_algebras.onsager.OnsagerAlgebraACE* method), 1202
- `basis()` (*sage.algebras.lie_algebras.subalgebra.LieSubalgebra_finite_dimensional_with_basis* method), 1250
- `basis()` (*sage.algebras.lie_algebras.verma_module.VermaModuleHomset* method), 1271
- `basis()` (*sage.algebras.lie_algebras.virasoro.VirasoroAlgebra* method), 1287
- `basis()` (*sage.algebras.octonion_algebra.OctonionAlgebra* method), 1048
- `basis()` (*sage.algebras.quantum_groups.quantum_group_gap.LowerHalfQuantumGroup* method), 548
- `basis()` (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroupModule* method), 564
- `basis()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract* method), 613
- `basis()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational* method), 622
- `basis()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionOrder* method), 643
- `basis()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic* method), 707
- `basis()` (*sage.algebras.weyl_algebra.DifferentialWeylAlgebra* method), 774
- `basis_coefficients()` (*sage.algebras.commutative_dga.GCAlgebra.Element* method), 980
- `basis_for_quaternion_lattice()` (in module *sage.algebras.quatalg.quaternion_algebra*), 654
- `basis_matrix()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_ideal.FiniteDimensionalAlgebraIdeal* method), 145
- `basis_matrix()` (*sage.algebras.lie_algebras.subalgebra.LieSubalgebra_finite_dimensional_with_basis* method), 1251
- `basis_matrix()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational* method), 622
- `basis_matrix()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionOrder* method), 643
- `basis_name()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic* method), 710
- `basis_name()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element* method), 690
- `BasisAbstract` (class in *sage.combinat.posets.moe-*

- `bius_algebra`), 480
`bch_iterator()` (in module `sage.algebras.lie_algebras.bch`), 1072
`bgg_resolution()` (`sage.algebras.lie_algebras.bgg_dual_module.FiniteDimensionalSimpleModule` method), 1082
`BGGDualModule` (class in `sage.algebras.lie_algebras.bgg_dual_module`), 1077
`BGGDualModule.Element` (class in `sage.algebras.lie_algebras.bgg_dual_module`), 1079
`BGGResolution` (class in `sage.algebras.lie_algebras.bgg_resolution`), 1074
`bijection_on_free_nodes()` (`sage.combinat.diagram_algebras.BrauerDiagram` method), 182
`binomial_mod2()` (in module `sage.algebras.steenrod.steenrod_algebra_mult`), 766
`binomial_modp()` (in module `sage.algebras.steenrod.steenrod_algebra_mult`), 766
`block_diagonal_list()` (`sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.CubicHeckeMatrixRep` method), 924
`BosonicGhostsLieConformalAlgebra` (class in `sage.algebras.lie_conformal_algebras.bosonic_ghosts_lie_conformal_algebra`), 1302
`boundary()` (`sage.algebras.clifford_algebra.ExteriorAlgebra` method), 274
`bracket()` (`sage.algebras.lie_algebras.lie_algebra_element.StructureCoefficientsElement` method), 1178
`bracket()` (`sage.algebras.lie_algebras.lie_algebra_element.UntwistedAffineLieAlgebraElement` method), 1180
`bracket_on_basis()` (`sage.algebras.lie_algebras.heisenberg.HeisenbergAlgebra_abstract` method), 1142
`bracket_on_basis()` (`sage.algebras.lie_algebras.onsager.OnsagerAlgebra` method), 1197
`bracket_on_basis()` (`sage.algebras.lie_algebras.onsager.OnsagerAlgebraACE` method), 1202
`bracket_on_basis()` (`sage.algebras.lie_algebras.rank_two_heisenberg_virasoro.RankTwoHeisenbergVirasoro` method), 1229
`bracket_on_basis()` (`sage.algebras.lie_algebras.symplectic_derivation.SymplecticDerivationLieAlgebra` method), 1259
`bracket_on_basis()` (`sage.algebras.lie_algebras.virasoro.LieAlgebraRegularVectorFields` method), 1282
`bracket_on_basis()` (`sage.algebras.lie_algebras.virasoro.VirasoroAlgebra` method), 1288
`bracket_on_basis()` (`sage.algebras.lie_algebras.virasoro.WittLieAlgebra_charp` method), 1291
`bracket_on_basis()` (`sage.combinat.free_prelie_algebra.bra.FreePreLieAlgebra` method), 1359
`braid_group()` (`sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra` method), 870
`braid_group_action()` (`sage.algebras.quantum_groups.quantum_group_gap.LowerHalfQuantumGroup.Element` method), 545
`braid_group_action()` (`sage.algebras.quantum_groups.quantum_group_gap.QuantumGroup.Element` method), 555
`braid_group_algebra()` (`sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra` method), 871
`braid_group_algebra_pre_image()` (`sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeElement` method), 886
`brauer_diagrams()` (in module `sage.combinat.diagram_algebras`), 248
`BrauerAlgebra` (class in `sage.combinat.diagram_algebras`), 179
`BrauerDiagram` (class in `sage.combinat.diagram_algebras`), 181
`BrauerDiagrams` (class in `sage.combinat.diagram_algebras`), 184
`build_differentials()` (in module `sage.algebras.lie_algebras.bgg_resolution`), 1076
- ## C
- `c()` (`sage.algebras.lie_algebras.affine_lie_algebra.AffineLieAlgebra` method), 1059
`c()` (`sage.algebras.lie_algebras.onsager.QuantumOnsagerAlgebra` method), 1207
`c()` (`sage.algebras.lie_algebras.virasoro.VirasoroAlgebra` method), 1288
`c_coefficient()` (`sage.algebras.lie_algebras.lie_algebra_element.UntwistedAffineLieAlgebraElement` method), 1181
`c_matrix()` (`sage.algebras.cluster_algebra.ClusterAlgebraSeed` method), 330
`C_prime` (`sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra` attribute), 834
`C_prime` (`sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_nonstandard` attribute), 850
`c_vector()` (`sage.algebras.cluster_algebra.ClusterAlgebraSeed` method), 331
`c_vectors()` (`sage.algebras.cluster_algebra.ClusterAlgebraSeed` method), 331
`canonical` (`sage.algebras.quantum_groups.fock_space.FockSpace` attribute), 32
`canonical` (`sage.algebras.quantum_groups.fock_space.FockSpaceTruncated` attribute), 44

- `canonical_basis_elements()` (*sage.algebras.quantum_groups.quantum_group_gap.LowerHalfQuantumGroup* method), 548
- `canonical_derivation()` (*sage.algebras.lie_algebras.lie_algebra_element.UntwistedAffineLieAlgebraElement* method), 1181
- `cardinality()` (*sage.algebras.clifford_algebra.CliffordAlgebraIndices* method), 273
- `cardinality()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra* method), 123
- `cardinality()` (*sage.algebras.lie_algebras.bgg_dual_module.SimpleModuleIndices* method), 1090
- `cardinality()` (*sage.combinat.diagram_algebras.BrauerDiagrams* method), 185
- `cardinality()` (*sage.combinat.diagram_algebras.HalfTemperleyLiebDiagrams* method), 189
- `cardinality()` (*sage.combinat.diagram_algebras.PartitionDiagrams* method), 225
- `cardinality()` (*sage.combinat.diagram_algebras.PlanarDiagrams* method), 228
- `cardinality()` (*sage.combinat.diagram_algebras.TemperleyLiebDiagrams* method), 248
- `cardinality()` (*sage.combinat.partition_algebra.SetPartitionsBk_k* method), 515
- `cardinality()` (*sage.combinat.partition_algebra.SetPartitionsBkhalf_k* method), 516
- `cardinality()` (*sage.combinat.partition_algebra.SetPartitionsIk_k* method), 517
- `cardinality()` (*sage.combinat.partition_algebra.SetPartitionsIkhalf_k* method), 517
- `cardinality()` (*sage.combinat.partition_algebra.SetPartitionsPk_k* method), 518
- `cardinality()` (*sage.combinat.partition_algebra.SetPartitionsPkhalf_k* method), 519
- `cardinality()` (*sage.combinat.partition_algebra.SetPartitionsPRk_k* method), 517
- `cardinality()` (*sage.combinat.partition_algebra.SetPartitionsPRkhalf_k* method), 517
- `cardinality()` (*sage.combinat.partition_algebra.SetPartitionsRk_k* method), 519
- `cardinality()` (*sage.combinat.partition_algebra.SetPartitionsRkhalf_k* method), 519
- `cardinality()` (*sage.combinat.partition_algebra.SetPartitionsSk_k* method), 521
- `cardinality()` (*sage.combinat.partition_algebra.SetPartitionsSkhalf_k* method), 521
- `cardinality()` (*sage.combinat.partition_algebra.SetPartitionsTk_k* method), 522
- `cardinality()` (*sage.combinat.partition_algebra.SetPartitionsTkhalf_k* method), 522
- `cartan_type()` (*sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra* method), 847
- `cartan_type()` (*sage.algebras.lie_algebras.affine_lie_algebra.AffineLieAlgebra* method), 1060
- `cartan_type()` (*sage.algebras.lie_algebras.classical_lie_algebra.ClassicalMatrixLieAlgebra* method), 1102
- `cartan_type()` (*sage.algebras.lie_conformal_algebras.affine_lie_conformal_algebra.AffineLieConformalAlgebra* method), 1302
- `cartan_type()` (*sage.algebras.q_system.QSystem* method), 1020
- `cartan_type()` (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroup* method), 560
- `cartan_type()` (*sage.algebras.quantum_groups.representations.QuantumGroupRepresentation* method), 57
- `casimir_element()` (*sage.algebras.askey_wilson.AskeyWilsonAlgebra* method), 160
- `casimir_element()` (*sage.algebras.lie_algebras.poincare_birkhoff_witt.PoincareBirkhoffWittBasis* method), 1213
- `cdg_algebra()` (*sage.algebras.commutative_dga.DifferentialGCAAlgebra* method), 952
- `cdg_algebra()` (*sage.algebras.commutative_dga.GCAAlgebra* method), 986
- `cdg_algebra()` (*sage.algebras.commutative_dga.GCAAlgebra_multigraded* method), 995
- `cell_module_indices()` (*sage.algebras.cellular_basis.CellularBasis* method), 1014
- `cell_module_indices()` (*sage.combinat.diagram_algebras.TemperleyLiebAlgebra* method), 245
- `cell_poset()` (*sage.algebras.cellular_basis.CellularBasis* method), 1015
- `cell_poset()` (*sage.combinat.diagram_algebras.TemperleyLiebAlgebra* method), 245
- `cellular_basis()` (*sage.algebras.cellular_basis.CellularBasis* method), 1015
- `cellular_basis_of()` (*sage.algebras.cellular_basis.CellularBasis* method), 1015
- `cellular_involution()` (*sage.combinat.diagram_algebras.TemperleyLiebAlgebra* method), 246
- `CellularBasis` (class in *sage.algebras.cellular_basis*), 1013
- `center()` (*sage.algebras.lie_algebras.poincare_birkhoff_witt.PoincareBirkhoffWittBasis* method), 1214
- `center_basis()` (*sage.algebras.clifford_algebra.CliffordAlgebra* method), 259
- `CenterIndices` (class in *sage.algebras.lie_algebras.center_uea*), 1092
- `CenterUEA` (class in *sage.algebras.lie_algebras.center_uea*), 1094
- `central_charge()` (*sage.algebras.lie_algebras.virasoro.VermaModule* method), 1285

- `central_elements()` (*sage.algebras.lie_conformal_algebras.finitely_freely_generated_lca.FinitelyFreelyGeneratedLCA* method), 1315
- `central_elements()` (*sage.algebras.lie_conformal_algebras.freely_generated_lie_conformal_algebra.FreelyGeneratedLieConformalAlgebra* method), 1317
- `certify_pentagons()` (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 395
- `chain_complex()` (*sage.algebras.clifford_algebra.ExteriorAlgebraBoundary* method), 286
- `chain_complex()` (*sage.algebras.clifford_algebra.ExteriorAlgebraCoboundary* method), 289
- `change_basis()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element* method), 691
- `change_ring()` (*sage.algebras.free_zinbiel_algebra.FreeZinbielAlgebra* method), 1388
- `change_ring()` (*sage.algebras.lie_algebras.structure_coefficients.LieAlgebraWithStructureCoefficients* method), 1241
- `change_ring()` (*sage.combinat.free_dendriform_algebra.FreeDendriformAlgebra* method), 1346
- `change_ring()` (*sage.combinat.free_prelie_algebra.FreePreLieAlgebra* method), 1359
- `change_ring()` (*sage.combinat.grossman_larson_algebras.GrossmanLarsonAlgebra* method), 474
- `char()` (*sage.algebras.fusion_rings.fusion_double.FusionDouble.Element* method), 422
- `characteristic_basis` (*sage.combinat.posets.moebius_algebra.QuantumMoebiusAlgebra* attribute), 486
- `characteristic_polynomial()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement* method), 139
- `characters()` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* method), 871
- `chargeless_representation()` (*sage.algebras.lie_algebras.virasoro.VirasoroAlgebra* method), 1288
- `ChargelessRepresentation` (class in *sage.algebras.lie_algebras.virasoro*), 1279
- `ChargelessRepresentation.Element` (class in *sage.algebras.lie_algebras.virasoro*), 1281
- `check()` (*sage.combinat.diagram_algebras.AbstractPartitionDiagram* method), 175
- `check()` (*sage.combinat.diagram_algebras.BrauerDiagram* method), 182
- `check()` (*sage.combinat.diagram_algebras.HalfTemperleyLiebDiagrams.Element* method), 188
- `check()` (*sage.combinat.diagram_algebras.IdealDiagram* method), 191
- `check()` (*sage.combinat.diagram_algebras.PlanarDiagram* method), 228
- `check()` (*sage.combinat.diagram_algebras.TemperleyLiebDiagram* method), 247
- `check()` (*sage.combinat.partition_algebra.SetPartitionSxkElement* method), 522
- `chevie()` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* method), 872
- `classical()` (*sage.algebras.lie_algebras.affine_lie_algebra.AffineLieAlgebra* method), 1060
- `ClassicalMatrixLieAlgebra` (class in *sage.algebras.lie_algebras.classical_lie_algebra*), 1101
- `clear_computed_data()` (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 309
- `clear_equations()` (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 396
- `clear_vars()` (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 397
- `CliffordAlgebra` (class in *sage.algebras.clifford_algebra*), 257
- `CliffordAlgebraIndices` (class in *sage.algebras.clifford_algebra*), 273
- `cluster_fan()` (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 309
- `cluster_variable()` (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 309
- `cluster_variable()` (*sage.algebras.cluster_algebra.ClusterAlgebraSeed* method), 331
- `cluster_variables()` (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 310
- `cluster_variables()` (*sage.algebras.cluster_algebra.ClusterAlgebraSeed* method), 332
- `cluster_variables_so_far()` (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 310
- `ClusterAlgebra` (class in *sage.algebras.cluster_algebra*), 305
- `ClusterAlgebraElement` (class in *sage.algebras.cluster_algebra*), 328
- `ClusterAlgebraSeed` (class in *sage.algebras.cluster_algebra*), 329
- `coboundaries()` (*sage.algebras.commutative_dga.Differential* method), 942
- `coboundaries()` (*sage.algebras.commutative_dga.Differential_multigraded* method), 974
- `coboundaries()` (*sage.algebras.commutative_dga.DifferentialGCAAlgebra* method), 953
- `coboundaries()` (*sage.algebras.commutative_dga.DifferentialGCAAlgebra_multigraded* method), 970
- `coboundary()` (*sage.algebras.clifford_algebra.ExteriorAlgebra* method), 275
- `cocycles()` (*sage.algebras.commutative_dga.Differential* method), 943
- `cocycles()` (*sage.algebras.commutative_dga.Differential_multigraded* method), 975

- `cocycles()` (*sage.algebras.commutative_dga.DifferentialGCAAlgebra method*), 954
- `cocycles()` (*sage.algebras.commutative_dga.DifferentialGCAAlgebra_multigraded method*), 971
- `coefficient()` (*sage.algebras.cluster_algebra.ClusterAlgebra method*), 311
- `coefficient_names()` (*sage.algebras.cluster_algebra.ClusterAlgebra method*), 311
- `coefficients()` (*sage.algebras.cluster_algebra.ClusterAlgebra method*), 311
- `cohomology()` (*sage.algebras.commutative_dga.Differential method*), 944
- `cohomology()` (*sage.algebras.commutative_dga.Differential_multigraded method*), 976
- `cohomology()` (*sage.algebras.commutative_dga.DifferentialGCAAlgebra method*), 954
- `cohomology()` (*sage.algebras.commutative_dga.DifferentialGCAAlgebra_multigraded method*), 972
- `cohomology_algebra()` (*sage.algebras.commutative_dga.DifferentialGCAAlgebra method*), 956
- `cohomology_class()` (*sage.algebras.commutative_dga.DifferentialGCAAlgebra.Element method*), 949
- `cohomology_generators()` (*sage.algebras.commutative_dga.DifferentialGCAAlgebra method*), 957
- `cohomology_raw()` (*sage.algebras.commutative_dga.Differential method*), 945
- `cohomology_raw()` (*sage.algebras.commutative_dga.Differential_multigraded method*), 976
- `cohomology_raw()` (*sage.algebras.commutative_dga.DifferentialGCAAlgebra method*), 958
- `cohomology_raw()` (*sage.algebras.commutative_dga.DifferentialGCAAlgebra_multigraded method*), 972
- `CohomologyClass` (class in *sage.algebras.commutative_dga*), 941
- `comm_long_mono_to_string()` (in module *sage.algebras.steenrod.steenrod_algebra_misc*), 747
- `comm_mono_to_string()` (in module *sage.algebras.steenrod.steenrod_algebra_misc*), 748
- `commutative_ring()` (*sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace method*), 86
- `compose()` (*sage.combinat.diagram_algebras.AbstractPartitionDiagram method*), 175
- `compute_known_powers()` (in module *sage.algebras.fusion_rings.poly_tup_engine*), 434
- `conformal_weight()` (*sage.algebras.lie_algebras.vera-soro.VermaModule method*), 1285
- `conj_matrix()` (*sage.algebras.fusion_rings.fusion_ring.FusionRing method*), 376
- `conjugate()` (*sage.algebras.octonion_algebra.Octonion_generic method*), 1050
- `conjugate()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational method*), 622
- `conjugation()` (*sage.algebras.hecke_algebras.cubic_hecke_base_ring.CubicHeckeExtensionRing method*), 897
- `constant_coeff()` (in module *sage.algebras.fusion_rings.poly_tup_engine*), 435
- `construction()` (*sage.algebras.free_algebra.FreeAlgebra_generic method*), 67
- `construction()` (*sage.algebras.free_zinbiel_algebra.FreeZinbielAlgebra method*), 1388
- `construction()` (*sage.algebras.hecke_algebras.cubic_hecke_base_ring.CubicHeckeExtensionRing method*), 898
- `construction()` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.CubicHeckeMatrixSpace method*), 927
- `construction()` (*sage.algebras.orlik_solomon.OrlikSolomonInvariantAlgebra method*), 509
- `construction()` (*sage.algebras.orlik_terao.OrlikTeraoInvariantAlgebra method*), 496
- `construction()` (*sage.algebras.schur_algebra.SchurTensorModule method*), 668
- `construction()` (*sage.algebras.tensor_algebra.TensorAlgebra method*), 116
- `construction()` (*sage.combinat.free_dendriform_algebra.FreeDendriformAlgebra method*), 1346
- `construction()` (*sage.combinat.free_prelie_algebra.FreePreLieAlgebra method*), 1360
- `contains_seed()` (*sage.algebras.cluster_algebra.ClusterAlgebra method*), 312
- `contravariant_form()` (*sage.algebras.lie_algebras.poincare_birkhoff_witt.PoincareBirkhoffWittBasisSemisimpleLieAlgebra method*), 1218
- `contravariant_form()` (*sage.algebras.lie_algebras.verma_module.VermaModule method*), 1262
- `convert_from_milnor_matrix()` (in module *sage.algebras.steenrod.steenrod_algebra_bases*), 734
- `convert_perm()` (in module *sage.algebras.steenrod.steenrod_algebra_misc*), 749
- `convert_to_milnor_matrix()` (in module *sage.algebras.steenrod.steenrod_algebra_bases*), 736
- `coproduct()` (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroup method*), 560
- `coproduct()` (*sage.algebras.shuffle_algebra.DualPBWBasis method*), 1372
- `coproduct()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method*), 710
- `coproduct()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method*), 692

- `coproduct_on_basis()` (*sage.algebras.clifford_algebra.ExteriorAlgebra* method), 275
- `coproduct_on_basis()` (*sage.algebras.free_zinbiel_algebra.FreeZinbielAlgebra* method), 1389
- `coproduct_on_basis()` (*sage.algebras.hall_algebra.HallAlgebra* method), 451
- `coproduct_on_basis()` (*sage.algebras.hall_algebra.HallAlgebraMonomials* method), 457
- `coproduct_on_basis()` (*sage.algebras.quantum_matrix_coordinate_algebra.QuantumGL* method), 573
- `coproduct_on_basis()` (*sage.algebras.quantum_matrix_coordinate_algebra.QuantumMatrixCoordinateAlgebra* method), 576
- `coproduct_on_basis()` (*sage.algebras.shuffle_algebra.ShuffleAlgebra* method), 1380
- `coproduct_on_basis()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic* method), 711
- `coproduct_on_basis()` (*sage.algebras.tensor_algebra.TensorAlgebra* method), 117
- `coproduct_on_basis()` (*sage.algebras.yangian.GradientYangianLoop* method), 788
- `coproduct_on_basis()` (*sage.algebras.yangian.Yangian* method), 793
- `coproduct_on_basis()` (*sage.combinat.free_dendriform_algebra.FreeDendriformAlgebra* method), 1347
- `coproduct_on_basis()` (*sage.combinat.grossman_larson_algebras.GrossmanLarsonAlgebra* method), 474
- `corolla()` (*sage.combinat.free_prelie_algebra.FreePrelieAlgebra* method), 1360
- `corolla_gen()` (in module *sage.combinat.free_prelie_algebra*), 1369
- `counit()` (*sage.algebras.clifford_algebra.ExteriorAlgebra* method), 276
- `counit()` (*sage.algebras.free_zinbiel_algebra.FreeZinbielAlgebra* method), 1389
- `counit()` (*sage.algebras.hall_algebra.HallAlgebra* method), 452
- `counit()` (*sage.algebras.hall_algebra.HallAlgebraMonomials* method), 458
- `counit()` (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroup* method), 561
- `counit()` (*sage.algebras.shuffle_algebra.DualPBWBasis* method), 1373
- `counit()` (*sage.algebras.shuffle_algebra.ShuffleAlgebra* method), 1380
- `counit()` (*sage.algebras.tensor_algebra.TensorAlgebra* method), 118
- `counit_on_basis()` (*sage.algebras.quantum_matrix_coordinate_algebra.QuantumMatrixCoordinateAlgebra_abstract* method), 578
- `counit_on_basis()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic* method), 712
- `counit_on_basis()` (*sage.algebras.yangian.GradientYangianLoop* method), 789
- `counit_on_basis()` (*sage.algebras.yangian.Yangian* method), 793
- `counit_on_basis()` (*sage.combinat.grossman_larson_algebras.GrossmanLarsonAlgebra* method), 475
- `count_blocks_of_size()` (*sage.combinat.diagram_algebras.AbstractPartitionDiagram* method), 175
- `coxeter_element()` (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 312
- `coxeter_group()` (*sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra* method), 847
- `coxeter_type()` (*sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra* method), 848
- `create_key()` (*sage.algebras.free_algebra.FreeAlgebraFactory* method), 66
- `create_key()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebraFactory* method), 595
- `create_object()` (*sage.algebras.free_algebra.FreeAlgebraFactory* method), 66
- `create_object()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebraFactory* method), 595
- `create_specialization()` (*sage.algebras.hecke_algebras.cubic_hecke_base_ring.CubicHeckeExtensionRing* method), 898
- `create_specialization()` (*sage.algebras.hecke_algebras.cubic_hecke_base_ring.CubicHeckeRingOfDefinition* method), 907
- `cross_product()` (in module *sage.algebras.lie_algebras.examples*), 1123
- `crystal_basis()` (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroupModule* method), 565
- `crystal_graph()` (*sage.algebras.quantum_groups.quantum_group_gap.HighestWeightSubmodule* method), 542
- `crystal_graph()` (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroupModule* method), 565
- `CrystalGraphVertex` (class in *sage.algebras.quantum_groups.quantum_group_gap*), 539
- `cubic_braid_group()` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* method), 872
- `cubic_braid_group_algebra()` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* method), 873
- `cubic_braid_group_algebra_pre_image()`

(*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeElement* method), 887

cubic_equation() (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* method), 873

cubic_equation() (*sage.algebras.hecke_algebras.cubic_hecke_base_ring.CubicHeckeRingOfDefinition* method), 908

cubic_equation_galois_group() (*sage.algebras.hecke_algebras.cubic_hecke_base_ring.CubicHeckeExtensionRing* method), 900

cubic_equation_parameters() (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* method), 874

cubic_equation_roots() (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* method), 874

cubic_hecke_subalgebra() (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* method), 875

CubicHeckeAlgebra (class in *sage.algebras.hecke_algebras.cubic_hecke_algebra*), 865

CubicHeckeElement (class in *sage.algebras.hecke_algebras.cubic_hecke_algebra*), 885

CubicHeckeExtensionRing (class in *sage.algebras.hecke_algebras.cubic_hecke_base_ring*), 895

CubicHeckeMatrixRep (class in *sage.algebras.hecke_algebras.cubic_hecke_matrix_rep*), 923

CubicHeckeMatrixSpace (class in *sage.algebras.hecke_algebras.cubic_hecke_matrix_rep*), 925

CubicHeckeRingOfDefinition (class in *sage.algebras.hecke_algebras.cubic_hecke_base_ring*), 905

current_ring() (*sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace* method), 87

current_seed() (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 313

cyclic_right_subideals() (*sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational* method), 623

CyclicRepresentation (class in *sage.algebras.quantum_groups.representations*), 52

cyclotomic_generator() (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* method), 875

cyclotomic_generator() (*sage.algebras.hecke_algebras.cubic_hecke_base_ring.CubicHeckeExtensionRing* method), 900

cyclotomic_parameters() (*sage.algebras.hecke_algebras.ariki_koike_algebra.ArikiKoikeAlgebra* method), 815

D

d() (*sage.algebras.lie_algebras.affine_lie_algebra.AffineLieAlgebra* method), 1060

d() (*sage.algebras.lie_algebras.virasoro.VirasoroAlgebra* method), 1289

d() (*sage.algebras.quantum_groups.fock_space.FockSpace.F.Element* method), 21

d_coefficient() (*sage.algebras.lie_algebras.lie_algebra_element.UntwistedAffineLieAlgebraElement* method), 1181

D_minus() (*sage.algebras.fusion_rings.fusion_double.FusionDouble* method), 421

D_minus() (*sage.algebras.fusion_rings.fusion_ring.FusionRing* method), 371

D_plus() (*sage.algebras.fusion_rings.fusion_double.FusionDouble* method), 422

D_plus() (*sage.algebras.fusion_rings.fusion_ring.FusionRing* method), 372

d_vector() (*sage.algebras.cluster_algebra.ClusterAlgebraElement* method), 328

d_vector_to_g_vector() (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 313

dagger() (*sage.algebras.quantum_groups.ace_quantum_onsager.ACEQuantumOnsagerAlgebra* method), 6

data_section() (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.RepresentationType* method), 930

defects() (*sage.combinat.diagram_algebras.HalfTemperleyLiebDiagrams.Element* method), 189

defining_ideal() (*sage.algebras.lie_algebras.quotient.LieQuotient_finite_dimensional_with_basis* method), 1225

defining_polynomial() (*sage.algebras.splitting_algebra.SplittingAlgebra* method), 1037

defining_polynomial() (*sage.algebras.yangian.YangianLevel* method), 798

deformed_euler() (*sage.algebras.rational_cherednik_algebra.RationalCherednikAlgebra* method), 660

degbound() (*sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace* method), 87

degree() (*sage.algebras.commutative_dga.GCAlgebra_multigraded.Element* method), 994

degree() (*sage.algebras.commutative_dga.GCAlgebra.Element* method), 981

degree() (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra* method), 123

degree() (*sage.algebras.letterplace.free_algebra_element_letterplace.FreeAlgebraElement_letterplace* method), 92

degree() (*sage.algebras.lie_algebras.center_uea.Cen-*

- terIndices method*), 1092
- `degree()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method*), 694
- `degree_on_basis()` (*sage.algebras.associated_graded.AssociatedGradedAlgebra method*), 1010
- `degree_on_basis()` (*sage.algebras.clifford_algebra.CliffordAlgebra method*), 261
- `degree_on_basis()` (*sage.algebras.clifford_algebra.ExteriorAlgebra method*), 276
- `degree_on_basis()` (*sage.algebras.down_up_algebra.DownUpAlgebra method*), 356
- `degree_on_basis()` (*sage.algebras.finite_gca.FiniteGCAAlgebra method*), 936
- `degree_on_basis()` (*sage.algebras.free_algebra.FreeAlgebra_generic method*), 67
- `degree_on_basis()` (*sage.algebras.free_zinbiel_algebra.FreeZinbielAlgebra method*), 1389
- `degree_on_basis()` (*sage.algebras.lie_algebras.bgg_dual_module.BGGDualModule method*), 1079
- `degree_on_basis()` (*sage.algebras.lie_algebras.center_uea.CenterUEA method*), 1097
- `degree_on_basis()` (*sage.algebras.lie_algebras.classical_lie_algebra.LieAlgebraChevalleyBasis method*), 1106
- `degree_on_basis()` (*sage.algebras.lie_algebras.onsager.QuantumOnsagerAlgebra method*), 1207
- `degree_on_basis()` (*sage.algebras.lie_algebras.poincare_birkhoff_witt.PoincareBirkhoffWittBasis method*), 1215
- `degree_on_basis()` (*sage.algebras.lie_algebras.symplectic_derivation.SymplecticDerivationLieAlgebra method*), 1260
- `degree_on_basis()` (*sage.algebras.lie_algebras.verma_module.VermaModule method*), 1263
- `degree_on_basis()` (*sage.algebras.lie_algebras.virasoro.ChargelessRepresentation method*), 1281
- `degree_on_basis()` (*sage.algebras.lie_algebras.virasoro.LieAlgebraRegularVectorFields method*), 1283
- `degree_on_basis()` (*sage.algebras.lie_algebras.virasoro.VermaModule method*), 1286
- `degree_on_basis()` (*sage.algebras.lie_algebras.virasoro.VirasoroAlgebra method*), 1289
- `degree_on_basis()` (*sage.algebras.lie_algebras.virasoro.WittLieAlgebra_charp method*), 1291
- `degree_on_basis()` (*sage.algebras.orlik_solomon.OrlikSolomonAlgebra method*), 501
- `degree_on_basis()` (*sage.algebras.orlik_terao.OrlikTeraoAlgebra method*), 489
- `degree_on_basis()` (*sage.algebras.q_commuting_polynomials.qCommutingPolynomials_generic method*), 1032
- `degree_on_basis()` (*sage.algebras.quantum_groups.ace_quantum_onsager.ACEQuantumOnsagerAlgebra method*), 6
- `degree_on_basis()` (*sage.algebras.rational_cherednik_algebra.RationalCherednikAlgebra method*), 660
- `degree_on_basis()` (*sage.algebras.shuffle_algebra.DualPBWBasis method*), 1373
- `degree_on_basis()` (*sage.algebras.shuffle_algebra.ShuffleAlgebra method*), 1380
- `degree_on_basis()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method*), 712
- `degree_on_basis()` (*sage.algebras.tensor_algebra.TensorAlgebra method*), 118
- `degree_on_basis()` (*sage.algebras.weyl_algebra.DifferentialWeylAlgebra method*), 775
- `degree_on_basis()` (*sage.algebras.yangian.Yangian method*), 794
- `degree_on_basis()` (*sage.combinat.free_dendriform_algebra.FreeDendriformAlgebra method*), 1348
- `degree_on_basis()` (*sage.combinat.free_prelie_algebra.FreePreLieAlgebra method*), 1361
- `degree_on_basis()` (*sage.combinat.grossman_larson_algebras.GrossmanLarsonAlgebra method*), 476
- `delta()` (*sage.combinat.posets.incidence_algebras.IncidenceAlgebra method*), 462
- `delta()` (*sage.combinat.posets.incidence_algebras.ReducedIncidenceAlgebra method*), 466
- `DendriformFunctor` (class in *sage.combinat.free_dendriform_algebra*), 1341
- `depth()` (*sage.algebras.cluster_algebra.ClusterAlgebraSeed method*), 332
- `derived_series()` (*sage.algebras.lie_algebras.affine_lie_algebra.AffineLieAlgebra method*), 1061
- `derived_subalgebra()` (*sage.algebras.lie_algebras.affine_lie_algebra.TwistedAffineLieAlgebra method*), 1067
- `derived_subalgebra()` (*sage.algebras.lie_algebras.affine_lie_algebra.UntwistedAffineLieAlgebra method*), 1071
- `DescentAlgebra` (class in *sage.combinat.descent_algebra*), 338
- `DescentAlgebra.B` (class in *sage.combinat.descent_algebra*), 340
- `DescentAlgebraBases` (class in *sage.combinat.descent_algebra*), 350
- `DescentAlgebraBases.ElementMethods` (class in *sage.combinat.descent_algebra*), 350
- `DescentAlgebraBases.ParentMethods` (class in

- sage.combinat.descent_algebra*), 351
- DescentAlgebra.D* (class in *sage.combinat.descent_algebra*), 343
- DescentAlgebra.I* (class in *sage.combinat.descent_algebra*), 346
- diagram()* (*sage.combinat.diagram_algebras.AbstractPartitionDiagram* method), 176
- diagram()* (*sage.combinat.diagram_algebras.DiagramAlgebra.Element* method), 186
- diagram_basis()* (*sage.combinat.diagram_algebras.OrbitBasis* method), 194
- diagram_latex()* (in module *sage.combinat.diagram_algebras*), 249
- DiagramAlgebra* (class in *sage.combinat.diagram_algebras*), 186
- DiagramAlgebra.Element* (class in *sage.combinat.diagram_algebras*), 186
- DiagramBasis* (class in *sage.combinat.diagram_algebras*), 188
- diagrams()* (*sage.combinat.diagram_algebras.DiagramAlgebra.Element* method), 187
- dict()* (*sage.algebras.commutative_dga.GCAlgebra.Element* method), 982
- dict()* (*sage.algebras.octonion_algebra.Octonion_generic* method), 1050
- dict()* (*sage.algebras.splitting_algebra.SplittingAlgebraElement* method), 1040
- diff()* (*sage.algebras.weyl_algebra.DifferentialWeylAlgebraElement* method), 780
- diff_action()* (*sage.algebras.weyl_algebra.DifferentialWeylAlgebra* method), 776
- Differential* (class in *sage.algebras.commutative_dga*), 942
- differential()* (*sage.algebras.commutative_dga.DifferentialGCAlgebra* method), 959
- differential()* (*sage.algebras.commutative_dga.DifferentialGCAlgebra.Element* method), 950
- differential()* (*sage.algebras.commutative_dga.GCAlgebra* method), 987
- differential()* (*sage.algebras.commutative_dga.GCAlgebra_multigraded* method), 996
- differential_matrix()* (*sage.algebras.commutative_dga.Differential* method), 946
- differential_matrix_multigraded()* (*sage.algebras.commutative_dga.Differential_multigraded* method), 978
- Differential_multigraded* (class in *sage.algebras.commutative_dga*), 974
- DifferentialGCAlgebra* (class in *sage.algebras.commutative_dga*), 948
- DifferentialGCAlgebra_multigraded* (class in *sage.algebras.commutative_dga*), 970
- DifferentialGCAlgebra_multigraded.Element* (class in *sage.algebras.commutative_dga*), 970
- DifferentialGCAlgebra.Element* (class in *sage.algebras.commutative_dga*), 949
- differentials()* (*sage.algebras.weyl_algebra.DifferentialWeylAlgebra* method), 776
- DifferentialWeylAlgebra* (class in *sage.algebras.weyl_algebra*), 773
- DifferentialWeylAlgebraAction* (class in *sage.algebras.weyl_algebra*), 779
- DifferentialWeylAlgebraElement* (class in *sage.algebras.weyl_algebra*), 780
- dimension()* (*sage.algebras.clifford_algebra.CliffordAlgebra* method), 262
- dimension()* (*sage.algebras.free_algebra_quotient.FreeAlgebraQuotient* method), 108
- dimension()* (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibeRep* method), 922
- dimension()* (*sage.algebras.lie_algebras.abelian.InfiniteDimensionalAbelianLieAlgebra* method), 1057
- dimension()* (*sage.algebras.lie_algebras.structure_coefficients.LieAlgebraWithStructureCoefficients* method), 1242
- dimension()* (*sage.algebras.lie_algebras.verma_module.VermaModuleHomset* method), 1272
- dimension()* (*sage.algebras.q_commuting_polynomials.qCommutingPolynomials_generic* method), 1033
- dimension()* (*sage.algebras.q_system.QSystem* method), 1021
- dimension()* (*sage.algebras.quantum_clifford.QuantumCliffordAlgebra* method), 530
- dimension()* (*sage.algebras.schur_algebra.SchurAlgebra* method), 664
- dimension()* (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic* method), 713
- dimension()* (*sage.algebras.yangian.Yangian* method), 795
- discriminant()* (*sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab* method), 596
- discriminant()* (*sage.algebras.quatalg.quaternion_algebra.QuaternionOrder* method), 644
- DownUpAlgebra* (class in *sage.algebras.down_up_algebra*), 353
- dual()* (*sage.algebras.fusion_rings.fusion_double.FusionDouble* method), 426
- dual()* (*sage.algebras.fusion_rings.fusion_double.FusionDouble.Element* method), 422
- dual()* (*sage.algebras.lie_algebras.bgg_dual_module.BG-GDualModule* method), 1080
- dual()* (*sage.algebras.lie_algebras.bgg_dual_module.SimpleModule* method), 1083
- dual()* (*sage.algebras.lie_algebras.verma_module.VermaModule* method), 1264
- dual()* (*sage.combinat.diagram_algebras.AbstractParti-*

- tionDiagram* method), 176
- `dual()` (*sage.combinat.diagram_algebras.PartitionAlgebra.Element* method), 205
- `dual_pbw_basis()` (*sage.algebras.shuffle_algebra.ShuffleAlgebra* method), 1381
- `DualPBWBasis` (class in *sage.algebras.shuffle_algebra*), 1370
- `DualPBWBasis.Element` (class in *sage.algebras.shuffle_algebra*), 1371
- ## E
- `e()` (*sage.algebras.lie_algebras.affine_lie_algebra.AffineLieAlgebra* method), 1061
- `e()` (*sage.algebras.lie_algebras.classical_lie_algebra.ClassicalMatrixLieAlgebra* method), 1103
- `e()` (*sage.algebras.lie_algebras.poincare_birkhoff_witt.PoincareBirkhoffWittBasisSemisimpleLieAlgebra* method), 1220
- `E()` (*sage.algebras.lie_algebras.rank_two_heisenberg_virasoro.RankTwoHeisenbergVirasoro* method), 1229
- `e()` (*sage.algebras.quantum_groups.fock_space.FockSpace.F.Element* method), 22
- `E()` (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroup* method), 554
- `e()` (*sage.algebras.yokonuma_hecke_algebra.YokonumaHeckeAlgebra* method), 859
- `e()` (*sage.combinat.diagram_algebras.PartitionAlgebra* method), 211
- `e6` (class in *sage.algebras.lie_algebras.classical_lie_algebra*), 1115
- `e7` (class in *sage.algebras.lie_algebras.classical_lie_algebra*), 1115
- `e8` (class in *sage.algebras.lie_algebras.classical_lie_algebra*), 1115
- `e_on_basis()` (*sage.algebras.quantum_groups.representations.AdjointRepresentation* method), 49
- `e_on_basis()` (*sage.algebras.quantum_groups.representations.MinusculeRepresentation* method), 54
- `E_simple()` (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroup* method), 554
- `e_tilde()` (*sage.algebras.quantum_groups.quantum_group_gap.QuaGroupModuleElement* method), 551
- `Element` (*sage.algebras.clifford_algebra.CliffordAlgebra* attribute), 259
- `Element` (*sage.algebras.clifford_algebra.ExteriorAlgebra* attribute), 274
- `Element` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra* attribute), 122
- `Element` (*sage.algebras.free_algebra_quotient.FreeAlgebraQuotient* attribute), 108
- `Element` (*sage.algebras.free_algebra.FreeAlgebra_generic* attribute), 67
- `Element` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* attribute), 870
- `Element` (*sage.algebras.lie_algebras.affine_lie_algebra.UntwistedAffineLieAlgebra* attribute), 1071
- `Element` (*sage.algebras.lie_algebras.free_lie_algebra.FreeLieBasis_abstract* attribute), 1139
- `Element` (*sage.algebras.lie_algebras.onsager.OnsagerAlgebra* attribute), 1197
- `Element` (*sage.algebras.lie_algebras.onsager.OnsagerAlgebraACE* attribute), 1202
- `Element` (*sage.algebras.lie_algebras.verma_module.VermaModuleHomset* attribute), 1271
- `Element` (*sage.algebras.octonion_algebra.OctonionAlgebra* attribute), 1048
- `Element` (*sage.algebras.quantum_groups.quantum_group_gap.HighestWeightModule* attribute), 539
- `Element` (*sage.algebras.quantum_groups.quantum_group_gap.HighestWeightSubmodule* attribute), 541
- `Element` (*sage.algebras.quantum_groups.quantum_group_gap.TensorProductOfHighestWeightModules* attribute), 567
- `Element` (*sage.algebras.splitting_algebra.SplittingAlgebra* attribute), 1037
- `Element` (*sage.algebras.weyl_algebra.DifferentialWeylAlgebra* attribute), 774
- `Element` (*sage.combinat.diagram_algebras.AbstractPartitionDiagrams* attribute), 179
- `Element` (*sage.combinat.diagram_algebras.BrauerDiagrams* attribute), 185
- `Element` (*sage.combinat.diagram_algebras.IdealDiagrams* attribute), 191
- `Element` (*sage.combinat.diagram_algebras.PartitionDiagrams* attribute), 225
- `Element` (*sage.combinat.diagram_algebras.PlanarDiagrams* attribute), 228
- `Element` (*sage.combinat.diagram_algebras.TemperleyLiebDiagrams* attribute), 248
- `Element` (*sage.combinat.partition_algebra.SetPartition-sAk_k* attribute), 514
- `Element` (*sage.combinat.partition_algebra.SetPartition-sAkhalf_k* attribute), 514
- `epsilon()` (*sage.algebras.lie_algebras.classical_lie_algebra.ClassicalMatrixLieAlgebra* method), 1103
- `equations_graph()` (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 397
- `euler_matrix()` (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 314
- `ExceptionalJordanAlgebra` (class in *sage.alge-*

- bras.jordan_algebra*), 1323
- ExceptionalJordanAlgebra.Element (class in *sage.algebras.jordan_algebra*), 1325
- ExceptionalMatrixLieAlgebra (class in *sage.algebras.lie_algebras.classical_lie_algebra*), 1105
- excess() (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element* method), 695
- executor() (in module *sage.algebras.fusion_rings.fast_parallel_fmats_methods*), 433
- executor() (in module *sage.algebras.fusion_rings.fast_parallel_fusion_ring_braid_repn*), 434
- expand() (*sage.algebras.free_algebra.PBWBasisOfFreeAlgebra.Element* method), 77
- expand() (*sage.algebras.shuffle_algebra.DualPBWBasis.Element* method), 1371
- expansion() (*sage.algebras.free_algebra.PBWBasisOfFreeAlgebra* method), 77
- expansion() (*sage.algebras.shuffle_algebra.DualPBWBasis* method), 1374
- expansion_on_basis() (*sage.algebras.shuffle_algebra.DualPBWBasis* method), 1374
- explore_to_depth() (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 314
- extension_ring() (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* method), 876
- extension_ring() (*sage.algebras.hecke_algebras.cubic_hecke_base_ring.CubicHeckeRingOfDefinition* method), 909
- exterior_algebra_basis() (in module *sage.algebras.commutative_dga*), 1003
- ExteriorAlgebra (class in *sage.algebras.clifford_algebra*), 273
- ExteriorAlgebraBoundary (class in *sage.algebras.clifford_algebra*), 283
- ExteriorAlgebraCoboundary (class in *sage.algebras.clifford_algebra*), 286
- ExteriorAlgebraDifferential (class in *sage.algebras.clifford_algebra*), 290
- ExteriorAlgebraIdeal (class in *sage.algebras.clifford_algebra*), 291
- ## F
- f() (*sage.algebras.lie_algebras.affine_lie_algebra.AffineLieAlgebra* method), 1062
- f() (*sage.algebras.lie_algebras.classical_lie_algebra.ClassicalMatrixLieAlgebra* method), 1103
- f() (*sage.algebras.lie_algebras.poincare_birkhoff_witt.PoincareBirkhoffWittBasisSemisimpleLieAlgebra* method), 1220
- f() (*sage.algebras.quantum_groups.fock_space.FockSpace.F.Element* method), 24
- F() (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroup* method), 557
- f4 (class in *sage.algebras.lie_algebras.classical_lie_algebra*), 1115
- f_from() (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 398
- f_on_basis() (*sage.algebras.quantum_groups.representations.AdjointRepresentation* method), 50
- f_on_basis() (*sage.algebras.quantum_groups.representations.MinusculeRepresentation* method), 55
- F_polynomial() (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 307
- F_polynomial() (*sage.algebras.cluster_algebra.ClusterAlgebraSeed* method), 329
- F_polynomial() (*sage.algebras.cluster_algebra.PrincipalClusterAlgebraElement* method), 336
- F_polynomials() (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 307
- F_polynomials() (*sage.algebras.cluster_algebra.ClusterAlgebraSeed* method), 329
- F_polynomials_so_far() (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 308
- F_simple() (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroup* method), 558
- f_tilde() (*sage.algebras.quantum_groups.quantum_group_gap.QuaGroupModuleElement* method), 551
- f_to() (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 399
- factor_differentials() (*sage.algebras.weyl_algebra.DifferentialWeylAlgebraElement* method), 781
- FaithfulRepresentationNilpotentPBW (class in *sage.algebras.lie_algebras.representation*), 1231
- FaithfulRepresentationNilpotentPBW.Element (class in *sage.algebras.lie_algebras.representation*), 1233
- FaithfulRepresentationPBWPosChar (class in *sage.algebras.lie_algebras.representation*), 1233
- FaithfulRepresentationPBWPosChar.Element (class in *sage.algebras.lie_algebras.representation*), 1234
- FermionicGhostsLieConformalAlgebra (class in *sage.algebras.lie_conformal_algebras.fermionic_ghosts_lie_conformal_algebra*), 1304
- field() (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 399
- field() (*sage.algebras.fusion_rings.fusion_double.FusionDouble* method), 427
- field() (*sage.algebras.fusion_rings.fusion_ring.FusionRing* method), 376

- `field_embedding()` (*sage.algebras.hecke_algebras.cubic_hecke_base_ring.CubicHeckeExtensionRing* method), 901
- `filecache_section()` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* method), 876
- `find_cyclotomic_solution()` (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 400
- `find_g_vector()` (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 315
- `find_orthogonal_solution()` (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 402
- `findcases()` (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 404
- `FiniteDimensionalAlgebra` (class in *sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra*), 121
- `FiniteDimensionalAlgebraElement` (class in *sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element*), 138
- `FiniteDimensionalAlgebraHomset` (class in *sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism*), 146
- `FiniteDimensionalAlgebraIdeal` (class in *sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_ideal*), 145
- `FiniteDimensionalAlgebraMorphism` (class in *sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism*), 147
- `FiniteDimensionalSimpleModule` (class in *sage.algebras.lie_algebras.bgg_dual_module*), 1082
- `FiniteGCAAlgebra` (class in *sage.algebras.finite_gca*), 933
- `FinitelyFreelyGeneratedLCA` (class in *sage.algebras.lie_conformal_algebras.finitely_freely_generated_lca*), 1315
- `FinitelyGeneratedLieAlgebra` (class in *sage.algebras.lie_algebras.lie_algebra*), 1151
- `fmat()` (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 404
- `FMatrix` (class in *sage.algebras.fusion_rings.f_matrix*), 391
- `fmatrix()` (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 405
- `fmats_are_orthogonal()` (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 406
- `fock_space_representation()` (*sage.algebras.quantum_oscillator.QuantumOscillatorAlgebra* method), 585
- `FockSpace` (class in *sage.algebras.quantum_groups.fock_space*), 13
- `FockSpace.A` (class in *sage.algebras.quantum_groups.fock_space*), 19
- `FockSpaceBases` (class in *sage.algebras.quantum_groups.fock_space*), 34
- `FockSpaceBases.ParentMethods` (class in *sage.algebras.quantum_groups.fock_space*), 34
- `FockSpace.F` (class in *sage.algebras.quantum_groups.fock_space*), 20
- `FockSpace.F.Element` (class in *sage.algebras.quantum_groups.fock_space*), 21
- `FockSpace.G` (class in *sage.algebras.quantum_groups.fock_space*), 27
- `FockSpaceOptions()` (in module *sage.algebras.quantum_groups.fock_space*), 37
- `FockSpaceRepresentation` (class in *sage.algebras.quantum_oscillator*), 581
- `FockSpaceRepresentation.Element` (class in *sage.algebras.quantum_oscillator*), 581
- `FockSpaceTruncated` (class in *sage.algebras.quantum_groups.fock_space*), 39
- `FockSpaceTruncated.A` (class in *sage.algebras.quantum_groups.fock_space*), 40
- `FockSpaceTruncated.F` (class in *sage.algebras.quantum_groups.fock_space*), 41
- `FockSpaceTruncated.F.Element` (class in *sage.algebras.quantum_groups.fock_space*), 42
- `FockSpaceTruncated.G` (class in *sage.algebras.quantum_groups.fock_space*), 42
- `formal_markov_trace()` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeElement* method), 887
- `FR()` (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 394
- `free_algebra()` (*sage.algebras.free_algebra_quotient.FreeAlgebraQuotient* method), 108
- `free_algebra()` (*sage.algebras.free_algebra.PBWBasisOfFreeAlgebra* method), 78
- `free_module()` (*sage.algebras.clifford_algebra.CliffordAlgebra* method), 262
- `free_module()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract* method), 614
- `free_module()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational* method), 625
- `free_module()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionOrder* method), 644
- `FreeAlgebra_generic` (class in *sage.algebras.free_algebra*), 66
- `FreeAlgebra_letterplace` (class in *sage.algebras.letterplace.free_algebra_letterplace*), 85
- `FreeAlgebra_letterplace_libsingular` (class in *sage.algebras.letterplace.free_algebra_letterplace*), 91
- `FreeAlgebraElement` (class in *sage.algebras.free_algebra_element*), 80
- `FreeAlgebraElement_letterplace` (class in *sage.algebras.letterplace.free_algebra_element_letterplace*), 91

- FreeAlgebraFactory (class in *sage.algebras.free_algebra*), 63
- FreeAlgebraQuotient (class in *sage.algebras.free_algebra_quotient*), 106
- FreeAlgebraQuotientElement (class in *sage.algebras.free_algebra_quotient_element*), 112
- FreeBosonsLieConformalAlgebra (class in *sage.algebras.lie_conformal_algebras.free_bosons_lie_conformal_algebra*), 1305
- FreeDendriformAlgebra (class in *sage.combinat.free_dendriform_algebra*), 1343
- FreeFermionsLieConformalAlgebra (class in *sage.algebras.lie_conformal_algebras.free_fermions_lie_conformal_algebra*), 1307
- FreeLieAlgebra (class in *sage.algebras.lie_algebras.free_lie_algebra*), 1133
- FreeLieAlgebraBases (class in *sage.algebras.lie_algebras.free_lie_algebra*), 1138
- FreeLieAlgebraElement (class in *sage.algebras.lie_algebras.free_lie_algebra_element*), 1173
- FreeLieAlgebra.Hall (class in *sage.algebras.lie_algebras.free_lie_algebra*), 1134
- FreeLieAlgebra.Lyndon (class in *sage.algebras.lie_algebras.free_lie_algebra*), 1135
- FreeLieBasis_abstract (class in *sage.algebras.lie_algebras.free_lie_algebra*), 1138
- FreelyGeneratedLieConformalAlgebra (class in *sage.algebras.lie_conformal_algebras.freely_generated_lie_conformal_algebra*), 1316
- FreeNilpotentLieAlgebra (class in *sage.algebras.lie_algebras.nilpotent_lie_algebra*), 1191
- FreePreLieAlgebra (class in *sage.combinat.free_prelie_algebra*), 1354
- FreePreLieAlgebra.Element (class in *sage.combinat.free_prelie_algebra*), 1357
- FreeZinbielAlgebra (class in *sage.algebras.free_zinbiel_algebra*), 1385
- from_base_ring() (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra* method), 124
- from_involution_permutation_triple() (*sage.combinat.diagram_algebras.BrauerDiagrams* method), 185
- from_vector() (*sage.algebras.lie_algebras.quotient.LieQuotient_finite_dimensional_with_basis* method), 1225
- from_vector() (*sage.algebras.lie_algebras.structure_coefficients.LieAlgebraWithStructureCoefficients* method), 1242
- from_vector() (*sage.algebras.lie_algebras.subalgebra.LieSubalgebra_finite_dimensional_with_basis* method), 1251
- fusion_l() (*sage.algebras.fusion_rings.fusion_ring.FusionRing* method), 377
- fusion_labels() (*sage.algebras.fusion_rings.fusion_ring.FusionRing* method), 377
- fusion_level() (*sage.algebras.fusion_rings.fusion_ring.FusionRing* method), 378
- FusionDouble (class in *sage.algebras.fusion_rings.fusion_double*), 419
- FusionDouble.Element (class in *sage.algebras.fusion_rings.fusion_double*), 422
- FusionRing (class in *sage.algebras.fusion_rings.fusion_ring*), 365
- FusionRing.Element (class in *sage.algebras.fusion_rings.fusion_ring*), 372
- fvars_are_real() (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 406
- fvars_field() (*sage.algebras.fusion_rings.fusion_double.FusionDouble* method), 427
- fvars_field() (*sage.algebras.fusion_rings.fusion_ring.FusionRing* method), 379
- FvarsHandler (class in *sage.algebras.fusion_rings.shm_managers*), 440
- ## G
- g() (*sage.algebras.fusion_rings.fusion_double.FusionDouble.Element* method), 423
- g() (*sage.algebras.yokonuma_hecke_algebra.YokonumaHeckeAlgebra* method), 860
- g2 (class in *sage.algebras.lie_algebras.classical_lie_algebra*), 1116
- g_algebra() (*sage.algebras.free_algebra.FreeAlgebra_generic* method), 68
- g_matrix() (*sage.algebras.cluster_algebra.ClusterAlgebraSeed* method), 333
- g_vector() (*sage.algebras.cluster_algebra.ClusterAlgebraSeed* method), 333
- g_vector() (*sage.algebras.cluster_algebra.PrincipalClusterAlgebraElement* method), 336
- g_vector_to_d_vector() (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 315
- g_vectors() (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 316
- g_vectors() (*sage.algebras.cluster_algebra.ClusterAlgebraSeed* method), 334
- g_vectors_so_far() (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 316
- GaloisGroupAction (class in *sage.algebras.hecke_algebras.cubic_hecke_base_ring*), 915
- gap() (*sage.algebras.quantum_groups.quantum_group_gap.QuaGroupModuleElement* method), 552
- gap() (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroup* method), 552

- 561
- `gap()` (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroupModule* method), 566
- `gap_index()` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* method), 922
- `garside_involution()` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* method), 877
- `GCAlgebra` (class in *sage.algebras.commutative_dga*), 979
- `GCAlgebra_multigraded` (class in *sage.algebras.commutative_dga*), 992
- `GCAlgebra_multigraded.Element` (class in *sage.algebras.commutative_dga*), 993
- `GCAlgebra.Element` (class in *sage.algebras.commutative_dga*), 980
- `GCAlgebraHomset` (class in *sage.algebras.commutative_dga*), 988
- `GCAlgebraMorphism` (class in *sage.algebras.commutative_dga*), 990
- `gen()` (*sage.algebras.associated_graded.AssociatedGradedAlgebra* method), 1010
- `gen()` (*sage.algebras.clifford_algebra.CliffordAlgebra* method), 262
- `gen()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra* method), 124
- `gen()` (*sage.algebras.finite_gca.FiniteGCAlgebra* method), 936
- `gen()` (*sage.algebras.free_algebra_quotient.FreeAlgebraQuotient* method), 109
- `gen()` (*sage.algebras.free_algebra.FreeAlgebra_generic* method), 69
- `gen()` (*sage.algebras.free_algebra.PBWBasisOfFreeAlgebra* method), 78
- `gen()` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* method), 877
- `gen()` (*sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace* method), 88
- `gen()` (*sage.algebras.lie_algebras.free_lie_algebra.FreeLieAlgebra* method), 1137
- `gen()` (*sage.algebras.lie_algebras.heisenberg.HeisenbergAlgebra_fd* method), 1144
- `gen()` (*sage.algebras.lie_algebras.lie_algebra.LieAlgebraWithGenerators* method), 1170
- `gen()` (*sage.algebras.q_commuting_polynomials.qCommutingPolynomials_generic* method), 1033
- `gen()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab* method), 597
- `gen()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionOrder* method), 645
- `gen()` (*sage.algebras.shuffle_algebra.DualPBWBasis* method), 1374
- `gen()` (*sage.algebras.shuffle_algebra.ShuffleAlgebra* method), 1381
- `gen()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic* method), 714
- `gen()` (*sage.algebras.weyl_algebra.DifferentialWeylAlgebra* method), 777
- `gen()` (*sage.algebras.yangian.Yangian* method), 795
- `gen()` (*sage.algebras.yangian.YangianLevel* method), 799
- `gen()` (*sage.combinat.free_dendriform_algebra.FreeDendriformAlgebra* method), 1348
- `gen()` (*sage.combinat.free_prelie_algebra.FreePreLieAlgebra* method), 1362
- `generator_a()` (*sage.combinat.diagram_algebras.PartitionAlgebra* method), 212
- `generator_degrees()` (*sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace* method), 88
- `generator_e()` (*sage.combinat.diagram_algebras.PartitionAlgebra* method), 213
- `generator_s()` (*sage.combinat.diagram_algebras.PartitionAlgebra* method), 214
- `gens()` (*sage.algebras.askey_wilson.AskeyWilsonAlgebra* method), 161
- `gens()` (*sage.algebras.clifford_algebra.CliffordAlgebra* method), 263
- `gens()` (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 317
- `gens()` (*sage.algebras.down_up_algebra.DownUpAlgebra* method), 357
- `gens()` (*sage.algebras.finite_gca.FiniteGCAlgebra* method), 937
- `gens()` (*sage.algebras.free_algebra_quotient.FreeAlgebraQuotient* method), 109
- `gens()` (*sage.algebras.free_algebra.FreeAlgebra_generic* method), 69
- `gens()` (*sage.algebras.free_algebra.PBWBasisOfFreeAlgebra* method), 78
- `gens()` (*sage.algebras.free_zinbiel_algebra.FreeZinbielAlgebra* method), 1390
- `gens()` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* method), 878
- `gens()` (*sage.algebras.jordan_algebra.ExceptionalJordanAlgebra* method), 1327
- `gens()` (*sage.algebras.jordan_algebra.JordanAlgebraSymmetricBilinear* method), 1337
- `gens()` (*sage.algebras.jordan_algebra.SpecialJordanAlgebra* method), 1340
- `gens()` (*sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace* method), 88
- `gens()` (*sage.algebras.lie_algebras.classical_lie_algebra.LieAlgebraChevalleyBasis* method), 1107
- `gens()` (*sage.algebras.lie_algebras.free_lie_algebra.FreeLieAlgebra* method), 1137
- `gens()` (*sage.algebras.lie_algebras.heisenberg.Heisenber-*

- gAlgebra_fd* method), 1144
- `gens()` (*sage.algebras.lie_algebras.lie_algebra.LieAlgebraWithGenerators* method), 1170
- `gens()` (*sage.algebras.lie_algebras.onsager.QuantumOnsagerAlgebra* method), 1207
- `gens()` (*sage.algebras.lie_algebras.poincare_birkhoff_witt.PoincareBirkhoffWittBasis* method), 1215
- `gens()` (*sage.algebras.lie_algebras.subalgebra.LieSubalgebraFiniteDimensionalWithBasis* method), 1252
- `gens()` (*sage.algebras.lie_algebras.verma_module.VermaModule* method), 1264
- `gens()` (*sage.algebras.lie_conformal_algebras.finitely_freely_generated_lca.FinitelyFreelyGeneratedLCA* method), 1315
- `gens()` (*sage.algebras.octonion_algebra.OctonionAlgebra* method), 1048
- `gens()` (*sage.algebras.q_commuting_polynomials.qCommutingPolynomials_generic* method), 1034
- `gens()` (*sage.algebras.q_system.QSystem* method), 1021
- `gens()` (*sage.algebras.quantum_clifford.QuantumCliffordAlgebra* method), 530
- `gens()` (*sage.algebras.quantum_groups.ace_quantum_onsager.ACEQuantumOnsagerAlgebra* method), 7
- `gens()` (*sage.algebras.quantum_groups.quantum_group_gap.LowerHalfQuantumGroup* method), 549
- `gens()` (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroup* method), 562
- `gens()` (*sage.algebras.quantum_matrix_coordinate_algebra.QuantumMatrixCoordinateAlgebra_abstract* method), 578
- `gens()` (*sage.algebras.quantum_oscillator.QuantumOscillatorAlgebra* method), 586
- `gens()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab* method), 598
- `gens()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionOrder* method), 646
- `gens()` (*sage.algebras.shuffle_algebra.DualPBWBasis* method), 1375
- `gens()` (*sage.algebras.shuffle_algebra.ShuffleAlgebra* method), 1382
- `gens()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic* method), 716
- `gens()` (*sage.algebras.tensor_algebra.TensorAlgebra* method), 118
- `gens()` (*sage.algebras.weyl_algebra.DifferentialWeylAlgebra* method), 777
- `gens()` (*sage.algebras.yangian.YangianLevel* method), 799
- `gens()` (*sage.algebras.yokonuma_hecke_algebra.YokonumaHeckeAlgebra* method), 860
- `gens()` (*sage.combinat.free_dendriform_algebra.FreeDendriformAlgebra* method), 1349
- `gens()` (*sage.combinat.free_prelie_algebra.FreePreLieAlgebra* method), 1362
- `gens_satisfy_braid_gp_rels()` (*sage.algebras.fusion_rings.fusion_ring.FusionRing* method), 380
- `GenSign` (class in *sage.algebras.hecke_algebras.cubic_hecke_matrix_rep*), 929
- `get_basis_name()` (in module *sage.algebras.steenrod.steenrod_algebra_misc*), 749
- `get_braid_generators()` (*sage.algebras.fusion_rings.fusion_ring.FusionRing* method), 380
- `get_coerce_map_from_fr_cyclotomic_field()` (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 407
- `get_computational_basis()` (*sage.algebras.fusion_rings.fusion_ring.FusionRing* method), 382
- `get_defining_equations()` (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 408
- `get_fmatrix()` (*sage.algebras.fusion_rings.fusion_double.FusionDouble* method), 427
- `get_fmatrix()` (*sage.algebras.fusion_rings.fusion_ring.FusionRing* method), 383
- `get_fr_str()` (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 409
- `get_fvars()` (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 409
- `get_fvars_by_size()` (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 410
- `get_fvars_in_alg_field()` (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 411
- `get_non_cyclotomic_roots()` (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 411
- `get_order()` (*sage.algebras.fusion_rings.fusion_ring.FusionRing* method), 383
- `get_order()` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* method), 878
- `get_order()` (*sage.algebras.lie_algebras.lie_algebra.LieAlgebra* method), 1162
- `get_orthogonality_constraints()` (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 412
- `get_poly_ring()` (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 413
- `get_qqbar_embedding()` (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 413
- `get_radical_expression()` (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 414
- `get_variables_degrees()` (in module *sage.algebras.fusion_rings.poly_tup_engine*), 436

- `gl` (class in `sage.algebras.lie_algebras.classical_lie_algebra`), 1116
`GL_irreducible_character()` (in module `sage.algebras.schur_algebra`), 663
`gl.Element` (class in `sage.algebras.lie_algebras.classical_lie_algebra`), 1116
`global_q_dimension()` (`sage.algebras.fusion_rings.fusion_double.FusionDouble` method), 428
`global_q_dimension()` (`sage.algebras.fusion_rings.fusion_ring.FusionRing` method), 384
`goldman_involution_on_basis()` (`sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.A` method), 828
`goldman_involution_on_basis()` (`sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.B` method), 831
`goldman_involution_on_basis()` (`sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T` method), 840
`graded_algebra()` (`sage.algebras.clifford_algebra.CliffordAlgebra` method), 263
`graded_algebra()` (`sage.algebras.yangian.Yangian` method), 796
`graded_basis()` (`sage.algebras.lie_algebras.free_lie_algebra.FreeLieAlgebra.Hall` method), 1134
`graded_basis()` (`sage.algebras.lie_algebras.free_lie_algebra.FreeLieAlgebra.Lyndon` method), 1135
`graded_basis()` (`sage.algebras.lie_algebras.free_lie_algebra.FreeLieBasis_abstract` method), 1139
`graded_commutative_algebra()` (`sage.algebras.commutative_dga.DifferentialGCA` method), 960
`graded_dimension()` (`sage.algebras.lie_algebras.free_lie_algebra.FreeLieBasis_abstract` method), 1139
`GradedCommutativeAlgebra()` (in module `sage.algebras.commutative_dga`), 998
`GradedLieBracket` (class in `sage.algebras.lie_algebras.lie_algebra_element`), 1174
`GradedLieConformalAlgebra` (class in `sage.algebras.lie_conformal_algebras.graded_lie_conformal_algebra`), 1318
`GradedYangianBase` (class in `sage.algebras.yangian`), 787
`GradedYangianLoop` (class in `sage.algebras.yangian`), 787
`GradedYangianNatural` (class in `sage.algebras.yangian`), 789
`gram_matrix()` (`sage.algebras.lie_conformal_algebras.free_bosons_lie_conformal_algebra.FreeBosonsLieConformalAlgebra` method), 1306
`gram_matrix()` (`sage.algebras.lie_conformal_algebras.free_fermions_lie_conformal_algebra.FreeFermionsLieConformalAlgebra` method), 1308
`gram_matrix()` (`sage.algebras.lie_conformal_algebras.weyl_lie_conformal_algebra.WeylLieConformalAlgebra` method), 1314
`gram_matrix()` (`sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational` method), 627
`greedy_element()` (`sage.algebras.cluster_algebra.ClusterAlgebra` method), 317
`groebner_basis()` (`sage.algebras.clifford_algebra.ExteriorAlgebraIdeal` method), 292
`groebner_basis()` (`sage.algebras.letterplace.letterplace_ideal.LetterplaceIdeal` method), 103
`groebner_basis()` (`sage.algebras.lie_algebras.representation.FaithfulRepresentationPBWPosChar` method), 1234
`GrossmanLarsonAlgebra` (class in `sage.combinat.grossman_larson_algebras`), 471
`group()` (`sage.algebras.fusion_rings.fusion_double.FusionDouble` method), 428
`group_product()` (`sage.combinat.free_prelie_algebra.FreePreLieAlgebra` method), 1362
`GroupAlgebra()` (in module `sage.algebras.group_algebra`), 469
`GroupAlgebra_class` (class in `sage.algebras.group_algebra`), 471
- ## H
- `h()` (`sage.algebras.lie_algebras.classical_lie_algebra.ClassicalMatrixLieAlgebra` method), 1104
`h()` (`sage.algebras.quantum_groups.fock_space.FockSpace.F.Element` method), 25
`h_inverse()` (`sage.algebras.quantum_groups.fock_space.FockSpace.F.Element` method), 26
`HalfTemperleyLiebDiagrams` (class in `sage.combinat.diagram_algebras`), 188
`HalfTemperleyLiebDiagrams.Element` (class in `sage.combinat.diagram_algebras`), 188
`HallAlgebra` (class in `sage.algebras.hall_algebra`), 447
`HallAlgebra.Element` (class in `sage.algebras.hall_algebra`), 450
`HallAlgebraMonomials` (class in `sage.algebras.hall_algebra`), 454
`HallAlgebraMonomials.Element` (class in `sage.algebras.hall_algebra`), 456
`hamilton_quatalg()` (in module `sage.algebras.free_algebra_quotient`), 111
`has_no_braid_relation()` (`sage.algebras.affine_nil_temperley_lieb.AffineNilTemperleyLiebTypeA` method), 152
`hash_involution_on_basis()` (`sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.C`

- method*), 833
- `hash_involution_on_basis()` (*sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.Cp* *method*), 837
- `hash_involution_on_basis()` (*sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T* *method*), 841
- `hecke_parameter()` (*sage.algebras.hecke_algebras.ariki_koike_algebra.ArikiKoikeAlgebra* *method*), 815
- `Heisenberg()` (*in module sage.algebras.lie_algebras.examples*), 1122
- `HeisenbergAlgebra` (*class in sage.algebras.lie_algebras.heisenberg*), 1141
- `HeisenbergAlgebra_abstract` (*class in sage.algebras.lie_algebras.heisenberg*), 1142
- `HeisenbergAlgebra_abstract.Element` (*class in sage.algebras.lie_algebras.heisenberg*), 1142
- `HeisenbergAlgebra_fd` (*class in sage.algebras.lie_algebras.heisenberg*), 1143
- `HeisenbergAlgebra_matrix` (*class in sage.algebras.lie_algebras.heisenberg*), 1145
- `HeisenbergAlgebra_matrix.Element` (*class in sage.algebras.lie_algebras.heisenberg*), 1148
- `highest_root_basis_elt()` (*sage.algebras.lie_algebras.classical_lie_algebra.ClassicalMatrixLieAlgebra* *method*), 1104
- `highest_root_basis_elt()` (*sage.algebras.lie_algebras.classical_lie_algebra.LieAlgebraChevalleyBasis* *method*), 1107
- `highest_weight()` (*sage.algebras.lie_algebras.bgg_dual_module.BGGDualModule* *method*), 1081
- `highest_weight()` (*sage.algebras.lie_algebras.bgg_dual_module.SimpleModule* *method*), 1083
- `highest_weight()` (*sage.algebras.lie_algebras.verma_module.VermaModule* *method*), 1265
- `highest_weight_decomposition()` (*sage.algebras.quantum_groups.quantum_group_gap.TensorProductOfHighestWeightModules* *method*), 567
- `highest_weight_image()` (*sage.algebras.lie_algebras.verma_module.VermaModuleHomset* *method*), 1272
- `highest_weight_module()` (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroup* *method*), 562
- `highest_weight_vector()` (*sage.algebras.down_up_algebra.VermaModule* *method*), 363
- `highest_weight_vector()` (*sage.algebras.lie_algebras.bgg_dual_module.BGGDualModule* *method*), 1081
- `highest_weight_vector()` (*sage.algebras.lie_algebras.bgg_dual_module.SimpleModule* *method*), 1084
- `highest_weight_vector()` (*sage.algebras.lie_algebras.verma_module.VermaModule* *method*), 1265
- `highest_weight_vector()` (*sage.algebras.lie_algebras.virasoro.VermaModule* *method*), 1286
- `highest_weight_vector()` (*sage.algebras.quantum_groups.fock_space.FockSpace* *method*), 32
- `highest_weight_vector()` (*sage.algebras.quantum_groups.fock_space.FockSpaceBases.ParentMethods* *method*), 35
- `highest_weight_vector()` (*sage.algebras.quantum_groups.quantum_group_gap.HighestWeightModule* *method*), 540
- `highest_weight_vector()` (*sage.algebras.quantum_groups.quantum_group_gap.HighestWeightSubmodule* *method*), 542
- `highest_weight_vector()` (*sage.algebras.quantum_groups.quantum_group_gap.LowerHalfQuantumGroup* *method*), 549
- `highest_weight_vectors()` (*sage.algebras.quantum_groups.quantum_group_gap.TensorProductOfHighestWeightModules* *method*), 568
- `HighestWeightModule` (*class in sage.algebras.quantum_groups.quantum_group_gap*), 539
- `HighestWeightSubmodule` (*class in sage.algebras.quantum_groups.quantum_group_gap*), 540
- `hom()` (*sage.algebras.hecke_algebras.cubic_hecke_base_ring.CubicHeckeExtensionRing* *method*), 903
- `hom()` (*sage.algebras.splitting_algebra.SplittingAlgebra* *method*), 1037
- `homogeneous_component()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic* *method*), 717
- `homogeneous_component_basis()` (*sage.algebras.lie_algebras.bgg_dual_module.SimpleModule* *method*), 1084
- `homogeneous_component_basis()` (*sage.algebras.lie_algebras.verma_module.VermaModule* *method*), 1265
- `homogeneous_components()` (*sage.algebras.cluster_algebra.PrincipalClusterAlgebraElement* *method*), 337
- `homogeneous_generator_noncommutative_variables()` (*sage.algebras.nil_coxeter_algebra.NilCoxeterAlgebra* *method*), 855
- `homogeneous_noncommutative_variables()` (*sage.algebras.nil_coxeter_algebra.NilCoxeterAlgebra* *method*), 856

- homogeneous_parts() (sage.algebras.commutative_dga.GCAlgebra.Element method), 983
- homology() (sage.algebras.clifford_algebra.ExteriorAlgebraDifferential method), 290
- homology() (sage.algebras.commutative_dga.Differential method), 947
- homology() (sage.algebras.commutative_dga.Differential_multigraded method), 979
- homology() (sage.algebras.commutative_dga.DifferentialGCAlgebra method), 960
- homology() (sage.algebras.commutative_dga.DifferentialGCAlgebra_multigraded method), 974
- I**
- ideal() (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 124
- ideal() (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab method), 598
- ideal_diagrams() (in module sage.combinat.diagram_algebras), 249
- ideal_monoid() (sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace method), 89
- IdealDiagram (class in sage.combinat.diagram_algebras), 190
- IdealDiagrams (class in sage.combinat.diagram_algebras), 191
- idempotent (sage.combinat.descent_algebra.DescentAlgebra attribute), 350
- idempotent (sage.combinat.posets.moebius_algebra.MoebiusAlgebra attribute), 482
- idempotent() (sage.combinat.descent_algebra.DescentAlgebra.I method), 347
- identity() (in module sage.combinat.partition_algebra), 523
- identity() (sage.algebras.commutative_dga.GCAlgebra-Homset method), 989
- identity_set_partition() (in module sage.combinat.diagram_algebras), 250
- im_gens() (sage.algebras.lie_algebras.morphism.LieAlgebraHomomorphism_im_gens method), 1186
- im_gens() (sage.algebras.quantum_groups.quantum_group_gap.QuantumGroupMorphism method), 566
- imag_part() (sage.algebras.octonion_algebra.Octonion_generic method), 1051
- image() (sage.algebras.lie_algebras.verma_module.VermaModuleMorphism method), 1277
- IncidenceAlgebra (class in sage.combinat.posets.incidence_algebras), 460
- IncidenceAlgebra.Element (class in sage.combinat.posets.incidence_algebras), 460
- index_cmp() (in module sage.algebras.iwahori_hecke_algebra), 853
- index_set() (sage.algebras.affine_nil_temperley_lieb.AffineNilTemperleyLiebTypeA method), 153
- index_set() (sage.algebras.lie_algebras.classical_lie_algebra.ClassicalMatrixLieAlgebra method), 1104
- index_set() (sage.algebras.q_system.QSystem method), 1021
- indices() (sage.algebras.lie_algebras.lie_algebra.LieAlgebraWithGenerators method), 1170
- indices() (sage.algebras.lie_algebras.subalgebra.LieSubalgebra_finite_dimensional_with_basis method), 1252
- indices_to_positive_roots_map() (sage.algebras.lie_algebras.classical_lie_algebra.LieAlgebraChevalleyBasis method), 1107
- InfiniteDimensionalAbelianLieAlgebra (class in sage.algebras.lie_algebras.abelian), 1056
- InfiniteDimensionalAbelianLieAlgebra.Element (class in sage.algebras.lie_algebras.abelian), 1057
- InfiniteHeisenbergAlgebra (class in sage.algebras.lie_algebras.heisenberg), 1150
- InfinitelyGeneratedLieAlgebra (class in sage.algebras.lie_algebras.lie_algebra), 1151
- initial_cluster_variable() (sage.algebras.cluster_algebra.ClusterAlgebra method), 317
- initial_cluster_variable_names() (sage.algebras.cluster_algebra.ClusterAlgebra method), 318
- initial_cluster_variables() (sage.algebras.cluster_algebra.ClusterAlgebra method), 318
- initial_seed() (sage.algebras.cluster_algebra.ClusterAlgebra method), 318
- inject_shorthands() (sage.algebras.quantum_groups.fock_space.FockSpace method), 33
- inject_variables() (sage.algebras.fusion_rings.fusion_double.FusionDouble method), 428
- inner_product_matrix() (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab method), 598
- inner_product_matrix() (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract method), 614
- interior_product_on_basis() (sage.algebras.clifford_algebra.ExteriorAlgebra method), 277
- internal_index() (sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibeRep method), 922
- intersection() (sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational method), 627

`intersection()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionOrder* method), 646
`intersection_of_row_modules_over_ZZ()` (in module *sage.algebras.quatalg.quaternion_algebra*), 654
`invariants()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab* method), 599
`inverse()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement* method), 140
`inverse()` (*sage.algebras.quantum_clifford.QuantumCliffordAlgebraGeneric.Element* method), 532
`inverse()` (*sage.algebras.quantum_clifford.QuantumCliffordAlgebraRootUnity.Element* method), 535
`inverse_g()` (*sage.algebras.yokonuma_hecke_algebra.YokonumaHeckeAlgebra* method), 860
`inverse_generator()` (*sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T* method), 842
`inverse_generators()` (*sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T* method), 843
`inverse_image()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism.FiniteDimensionalAlgebraMorphism* method), 148
`inverse_T()` (*sage.algebras.hecke_algebras.ariki_koike_algebra.ArikiKoikeAlgebra.LT* method), 809
`involution_permutation_triple()` (*sage.combinat.diagram_algebras.BrauerDiagram* method), 182
`irred_repr` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* attribute), 878
`is_abelian()` (*sage.algebras.lie_algebras.abelian.AbelianLieAlgebra* method), 1056
`is_abelian()` (*sage.algebras.lie_algebras.abelian.FiniteDimensionalAbelianLieAlgebra* method), 1057
`is_abelian()` (*sage.algebras.lie_algebras.free_lie_algebra.FreeLieBasis_abstract* method), 1140
`is_abelian()` (*sage.algebras.lie_algebras.lie_algebra.LieAlgebraFromAssociative* method), 1167
`is_acyclic()` (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 319
`is_associative()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra* method), 125
`is_coboundary()` (*sage.algebras.commutative_dga.DifferentialGCAAlgebra.Element* method), 950
`is_cohomologous_to()` (*sage.algebras.commutative_dga.DifferentialGCAAlgebra.Element* method), 951
`is_commutative()` (*sage.algebras.clifford_algebra.CliffordAlgebra* method), 264
`is_commutative()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra* method), 126
`is_commutative()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract* method), 615
`is_commutative()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic* method), 719
`is_completely_split()` (*sage.algebras.splitting_algebra.SplittingAlgebra* method), 1038
`is_decomposable()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element* method), 696
`is_definite()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab* method), 600
`is_division_algebra()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract* method), 615
`is_division_algebra()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic* method), 720
`is_elementary_symmetric()` (*sage.combinat.diagram_algebras.BrauerDiagram* method), 183
`is_equivalent()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational* method), 628
`is_exact()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract* method), 616
`is_field()` (*sage.algebras.free_algebra.FreeAlgebra_generic* method), 70
`is_field()` (*sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace* method), 89
`is_field()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract* method), 617
`is_field()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic* method), 721
`is_filecache_empty()` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* method), 878
`is_finite()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra* method), 126
`is_finite()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract* method), 617
`is_finite()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic* method), 721
`is_formal()` (*sage.algebras.commutative_dga.DifferentialGCAAlgebra* method), 961
`is_FreeAlgebra()` (in module *sage.algebras.free_algebra*), 80

- `is_FreeAlgebraQuotientElement()` (in module `sage.algebras.free_algebra_quotient_element`), 113
- `is_generic()` (`sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic` method), 721
- `is_graded()` (`sage.algebras.commutative_dga.GCAlgebraMorphism` method), 991
- `is_homogeneous()` (`sage.algebras.cluster_algebra.PrincipalClusterAlgebraElement` method), 337
- `is_homogeneous()` (`sage.algebras.commutative_dga.GCAlgebra.Element` method), 983
- `is_homogeneous()` (`sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element` method), 697
- `is_ideal()` (`sage.algebras.lie_algebras.subalgebra.LieSubalgebra_finite_dimensional_with_basis` method), 1253
- `is_injective()` (`sage.algebras.lie_algebras.verma_module.VermaModuleMorphism` method), 1278
- `is_integral()` (`sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational` method), 628
- `is_integral_domain()` (`sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract` method), 617
- `is_integral_domain()` (`sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic` method), 722
- `is_invertible()` (`sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement` method), 140
- `is_isomorphic()` (`sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab` method), 600
- `is_left_equivalent()` (`sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational` method), 629
- `is_lyndon()` (in module `sage.algebras.lie_algebras.free_lie_algebra`), 1140
- `is_matrix_ring()` (`sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract` method), 618
- `is_maximal()` (`sage.algebras.quatalg.quaternion_algebra.QuaternionOrder` method), 647
- `is_monomial()` (`sage.algebras.lie_conformal_algebras.lie_conformal_algebra_element.LCAWithGeneratorsElement` method), 1299
- `is_multiplicity_free()` (`sage.algebras.fusion_rings.fusion_double.FusionDouble` method), 428
- `is_multiplicity_free()` (`sage.algebras.fusion_rings.fusion_ring.FusionRing` method), 384
- `is_nilpotent()` (`sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement` method), 141
- `is_nilpotent()` (`sage.algebras.lie_algebras.abelian.AbelianLieAlgebra` method), 1056
- `is_nilpotent()` (`sage.algebras.lie_algebras.abelian.InfiniteDimensionalAbelianLieAlgebra` method), 1057
- `is_nilpotent()` (`sage.algebras.lie_algebras.affine_lie_algebra.AffineLieAlgebra` method), 1062
- `is_nilpotent()` (`sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element` method), 697
- `is_noetherian()` (`sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract` method), 619
- `is_noetherian()` (`sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic` method), 722
- `is_planar()` (in module `sage.combinat.diagram_algebras`), 250
- `is_planar()` (in module `sage.combinat.partition_algebra`), 523
- `is_planar()` (`sage.combinat.diagram_algebras.AbstractPartitionDiagram` method), 177
- `is_primitive()` (`sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational` method), 629
- `is_principal()` (`sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational` method), 630
- `is_projective()` (`sage.algebras.lie_algebras.verma_module.VermaModule` method), 1266
- `is_QuaternionAlgebra()` (in module `sage.algebras.quatalg.quaternion_algebra`), 655
- `is_regular()` (`sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.RepresentationType` method), 931
- `is_right_equivalent()` (`sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational` method), 630
- `is_simple()` (`sage.algebras.lie_algebras.verma_module.VermaModule` method), 1267
- `is_simple_object()` (`sage.algebras.fusion_rings.fusion_double.FusionDouble.Element` method), 423
- `is_simple_object()` (`sage.algebras.fusion_rings.fusion_ring.FusionRing.Element` method), 372
- `is_singular()` (`sage.algebras.lie_algebras.verma_module.VermaModule` method), 1268
- `is_solvable()` (`sage.algebras.lie_algebras.abelian.AbelianLieAlgebra` method),

- 1056
- `is_solvable()` (*sage.algebras.lie_algebras.abelian.InfiniteDimensionalAbelianLieAlgebra* method), 1057
- `is_solvable()` (*sage.algebras.lie_algebras.affine_lie_algebra.AffineLieAlgebra* method), 1062
- `is_split()` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.RepresentationType* method), 931
- `is_surjective()` (*sage.algebras.lie_algebras.verma_module.VermaModuleMorphism* method), 1279
- `is_tamely_laced()` (in module *sage.algebras.q_system*), 1022
- `is_totally_definite()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab* method), 601
- `is_unit()` (*sage.algebras.free_algebra_element.FreeAlgebraElement* method), 80
- `is_unit()` (*sage.algebras.octonion_algebra.Octonion_generic* method), 1051
- `is_unit()` (*sage.algebras.splitting_algebra.SplittingAlgebraElement* method), 1041
- `is_unit()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element* method), 698
- `is_unit()` (*sage.combinat.posets.incidence_algebras.IncidenceAlgebra.Element* method), 460
- `is_unit()` (*sage.combinat.posets.incidence_algebras.ReducedIncidenceAlgebra.Element* method), 465
- `is_unitary()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra* method), 127
- `is_valid_profile()` (in module *sage.algebras.steenrod.steenrod_algebra_misc*), 751
- `is_weight_vector()` (*sage.algebras.down_up_algebra.VermaModule.Element* method), 362
- `is_zero()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra* method), 129
- `is_zerodivisor()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement* method), 141
- `isomorphism_to()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionOrder* method), 647
- `items()` (*sage.algebras.fusion_rings.shm_managers.FvarsHandler* method), 442
- `items()` (*sage.algebras.fusion_rings.shm_managers.KSHandler* method), 444
- IwahoriHeckeAlgebra* (class in *sage.algebras.iwahori_hecke_algebra*), 821
- IwahoriHeckeAlgebra_nonstandard* (class in *sage.algebras.iwahori_hecke_algebra*), 849
- IwahoriHeckeAlgebra_nonstandard.C* (class in *sage.algebras.iwahori_hecke_algebra*), 849
- IwahoriHeckeAlgebra_nonstandard.Cp* (class in *sage.algebras.iwahori_hecke_algebra*), 850
- IwahoriHeckeAlgebra_nonstandard.T* (class in *sage.algebras.iwahori_hecke_algebra*), 851
- IwahoriHeckeAlgebra.A* (class in *sage.algebras.iwahori_hecke_algebra*), 828
- IwahoriHeckeAlgebra.B* (class in *sage.algebras.iwahori_hecke_algebra*), 829
- IwahoriHeckeAlgebra.C* (class in *sage.algebras.iwahori_hecke_algebra*), 832
- IwahoriHeckeAlgebra.Cp* (class in *sage.algebras.iwahori_hecke_algebra*), 834
- IwahoriHeckeAlgebra.T* (class in *sage.algebras.iwahori_hecke_algebra*), 839
- IwahoriHeckeAlgebra.T.Element* (class in *sage.algebras.iwahori_hecke_algebra*), 839
- ## J
- JordanAlgebra* (class in *sage.algebras.jordan_algebra*), 1332
- JordanAlgebraSymmetricBilinear* (class in *sage.algebras.jordan_algebra*), 1335
- JordanAlgebraSymmetricBilinear.Element* (class in *sage.algebras.jordan_algebra*), 1335
- `jucys_murphy()` (*sage.combinat.diagram_algebras.BrauerAlgebra* method), 181
- `jucys_murphy_element()` (*sage.combinat.diagram_algebras.PartitionAlgebra* method), 215
- ## K
- `K()` (*sage.algebras.lie_algebras.rank_two_heisenberg_virasoro.RankTwoHeisenbergVirasoro* method), 1229
- `K()` (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroup* method), 558
- `K_inverse()` (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroup* method), 558
- `K_on_basis()` (*sage.algebras.quantum_groups.representations.QuantumGroupRepresentation* method), 56
- `k_schur_noncommutative_variables()` (*sage.algebras.nil_coxeter_algebra.NilCoxeterAlgebra* method), 857
- `kazhdan_lusztig` (*sage.combinat.posets.moebius_algebra.QuantumMoebiusAlgebra* attribute), 486
- `killing_form()` (*sage.algebras.lie_algebras.classical_lie_algebra.gl* method), 1117
- `killing_form()` (*sage.algebras.lie_algebras.classical_lie_algebra.LieAlgebraChevalleyBasis* method), 1108

- `killring_form()` (*sage.algebras.lie_algebras.classical_lie_algebra.sl method*), 1118
`killring_form()` (*sage.algebras.lie_algebras.classical_lie_algebra.so method*), 1119
`killring_form()` (*sage.algebras.lie_algebras.classical_lie_algebra.sp method*), 1120
`killring_form_matrix()` (*sage.algebras.lie_algebras.classical_lie_algebra.LieAlgebraChevalleyBasis method*), 1108
`KSHandler` (*class in sage.algebras.fusion_rings.shm_managers*), 443
- ## L
- `L()` (*sage.algebras.hecke_algebras.ariki_koike_algebra.ArikiKoikeAlgebra.LT method*), 807
`L()` (*sage.algebras.hecke_algebras.ariki_koike_algebra.ArikiKoikeAlgebra.T method*), 811
`L()` (*sage.algebras.hecke_algebras.ariki_koike_specht_modules.SpechtModule.Element method*), 818
`L()` (*sage.combinat.diagram_algebras.PartitionAlgebra method*), 206
`largest_fmat_size()` (*sage.algebras.fusion_rings.f_matrix.FMatrix method*), 415
`lattice()` (*sage.combinat.posets.moebius_algebra.MoebiusAlgebra method*), 482
`lattice()` (*sage.combinat.posets.moebius_algebra.QuantumMoebiusAlgebra method*), 486
`lc()` (*sage.algebras.letterplace.free_algebra_element_letterplace.FreeAlgebraElement_letterplace method*), 93
`LCAStructureCoefficientsElement` (*class in sage.algebras.lie_conformal_algebras.lie_conformal_algebra_element*), 1298
`LCAWithGeneratorsElement` (*class in sage.algebras.lie_conformal_algebras.lie_conformal_algebra_element*), 1298
`leading_monomials()` (*sage.algebras.lie_algebras.subalgebra.LieSubalgebra_finite_dimensional_with_basis method*), 1254
`left_ideal()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionOrder method*), 649
`left_matrix()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement method*), 142
`left_order()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational method*), 632
`left_table()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method*), 129
`length_orbit()` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep method*), 923
`letterplace_polynomial()` (*sage.algebras.letterplace.free_algebra_element_letterplace.FreeAlgebraElement_letterplace method*), 93
`LetterplaceIdeal` (*class in sage.algebras.letterplace.letterplace_ideal*), 99
`level()` (*sage.algebras.q_system.QSystem method*), 1022
`level()` (*sage.algebras.yangian.YangianLevel method*), 799
`lie_algebra()` (*sage.algebras.lie_algebras.bgg_dual_module.BGGDualModule method*), 1081
`lie_algebra()` (*sage.algebras.lie_algebras.bgg_dual_module.SimpleModule method*), 1086
`lie_algebra()` (*sage.algebras.lie_algebras.onsager.QuantumOnsagerAlgebra method*), 1208
`lie_algebra()` (*sage.algebras.lie_algebras.poincare_birkhoff_witt.PoincareBirkhoffWittBasis method*), 1216
`lie_algebra()` (*sage.algebras.lie_algebras.representation.Representation_abstract method*), 1238
`lie_algebra()` (*sage.algebras.lie_algebras.verma_module.VermaModule method*), 1269
`lie_algebra_generators()` (*sage.algebras.lie_algebras.affine_lie_algebra.AffineLieAlgebra method*), 1063
`lie_algebra_generators()` (*sage.algebras.lie_algebras.classical_lie_algebra.LieAlgebraChevalleyBasis method*), 1109
`lie_algebra_generators()` (*sage.algebras.lie_algebras.free_lie_algebra.FreeLieAlgebra method*), 1138
`lie_algebra_generators()` (*sage.algebras.lie_algebras.heisenberg.HeisenbergAlgebra_fd method*), 1145
`lie_algebra_generators()` (*sage.algebras.lie_algebras.heisenberg.InfiniteHeisenbergAlgebra method*), 1151
`lie_algebra_generators()` (*sage.algebras.lie_algebras.lie_algebra.LieAlgebraFromAssociative method*), 1168
`lie_algebra_generators()` (*sage.algebras.lie_algebras.lie_algebra.LieAlgebraWithGenerators method*), 1171
`lie_algebra_generators()` (*sage.algebras.lie_algebras.onsager.OnsagerAlgebra method*), 1198
`lie_algebra_generators()` (*sage.algebras.lie_algebras.onsager.OnsagerAlgebraACE method*), 1203
`lie_algebra_generators()` (*sage.algebras.lie_algebras.subalgebra.LieSubalgebra_finite_dimensional_with_basis method*), 1254
`lie_algebra_generators()` (*sage.algebras.lie_algebras.virasoro.LieAlgebraRegularVectorFields*

- method*), 1283
- `lie_algebra_generators()` (*sage.algebras.lie_algebras.virasoro.VirasoroAlgebra method*), 1289
- `lie_algebra_generators()` (*sage.algebras.lie_algebras.virasoro.WittLieAlgebra_charp method*), 1291
- `lie_conformal_algebra_generators()` (*sage.algebras.lie_conformal_algebras.freely_generated_lie_conformal_algebra.FreelyGeneratedLieConformalAlgebra method*), 1317
- `lie_polynomial()` (*sage.algebras.free_algebra.FreeAlgebra_generic method*), 70
- `LieAlgebra` (class in *sage.algebras.lie_algebras.lie_algebra*), 1152
- `LieAlgebraChevalleyBasis` (class in *sage.algebras.lie_algebras.classical_lie_algebra*), 1105
- `LieAlgebraChevalleyBasis_simply_laced` (class in *sage.algebras.lie_algebras.classical_lie_algebra*), 1109
- `LieAlgebraElement` (class in *sage.algebras.lie_algebras.lie_algebra_element*), 1174
- `LieAlgebraElementWrapper` (class in *sage.algebras.lie_algebras.lie_algebra_element*), 1175
- `LieAlgebraFromAssociative` (class in *sage.algebras.lie_algebras.lie_algebra*), 1163
- `LieAlgebraFromAssociative.Element` (class in *sage.algebras.lie_algebras.lie_algebra*), 1166
- `LieAlgebraHomomorphism_in_gens` (class in *sage.algebras.lie_algebras.morphism*), 1183
- `LieAlgebraHomset` (class in *sage.algebras.lie_algebras.morphism*), 1186
- `LieAlgebraMatrixWrapper` (class in *sage.algebras.lie_algebras.lie_algebra_element*), 1175
- `LieAlgebraMorphism_from_generators` (class in *sage.algebras.lie_algebras.morphism*), 1187
- `LieAlgebraRegularVectorFields` (class in *sage.algebras.lie_algebras.virasoro*), 1282
- `LieAlgebraRegularVectorFields.Element` (class in *sage.algebras.lie_algebras.virasoro*), 1282
- `LieAlgebraWithGenerators` (class in *sage.algebras.lie_algebras.lie_algebra*), 1170
- `LieAlgebraWithStructureCoefficients` (class in *sage.algebras.lie_algebras.structure_coefficients*), 1240
- `LieAlgebraWithStructureCoefficients.Element` (class in *sage.algebras.lie_algebras.structure_coefficients*), 1241
- `LieBracket` (class in *sage.algebras.lie_algebras.lie_algebra_element*), 1175
- `LieConformalAlgebra` (class in *sage.algebras.lie_conformal_algebras.lie_conformal_algebra*), 1295
- `LieConformalAlgebraWithBasis` (class in *sage.algebras.lie_conformal_algebras.lie_conformal_algebra_with_basis*), 1320
- `LieConformalAlgebraWithStructureCoefficients` (class in *sage.algebras.lie_conformal_algebras.lie_conformal_algebra_with_structure_coefs*), 1321
- `LieGenerator` (class in *sage.algebras.lie_algebras.lie_algebra_element*), 1176
- `LieObject` (class in *sage.algebras.lie_algebras.lie_algebra_element*), 1177
- `LieQuotient_finite_dimensional_with_basis` (class in *sage.algebras.lie_algebras.quotient*), 1221
- `LieSubalgebra_finite_dimensional_with_basis` (class in *sage.algebras.lie_algebras.subalgebra*), 1244
- `LieSubalgebra_finite_dimensional_with_basis.Element` (class in *sage.algebras.lie_algebras.subalgebra*), 1248
- `LieSubalgebraElementWrapper` (class in *sage.algebras.lie_algebras.lie_algebra_element*), 1177
- `lift()` (*sage.algebras.cluster_algebra.ClusterAlgebra method*), 319
- `lift()` (*sage.algebras.lie_algebras.bgg_dual_module.SimpleModule method*), 1086
- `lift()` (*sage.algebras.lie_algebras.center_uea.CenterUEA method*), 1098
- `lift()` (*sage.algebras.lie_algebras.lie_algebra_element.FreeLieAlgebraElement method*), 1173
- `lift()` (*sage.algebras.lie_algebras.lie_algebra_element.LieAlgebraElement method*), 1174
- `lift()` (*sage.algebras.lie_algebras.lie_algebra_element.LieBracket method*), 1175
- `lift()` (*sage.algebras.lie_algebras.lie_algebra_element.LieGenerator method*), 1176
- `lift()` (*sage.algebras.lie_algebras.lie_algebra_element.StructureCoefficientsElement method*), 1179
- `lift()` (*sage.algebras.lie_algebras.quotient.LieQuotient_finite_dimensional_with_basis method*), 1226
- `lift()` (*sage.algebras.lie_algebras.subalgebra.LieSubalgebra_finite_dimensional_with_basis method*), 1255
- `lift()` (*sage.algebras.quantum_groups.quantum_group_gap.HighestWeightSubmodule method*), 543
- `lift()` (*sage.algebras.quantum_groups.quantum_group_gap.LowerHalfQuantumGroup method*), 549
- `lift()` (*sage.combinat.diagram_algebras.SubPartitionAlgebra method*), 238
- `lift()` (*sage.combinat.free_prelie_algebra.FreePreLieAlgebra.Element method*), 1357
- `lift()` (*sage.combinat.posets.incidence_algebras.ReducedIncidenceAlgebra method*), 467

`lift()` (*sage.combinat.posets.incidence_algebras.ReducedIncidenceAlgebra.Element* method), 465
`lift_associative()` (*sage.algebras.lie_algebras.lie_algebra.LieAlgebraFromAssociative.Element* method), 1166
`lift_isometry()` (*sage.algebras.clifford_algebra.CliffordAlgebra* method), 264
`lift_module_morphism()` (*sage.algebras.clifford_algebra.CliffordAlgebra* method), 265
`lift_morphism()` (*sage.algebras.clifford_algebra.ExteriorAlgebra* method), 278
`lift_on_basis()` (*sage.algebras.lie_algebras.center_uea.CenterIndices* method), 1092
`lifted_bilinear_form()` (*sage.algebras.clifford_algebra.ExteriorAlgebra* method), 280
`lifting_map()` (*sage.algebras.splitting_algebra.SplittingAlgebra* method), 1038
`LiftMorphismToAssociative` (class in *sage.algebras.lie_algebras.lie_algebra*), 1171
`list()` (*sage.algebras.lie_algebras.lie_algebra_element.FreeLieAlgebraElement* method), 1174
`list()` (*sage.algebras.weyl_algebra.DifferentialWeylAlgebraElement* method), 782
`lm()` (*sage.algebras.letterplace.free_algebra_element_letterplace.FreeAlgebraElement_letterplace* method), 94
`lm_divides()` (*sage.algebras.letterplace.free_algebra_element_letterplace.FreeAlgebraElement_letterplace* method), 94
`load_fvars()` (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 415
`loop_representation()` (*sage.algebras.askey_wilson.AskeyWilsonAlgebra* method), 161
`lower_bound()` (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 319
`lower_central_series()` (*sage.algebras.lie_algebras.affine_lie_algebra.AffineLieAlgebra* method), 1064
`lower_global_crystal` (*sage.algebras.quantum_groups.fock_space.FockSpace* attribute), 33
`lower_global_crystal` (*sage.algebras.quantum_groups.fock_space.FockSpaceTruncated* attribute), 44
`lower_half()` (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroup* method), 562
`LowerHalfQuantumGroup` (class in *sage.algebras.quantum_groups.quantum_group_gap*), 544
`LowerHalfQuantumGroup.Element` (class in *sage.algebras.quantum_groups.quantum_group_gap*), 544
`lt()` (*sage.algebras.letterplace.free_algebra_element_letterplace.FreeAlgebraElement_letterplace* method), 95
`LyndonBracket` (class in *sage.algebras.lie_alge-*

bras.lie_algebra_element), 1178

M

`m()` (*sage.algebras.quantum_matrix_coordinate_algebra.QuantumMatrixCoordinateAlgebra* method), 576
`magnetic_field_direction()` (*sage.combinat.digraph_algebras.PottsRepresentation* method), 232
`make_FvarsHandler()` (in module *sage.algebras.fusion_rings.shm_managers*), 447
`make_KSHandler()` (in module *sage.algebras.fusion_rings.shm_managers*), 447
`make_mono_admissible()` (in module *sage.algebras.steenrod.steenrod_algebra_mult*), 767
`markov_trace_version()` (*sage.algebras.hecke_algebras.cubic_hecke_base_ring.CubicHeckeExtensionRing* method), 903
`markov_trace_version()` (*sage.algebras.hecke_algebras.cubic_hecke_base_ring.CubicHeckeRingOfDefinition* method), 909
`matrix()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement* method), 142
`matrix()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism.FiniteDimensionalAlgebraMorphism* method), 149
`matrix()` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeElement* method), 890
`matrix()` (*sage.algebras.lie_algebras.lie_algebra.MatrixLieAlgebraFromAssociative.Element* method), 1172
`matrix_action()` (*sage.algebras.free_algebra_quotient.FreeAlgebraQuotient* method), 110
`MatrixCompactRealForm` (class in *sage.algebras.lie_algebras.classical_lie_algebra*), 1111
`MatrixCompactRealForm.Element` (class in *sage.algebras.lie_algebras.classical_lie_algebra*), 1112
`MatrixLieAlgebraFromAssociative` (class in *sage.algebras.lie_algebras.lie_algebra*), 1172
`MatrixLieAlgebraFromAssociative.Element` (class in *sage.algebras.lie_algebras.lie_algebra*), 1172
`max_degree()` (*sage.algebras.finite_gca.FiniteGCAAlgebra* method), 937
`max_len()` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeElement* method), 893
`maximal_degree()` (*sage.algebras.finite_gca.FiniteGCAAlgebra* method), 937
`maximal_ideal()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra* method), 130

`maximal_ideals()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra* method), 132
`maximal_order()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab* method), 602
`maxord_solve_aux_eq()` (in module *sage.algebras.quatalg.quaternion_algebra*), 656
`may_weight()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element* method), 698
`merge()` (*sage.algebras.free_algebra.AssociativeFunctor* method), 62
`merge()` (*sage.algebras.free_zinbiel_algebra.ZinbielFunctor* method), 1392
`merge()` (*sage.combinat.free_dendriform_algebra.DendriformFunctor* method), 1342
`merge()` (*sage.combinat.free_prelie_algebra.PreLieFunctor* method), 1368
`milnor()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic* method), 723
`milnor()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element* method), 700
`milnor_basis()` (in module *sage.algebras.steenrod.steenrod_algebra_bases*), 737
`milnor_mono_to_string()` (in module *sage.algebras.steenrod.steenrod_algebra_misc*), 753
`milnor_multiplication()` (in module *sage.algebras.steenrod.steenrod_algebra_mult*), 768
`milnor_multiplication_odd()` (in module *sage.algebras.steenrod.steenrod_algebra_mult*), 769
`minimal_element()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational* method), 632
`minimal_model()` (*sage.algebras.commutative_dga.DifferentialGCAAlgebra* method), 962
`minimal_polynomial()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement* method), 143
`MinusculeRepresentation` (class in *sage.algebras.quantum_groups.representations*), 52
`mirror_image()` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* method), 878
`mirror_involution()` (*sage.algebras.hecke_algebras.cubic_hecke_base_ring.CubicHeckeExtensionRing* method), 904
`mirror_involution()` (*sage.algebras.hecke_algebras.cubic_hecke_base_ring.CubicHeckeRingOfDefinition* method), 910
`mirror_isomorphism()` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* method), 881
`modp_splitting_data()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab* method), 605
`modp_splitting_map()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab* method), 607
`module`
 sage.algebras.affine_nil_temperley_lieb, 151
 sage.algebras.askey_wilson, 155
 sage.algebras.associated_graded, 1007
 sage.algebras.catalog, 1
 sage.algebras.cellular_basis, 1011
 sage.algebras.clifford_algebra, 257
 sage.algebras.cluster_algebra, 295
 sage.algebras.commutative_dga, 939
 sage.algebras.down_up_algebra, 353
 sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra, 121
 sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element, 138
 sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_ideal, 145
 sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism, 146
 sage.algebras.finite_gca, 933
 sage.algebras.free_algebra, 59
 sage.algebras.free_algebra_element, 80
 sage.algebras.free_algebra_quotient, 106
 sage.algebras.free_algebra_quotient_element, 112
 sage.algebras.free_zinbiel_algebra, 1385
 sage.algebras.fusion_rings.f_matrix, 391
 sage.algebras.fusion_rings.fast_parallel_fmats_methods, 433
 sage.algebras.fusion_rings.fast_parallel_fusion_ring_braid_repn, 434
 sage.algebras.fusion_rings.fusion_double, 419
 sage.algebras.fusion_rings.fusion_ring, 365
 sage.algebras.fusion_rings.poly_tup_engine, 434
 sage.algebras.fusion_rings.shm_managers, 440
 sage.algebras.group_algebra, 469
 sage.algebras.hall_algebra, 447
 sage.algebras.hecke_algebras.ariki_koike_algebra, 803
 sage.algebras.hecke_algebras.ariki_koike_specht_modules,

- 816
- sage.algebras.hecke_algebras.cubic_hecke_algebra, 862
 - sage.algebras.hecke_algebras.cubic_hecke_base_ring, 895
 - sage.algebras.hecke_algebras.cubic_hecke_matrix_rep, 917
 - sage.algebras.iwahori_hecke_algebra, 821
 - sage.algebras.jordan_algebra, 1323
 - sage.algebras.letterplace.free_algebra_element_letterplace, 91
 - sage.algebras.letterplace.free_algebra_letterplace, 82
 - sage.algebras.letterplace.letterplace_ideal, 98
 - sage.algebras.lie_algebras.abelian, 1055
 - sage.algebras.lie_algebras.affine_lie_algebra, 1058
 - sage.algebras.lie_algebras.bch, 1072
 - sage.algebras.lie_algebras.bgg_dual_module, 1077
 - sage.algebras.lie_algebras.bgg_resolution, 1074
 - sage.algebras.lie_algebras.center_uea, 1092
 - sage.algebras.lie_algebras.classical_lie_algebra, 1100
 - sage.algebras.lie_algebras.examples, 1121
 - sage.algebras.lie_algebras.free_lie_algebra, 1133
 - sage.algebras.lie_algebras.heisenberg, 1141
 - sage.algebras.lie_algebras.lie_algebra, 1151
 - sage.algebras.lie_algebras.lie_algebra_element, 1173
 - sage.algebras.lie_algebras.morphism, 1183
 - sage.algebras.lie_algebras.nilpotent_lie_algebra, 1191
 - sage.algebras.lie_algebras.onsager, 1195
 - sage.algebras.lie_algebras.poincare_birkhoff_witt, 1211
 - sage.algebras.lie_algebras.quotient, 1221
 - sage.algebras.lie_algebras.rank_two_heisenberg_virasoro, 1228
 - sage.algebras.lie_algebras.representation, 1231
 - sage.algebras.lie_algebras.structure_coefficients, 1240
 - sage.algebras.lie_algebras.subalgebra, 1244
 - sage.algebras.lie_algebras.symplectic_derivation, 1258
 - sage.algebras.lie_algebras.verma_module, 1261
 - sage.algebras.lie_algebras.virasoro, 1279
 - sage.algebras.lie_conformal_algebras.abelian_lie_conformal_algebra, 1299
 - sage.algebras.lie_conformal_algebras.affine_lie_conformal_algebra, 1300
 - sage.algebras.lie_conformal_algebras.bosonic_ghosts_lie_conformal_algebra, 1302
 - sage.algebras.lie_conformal_algebras.examples, 1297
 - sage.algebras.lie_conformal_algebras.fermionic_ghosts_lie_conformal_algebra, 1303
 - sage.algebras.lie_conformal_algebras.finitely_freely_generated_lca, 1315
 - sage.algebras.lie_conformal_algebras.free_bosons_lie_conformal_algebra, 1305
 - sage.algebras.lie_conformal_algebras.free_fermions_lie_conformal_algebra, 1307
 - sage.algebras.lie_conformal_algebras.freely_generated_lie_conformal_algebra, 1316
 - sage.algebras.lie_conformal_algebras.graded_lie_conformal_algebra, 1318
 - sage.algebras.lie_conformal_algebras.lie_conformal_algebra, 1292
 - sage.algebras.lie_conformal_algebras.lie_conformal_algebra_element, 1298
 - sage.algebras.lie_conformal_algebras.lie_conformal_algebra_with_basis, 1320
 - sage.algebras.lie_conformal_algebras.lie_conformal_algebra_with_structure_coefs, 1321
 - sage.algebras.lie_conformal_algebras.n2_lie_conformal_algebra, 1309
 - sage.algebras.lie_conformal_algebras.neveu_schwarz_lie_conformal_algebra, 1310
 - sage.algebras.lie_conformal_algebras.virasoro_lie_conformal_algebra, 1315

- gebra, 1311
- sage.algebras.lie_conformal_algebra.weyl_lie_conformal_algebra, 1312
- sage.algebras.nil_coxeter_algebra, 855
- sage.algebras.octonion_algebra, 1043
- sage.algebras.orlik_solomon, 496
- sage.algebras.orlik_terao, 487
- sage.algebras.q_commuting_polynomials, 1023
- sage.algebras.q_system, 1017
- sage.algebras.quantum_clifford, 526
- sage.algebras.quantum_groups.ace_quantum_onsager, 3
- sage.algebras.quantum_groups.fock_space, 13
- sage.algebras.quantum_groups.q_numbers, 44
- sage.algebras.quantum_groups.quantum_group_gap, 539
- sage.algebras.quantum_groups.representations, 47
- sage.algebras.quantum_matrix_coordinate_algebra, 570
- sage.algebras.quantum_oscillator, 581
- sage.algebras.quatalg.quaternion_algebra, 589
- sage.algebras.rational_cherednik_algebra, 658
- sage.algebras.schur_algebra, 662
- sage.algebras.shuffle_algebra, 1370
- sage.algebras.splitting_algebra, 1034
- sage.algebras.steenrod.steenrod_algebra, 669
- sage.algebras.steenrod.steenrod_algebra_bases, 729
- sage.algebras.steenrod.steenrod_algebra_misc, 745
- sage.algebras.steenrod.steenrod_algebra_mult, 762
- sage.algebras.tensor_algebra, 114
- sage.algebras.weyl_algebra, 773
- sage.algebras.yangian, 787
- sage.algebras.yokonuma_hecke_algebra, 857
- sage.combinat.descent_algebra, 338
- sage.combinat.diagram_algebras, 173
- sage.combinat.free_dendriform_algebra, 1341
- sage.combinat.free_prelie_algebra, 1354
- sage.combinat.grossman_larson_algebras, 471
- sage.combinat.partition_algebra, 509
- sage.combinat.posets.incidence_algebra, 460
- sage.combinat.posets.moebius_algebra, 480
- module() (*sage.algebras.free_algebra_quotient.FreeAlgebraQuotient method*), 110
- module() (*sage.algebras.lie_algebras.structure_coefficients.LieAlgebraWithStructureCoefficients method*), 1242
- module() (*sage.algebras.lie_algebras.subalgebra.LieSubalgebraFiniteDimensionalWithBasis method*), 1256
- module_generator() (*sage.algebras.quantum_groups.representations.CyclicRepresentation method*), 52
- module_order() (*sage.algebras.lie_algebras.bgg_resolution.BGGResolution method*), 1075
- ModulePrinting (*class in sage.algebras.lie_algebras.verma_module*), 1261
- moebius() (*sage.combinat.posets.incidence_algebras.IncidenceAlgebra method*), 462
- moebius() (*sage.combinat.posets.incidence_algebras.ReducedIncidenceAlgebra method*), 467
- MoebiusAlgebra (*class in sage.combinat.posets.moebius_algebra*), 480
- MoebiusAlgebraBases (*class in sage.combinat.posets.moebius_algebra*), 483
- MoebiusAlgebraBases.ElementMethods (*class in sage.combinat.posets.moebius_algebra*), 483
- MoebiusAlgebraBases.ParentMethods (*class in sage.combinat.posets.moebius_algebra*), 483
- MoebiusAlgebra.E (*class in sage.combinat.posets.moebius_algebra*), 480
- MoebiusAlgebra.I (*class in sage.combinat.posets.moebius_algebra*), 481
- monoid() (*sage.algebras.free_algebra_quotient.FreeAlgebraQuotient method*), 110
- monoid() (*sage.algebras.free_algebra.FreeAlgebraGeneric method*), 71
- monomial() (*sage.algebras.lie_algebras.affine_lie_algebra.AffineLieAlgebra method*), 1064
- monomial() (*sage.algebras.lie_algebras.classical_lie_algebra.gl method*), 1117
- monomial() (*sage.algebras.lie_algebras.classical_lie_algebra.MatrixCompactRealForm method*), 1113
- monomial() (*sage.algebras.lie_algebras.free_lie_algebra.FreeLieBasisAbstract method*), 1140
- monomial() (*sage.algebras.lie_algebras.lie_algebra.LieAlgebra method*), 1162
- monomial() (*sage.algebras.lie_algebras.lie_algebra.LieAlgebraFromAssociative method*), 1169
- monomial() (*sage.algebras.lie_algebras.structure_coefficients.LieAlgebraWithStructureCoefficients method*), 1242

- `monomial_basis()` (*sage.algebras.free_algebra_quotient.FreeAlgebraQuotient* method), 111
`monomial_basis()` (*sage.algebras.hall_algebra.HallAlgebra* method), 452
`monomial_coefficients()` (*sage.algebras.commutative_dga.GCAlgebra.Element* method), 984
`monomial_coefficients()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement* method), 143
`monomial_coefficients()` (*sage.algebras.jordan_algebra.ExceptionalJordanAlgebra.Element* method), 1325
`monomial_coefficients()` (*sage.algebras.jordan_algebra.JordanAlgebraSymmetricBilinear.Element* method), 1336
`monomial_coefficients()` (*sage.algebras.jordan_algebra.SpecialJordanAlgebra.Element* method), 1338
`monomial_coefficients()` (*sage.algebras.lie_algebras.classical_lie_algebra.gl.Element* method), 1116
`monomial_coefficients()` (*sage.algebras.lie_algebras.classical_lie_algebra.MatrixCompactRealForm.Element* method), 1112
`monomial_coefficients()` (*sage.algebras.lie_algebras.heisenberg.HeisenbergAlgebra_matrix.Element* method), 1148
`monomial_coefficients()` (*sage.algebras.lie_algebras.lie_algebra_element.LieSubalgebraElementWrapper* method), 1177
`monomial_coefficients()` (*sage.algebras.lie_algebras.lie_algebra_element.StructureCoefficientsElement* method), 1179
`monomial_coefficients()` (*sage.algebras.lie_algebras.lie_algebra_element.UntwistedAffineLieAlgebraElement* method), 1182
`monomial_coefficients()` (*sage.algebras.lie_algebras.lie_algebra.LieAlgebraFromAssociative.Element* method), 1166
`monomial_coefficients()` (*sage.algebras.octonion_algebra.Octonion_generic* method), 1053
`monomial_coefficients()` (*sage.algebras.quantum_groups.quantum_group_gap.LowerHalfQuantumGroup.Element* method), 545
`monomial_coefficients()` (*sage.algebras.quantum_groups.quantum_group_gap.QuaGroupRepresentationElement* method), 552
`monomial_coefficients()` (*sage.algebras.splitting_algebra.SplittingAlgebraElement* method), 1041
`multicharge()` (*sage.algebras.quantum_groups.fock_space.FockSpace* method), 34
`multicharge()` (*sage.algebras.quantum_groups.fock_space.FockSpaceBases.ParentMethods* method), 35
`multinomial()` (*in module sage.algebras.steenrod.steenrod_algebra_mult*), 771
`multinomial_odd()` (*in module sage.algebras.steenrod.steenrod_algebra_mult*), 772
`multiply_by_conjugate()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational* method), 633
`mutate()` (*sage.algebras.cluster_algebra.ClusterAlgebraSeed* method), 334
`mutate_initial()` (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 320
- ## N
- `n()` (*sage.algebras.lie_algebras.heisenberg.HeisenbergAlgebra_fd* method), 1145
`n()` (*sage.algebras.quantum_matrix_coordinate_algebra.QuantumMatrixCoordinateAlgebra_abstract* method), 578
`N2LieConformalAlgebra` (*class in sage.algebras.lie_conformal_algebras.n2_lie_conformal_algebra*), 1309
`N_ijk()` (*sage.algebras.fusion_rings.fusion_double.FusionDouble* method), 425
`N_ijk()` (*sage.algebras.fusion_rings.fusion_ring.FusionRing* method), 375
`nap_product()` (*sage.combinat.free_prelie_algebra.FreePreLieAlgebra* method), 1363
`nap_product_on_basis()` (*sage.combinat.free_prelie_algebra.FreePreLieAlgebra* method), 1364
`natural` (*sage.algebras.quantum_groups.fock_space.FockSpace* attribute), 34
`natural` (*sage.algebras.quantum_groups.fock_space.FockSpaceTruncated* attribute), 44
`natural` (*sage.combinat.posets.moebius_algebra.MoebiusAlgebra* attribute), 483
`natural` (*sage.combinat.posets.moebius_algebra.QuantumMoebiusAlgebra* attribute), 487
`natural_map()` (*sage.algebras.lie_algebras.verma_module.VermaModuleHomset* method), 1273
`neg` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.GenSign* attribute), 930
`NeveuSchwarzLieConformalAlgebra` (*class in sage.algebras.lie_conformal_algebras.neveu_schwarz_lie_conformal_algebra*), 1310
`ngens()` (*sage.algebras.clifford_algebra.CliffordAlgebra* method), 269
`ngens()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra* method), 132

- `ngens()` (*sage.algebras.finite_gca.FiniteGCAAlgebra* method), 937
`ngens()` (*sage.algebras.free_algebra_quotient.FreeAlgebraQuotient* method), 111
`ngens()` (*sage.algebras.free_algebra.FreeAlgebra_generic* method), 71
`ngens()` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* method), 882
`ngens()` (*sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace* method), 89
`ngens()` (*sage.algebras.lie_conformal_algebras.finitely_freely_generated_lca.FinitelyFreelyGeneratedLCA* method), 1316
`ngens()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract* method), 619
`ngens()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionOrder* method), 650
`ngens()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic* method), 723
`ngens()` (*sage.algebras.weyl_algebra.DifferentialWeylAlgebra* method), 777
`NilCoxeterAlgebra` (class in *sage.algebras.nil_coxeter_algebra*), 855
`NilpotentLieAlgebra_dense` (class in *sage.algebras.lie_algebras.nilpotent_lie_algebra*), 1194
`Nk_ij()` (*sage.algebras.fusion_rings.fusion_double.FusionDouble* method), 425
`Nk_ij()` (*sage.algebras.fusion_rings.fusion_ring.FusionRing* method), 376
`norm()` (*sage.algebras.jordan_algebra.JordanAlgebraSymmetricBilinear.Element* method), 1336
`norm()` (*sage.algebras.octonion_algebra.Octonion* method), 1043
`norm()` (*sage.algebras.octonion_algebra.Octonion_generic* method), 1053
`norm()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational* method), 634
`normal_form()` (*sage.algebras.letterplace.free_algebra_element_letterplace.FreeAlgebraElement_letterplace* method), 95
`normalize_basis_at_p()` (in module *sage.algebras.quatalg.quaternion_algebra*), 656
`normalize_names_markov()` (in module *sage.algebras.hecke_algebras.cubic_hecke_base_ring*), 916
`normalize_profile()` (in module *sage.algebras.steenrod.steenrod_algebra_misc*), 754
`normalized_laurent_polynomial()` (in module *sage.algebras.iwahori_hecke_algebra*), 853
`number_gens()` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* method), 923
`number_of_colors()` (*sage.combinat.diagram_algebras.PottsRepresentation* method), 232
`number_of_factors()` (*sage.combinat.diagram_algebras.PottsRepresentation* method), 233
`number_of_representations()` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.RepresentationType* method), 931
`numerical_invariants()` (*sage.algebras.commutative_dga.DifferentialGCAAlgebra* method), 967
- ## O
- `Octonion` (class in *sage.algebras.octonion_algebra*), 1043
`Octonion_generic` (class in *sage.algebras.octonion_algebra*), 1049
`OctonionAlgebra` (class in *sage.algebras.octonion_algebra*), 1044
`omega()` (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroup.Element* method), 556
`one()` (*sage.algebras.cellular_basis.CellularBasis* method), 1016
`one()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra* method), 133
`one()` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.CubicHeckeMatrixSpace* method), 927
`one()` (*sage.algebras.jordan_algebra.ExceptionalJordanAlgebra* method), 1328
`one()` (*sage.algebras.jordan_algebra.JordanAlgebraSymmetricBilinear* method), 1338
`one()` (*sage.algebras.jordan_algebra.SpecialJordanAlgebra* method), 1340
`one()` (*sage.algebras.quantum_groups.quantum_group_gap.LowerHalfQuantumGroup* method), 550
`one()` (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroup* method), 563
`one()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionOrder* method), 650
`one()` (*sage.algebras.schur_algebra.SchurAlgebra* method), 664
`one()` (*sage.algebras.weyl_algebra.DifferentialWeylAlgebra* method), 778
`one()` (*sage.combinat.descent_algebra.DescentAlgebra.I* method), 348
`one()` (*sage.combinat.diagram_algebras.OrbitBasis* method), 194
`one()` (*sage.combinat.posets.incidence_algebras.IncidenceAlgebra* method), 463
`one()` (*sage.combinat.posets.moebius_algebra.MoebiusAlgebraBases.ParentMethods* method), 483

- `one()` (*sage.combinat.posets.moebius_algebra.MoebiusAlgebra.E method*), 480
- `one()` (*sage.combinat.posets.moebius_algebra.MoebiusAlgebra.I method*), 481
- `one()` (*sage.combinat.posets.moebius_algebra.QuantumMoebiusAlgebra.E method*), 485
- `one_basis()` (*sage.algebras.affine_nil_temperley_lieb.AffineNilTemperleyLiebTypeA method*), 153
- `one_basis()` (*sage.algebras.askey_wilson.AskeyWilsonAlgebra method*), 164
- `one_basis()` (*sage.algebras.associated_graded.AssociatedGradedAlgebra method*), 1010
- `one_basis()` (*sage.algebras.clifford_algebra.CliffordAlgebra method*), 269
- `one_basis()` (*sage.algebras.down_up_algebra.DownUpAlgebra method*), 357
- `one_basis()` (*sage.algebras.finite_gca.FiniteGCAAlgebra method*), 938
- `one_basis()` (*sage.algebras.free_algebra.FreeAlgebra_generic method*), 71
- `one_basis()` (*sage.algebras.free_algebra.PBWBasisOfFreeAlgebra method*), 79
- `one_basis()` (*sage.algebras.fusion_rings.fusion_double.FusionDouble method*), 429
- `one_basis()` (*sage.algebras.hall_algebra.HallAlgebra method*), 453
- `one_basis()` (*sage.algebras.hall_algebra.HallAlgebraMonomials method*), 458
- `one_basis()` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra method*), 882
- `one_basis()` (*sage.algebras.lie_algebras.center_uea.CenterUEA method*), 1098
- `one_basis()` (*sage.algebras.lie_algebras.onsager.QuantumOnsagerAlgebra method*), 1208
- `one_basis()` (*sage.algebras.lie_algebras.poincare_birkhoff_witt.PoincareBirkhoffWittBasis method*), 1216
- `one_basis()` (*sage.algebras.octonion_algebra.OctonionAlgebra method*), 1048
- `one_basis()` (*sage.algebras.orlik_solomon.OrlikSolomonAlgebra method*), 501
- `one_basis()` (*sage.algebras.orlik_terao.OrlikTeraoAlgebra method*), 490
- `one_basis()` (*sage.algebras.q_commuting_polynomials.qCommutingLaurentPolynomials method*), 1025
- `one_basis()` (*sage.algebras.q_commuting_polynomials.qCommutingPolynomials method*), 1030
- `one_basis()` (*sage.algebras.q_system.QSystem method*), 1022
- `one_basis()` (*sage.algebras.quantum_clifford.QuantumCliffordAlgebra method*), 531
- `one_basis()` (*sage.algebras.quantum_groups.ace_quantum_onsager.ACEQuantumOnsagerAlgebra method*), 7
- `one_basis()` (*sage.algebras.quantum_matrix_coordinate_algebra.QuantumMatrixCoordinateAlgebra_abstract method*), 579
- `one_basis()` (*sage.algebras.quantum_oscillator.QuantumOscillatorAlgebra method*), 586
- `one_basis()` (*sage.algebras.rational_cherednik_algebra.RationalCherednikAlgebra method*), 660
- `one_basis()` (*sage.algebras.shuffle_algebra.DualPBWBasis method*), 1375
- `one_basis()` (*sage.algebras.shuffle_algebra.ShuffleAlgebra method*), 1382
- `one_basis()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method*), 724
- `one_basis()` (*sage.algebras.tensor_algebra.TensorAlgebra method*), 119
- `one_basis()` (*sage.algebras.yangian.Yangian method*), 796
- `one_basis()` (*sage.algebras.yokonuma_hecke_algebra.YokonumaHeckeAlgebra method*), 861
- `one_basis()` (*sage.combinat.descent_algebra.DescentAlgebra.B method*), 341
- `one_basis()` (*sage.combinat.descent_algebra.DescentAlgebra.D method*), 344
- `one_basis()` (*sage.combinat.descent_algebra.DescentAlgebra.I method*), 348
- `one_basis()` (*sage.combinat.diagram_algebras.UnitDiagramMixin method*), 248
- `one_basis()` (*sage.combinat.free_dendriform_algebra.FreeDendriformAlgebra method*), 1349
- `one_basis()` (*sage.combinat.grossman_larson_algebras.GrossmanLarsonAlgebra method*), 476
- `one_basis()` (*sage.combinat.partition_algebra.PartitionAlgebra_generic method*), 510
- `one_basis()` (*sage.combinat.posets.incidence_algebras.ReducedIncidenceAlgebra method*), 468
- `OnsagerAlgebra` (class in *sage.algebras.lie_algebras.onsager*), 1195
- `OnsagerAlgebraACE` (class in *sage.algebras.lie_algebras.onsager*), 1199
- `options` (*sage.algebras.quantum_groups.fock_space.FockSpace attribute*), 34
- `options` (*sage.algebras.quantum_groups.fock_space.FockSpace.A attribute*), 20
- `options` (*sage.algebras.quantum_groups.fock_space.FockSpace.F attribute*), 27
- `options` (*sage.algebras.quantum_groups.fock_space.FockSpace.G attribute*), 32

- options (sage.algebras.quantum_groups.fock_space.FockSpaceTruncated.A attribute), 41
- options (sage.algebras.quantum_groups.fock_space.FockSpaceTruncated.F attribute), 42
- options (sage.algebras.quantum_groups.fock_space.FockSpaceTruncated.G attribute), 44
- options (sage.algebras.weyl_algebra.DifferentialWeylAlgebra attribute), 778
- options (sage.combinat.diagram_algebras.BrauerAlgebra attribute), 181
- options (sage.combinat.diagram_algebras.BrauerDiagram attribute), 183
- options (sage.combinat.diagram_algebras.BrauerDiagrams attribute), 186
- orbit_basis() (sage.combinat.diagram_algebras.PartitionAlgebra method), 219
- OrbitBasis (class in sage.combinat.diagram_algebras), 192
- OrbitBasis.Element (class in sage.combinat.diagram_algebras), 193
- order() (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract method), 619
- order() (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 724
- order() (sage.combinat.diagram_algebras.AbstractPartitionDiagram method), 177
- order() (sage.combinat.diagram_algebras.DiagramAlgebra method), 187
- order_with_level() (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab method), 608
- orientation_antiinvolution() (sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra method), 882
- OrlikSolomonAlgebra (class in sage.algebras.orlik_solomon), 496
- OrlikSolomonInvariantAlgebra (class in sage.algebras.orlik_solomon), 504
- OrlikTeraoAlgebra (class in sage.algebras.orlik_terao), 487
- OrlikTeraoInvariantAlgebra (class in sage.algebras.orlik_terao), 493
- over() (sage.combinat.free_dendriform_algebra.FreeDendriformAlgebra method), 1349
- ## P
- p() (sage.algebras.lie_algebras.heisenberg.HeisenbergAlgebra_abstract method), 1142
- p() (sage.algebras.lie_algebras.heisenberg.HeisenbergAlgebra_matrix method), 1149
- P() (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 701
- p_exponents() (sage.algebras.lie_algebras.representation.FaithfulRepresentationPBWPosChar method), 1234
- pair_to_graph() (in module sage.combinat.diagram_algebras), 251
- pair_to_graph() (in module sage.combinat.partition_algebra), 523
- parameters() (sage.algebras.lie_algebras.virasoro.ChargelessRepresentation method), 1281
- parent() (sage.algebras.cluster_algebra.ClusterAlgebraSeed method), 335
- partition_algebra() (sage.combinat.diagram_algebras.PottsRepresentation method), 233
- partition_diagrams() (in module sage.combinat.diagram_algebras), 252
- PartitionAlgebra (class in sage.combinat.diagram_algebras), 197
- PartitionAlgebra_ak (class in sage.combinat.partition_algebra), 509
- PartitionAlgebra_bk (class in sage.combinat.partition_algebra), 510
- PartitionAlgebra_generic (class in sage.combinat.partition_algebra), 510
- PartitionAlgebra_pk (class in sage.combinat.partition_algebra), 511
- PartitionAlgebra_prk (class in sage.combinat.partition_algebra), 511
- PartitionAlgebra_rk (class in sage.combinat.partition_algebra), 512
- PartitionAlgebra_sk (class in sage.combinat.partition_algebra), 512
- PartitionAlgebra_tk (class in sage.combinat.partition_algebra), 512
- PartitionAlgebra.Element (class in sage.combinat.diagram_algebras), 205
- PartitionAlgebraElement_ak (class in sage.combinat.partition_algebra), 509
- PartitionAlgebraElement_bk (class in sage.combinat.partition_algebra), 509
- PartitionAlgebraElement_generic (class in sage.combinat.partition_algebra), 509
- PartitionAlgebraElement_pk (class in sage.combinat.partition_algebra), 509
- PartitionAlgebraElement_prk (class in sage.combinat.partition_algebra), 509
- PartitionAlgebraElement_rk (class in sage.combinat.partition_algebra), 509
- PartitionAlgebraElement_sk (class in sage.combinat.partition_algebra), 509
- PartitionAlgebraElement_tk (class in sage.combinat.partition_algebra), 509
- PartitionDiagram (class in sage.combinat.diagram_al-

- gebras*), 223
- PartitionDiagrams* (class in *sage.combinat.diagram_algebras*), 224
- path_from_initial_seed()* (*sage.algebras.cluster_algebra.ClusterAlgebraSeed* method), 335
- pbw_basis()* (*sage.algebras.free_algebra.FreeAlgebra_generic* method), 72
- pbw_basis()* (*sage.algebras.lie_algebras.bgg_dual_module.SimpleModule* method), 1087
- pbw_basis()* (*sage.algebras.lie_algebras.free_lie_algebra.FreeLieAlgebra.Lyndon* method), 1136
- pbw_basis()* (*sage.algebras.lie_algebras.verma_module.VermaModule* method), 1269
- pbw_element()* (*sage.algebras.free_algebra.FreeAlgebra_generic* method), 72
- PBWBasisOfFreeAlgebra* (class in *sage.algebras.free_algebra*), 75
- PBWBasisOfFreeAlgebra.Element* (class in *sage.algebras.free_algebra*), 76
- perm()* (*sage.combinat.diagram_algebras.BrauerDiagram* method), 184
- permutation_automorphism()* (*sage.algebras.askey_wilson.AskeyWilsonAlgebra* method), 164
- pi()* (*sage.algebras.askey_wilson.AskeyWilsonAlgebra* method), 165
- planar_diagrams()* (in module *sage.combinat.diagram_algebras*), 253
- planar_partitions_rec()* (in module *sage.combinat.diagram_algebras*), 254
- PlanarAlgebra* (class in *sage.combinat.diagram_algebras*), 225
- PlanarDiagram* (class in *sage.combinat.diagram_algebras*), 227
- PlanarDiagrams* (class in *sage.combinat.diagram_algebras*), 228
- poincare_birkhoff_witt_basis()* (*sage.algebras.free_algebra.FreeAlgebra_generic* method), 72
- poincare_birkhoff_witt_basis()* (*sage.algebras.lie_algebras.free_lie_algebra.FreeLieAlgebra.Lyndon* method), 1136
- poincare_birkhoff_witt_basis()* (*sage.algebras.lie_algebras.verma_module.VermaModule* method), 1269
- PoincareBirkhoffWittBasis* (class in *sage.algebras.lie_algebras.poincare_birkhoff_witt*), 1211
- PoincareBirkhoffWittBasis.Element* (class in *sage.algebras.lie_algebras.poincare_birkhoff_witt*), 1213
- PoincareBirkhoffWittBasisSemisimpleLieAlgebra* (class in *sage.algebras.lie_algebras.poincare_birkhoff_witt*), 1218
- PoincareBirkhoffWittBasisSemisimpleLieAlgebra.Element* (class in *sage.algebras.lie_algebras.poincare_birkhoff_witt*), 1218
- poly_to_tup()* (in module *sage.algebras.fusion_rings.poly_tup_engine*), 436
- poly_tup_sortkey()* (in module *sage.algebras.fusion_rings.poly_tup_engine*), 437
- polynomial_ring()* (*sage.algebras.weyl_algebra.DifferentialWeylAlgebra* method), 778
- pos* (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.GenSign* attribute), 930
- poset()* (*sage.combinat.posets.incidence_algebras.IncidenceAlgebra* method), 463
- poset()* (*sage.combinat.posets.incidence_algebras.ReducedIncidenceAlgebra* method), 468
- potts_representation()* (*sage.combinat.diagram_algebras.PartitionAlgebra* method), 220
- PottsRepresentation* (class in *sage.combinat.diagram_algebras*), 229
- PottsRepresentation.Element* (class in *sage.combinat.diagram_algebras*), 232
- pre_lie_product()* (*sage.combinat.free_prelie_algebra.FreePreLieAlgebra* method), 1364
- pre_lie_product_on_basis()* (*sage.combinat.free_prelie_algebra.FreePreLieAlgebra* method), 1365
- prec()* (*sage.combinat.free_dendriform_algebra.FreeDendriformAlgebra* method), 1350
- prec_product_on_basis()* (*sage.combinat.free_dendriform_algebra.FreeDendriformAlgebra* method), 1350
- preimage()* (*sage.algebras.lie_algebras.lie_algebra.LiftMorphismToAssociative* method), 1171
- PreLieFunctor* (class in *sage.combinat.free_prelie_algebra*), 1367
- primary_decomposition()* (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra* method), 134
- prime()* (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic* method), 725
- prime()* (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element* method), 700
- primitive_decomposition()* (*sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational* method), 635
- PrincipalClusterAlgebraElement* (class in *sage.algebras.cluster_algebra*), 336
- product()* (*sage.algebras.free_algebra.PBWBasisOfFreeAlgebra* method), 79
- product()* (*sage.algebras.shuffle_algebra.DualPBWBasis* method), 1375
- product_by_generator()* (*sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T* method), 844

`product_by_generator_on_basis()` (*sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T* method), 844
`product_on_basis()` (*sage.algebras.affine_nil_temperley_lieb.AffineNilTemperleyLiebTypeA* method), 153
`product_on_basis()` (*sage.algebras.askey_wilson.AskeyWilsonAlgebra* method), 168
`product_on_basis()` (*sage.algebras.associated_graded.AssociatedGradedAlgebra* method), 1011
`product_on_basis()` (*sage.algebras.cellular_basis.CellularBasis* method), 1016
`product_on_basis()` (*sage.algebras.down_up_algebra.DownUpAlgebra* method), 357
`product_on_basis()` (*sage.algebras.finite_gca.FiniteGCAAlgebra* method), 938
`product_on_basis()` (*sage.algebras.free_algebra.FreeAlgebra_generic* method), 73
`product_on_basis()` (*sage.algebras.fusion_rings.fusion_double.FusionDouble* method), 429
`product_on_basis()` (*sage.algebras.hall_algebra.HallAlgebra* method), 453
`product_on_basis()` (*sage.algebras.hall_algebra.HallAlgebraMonomials* method), 459
`product_on_basis()` (*sage.algebras.hecke_algebras.ariki_koike_algebra.ArikiKoikeAlgebra.LT* method), 809
`product_on_basis()` (*sage.algebras.hecke_algebras.ariki_koike_algebra.ArikiKoikeAlgebra.T* method), 813
`product_on_basis()` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* method), 883
`product_on_basis()` (*sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.Cp* method), 837
`product_on_basis()` (*sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T* method), 845
`product_on_basis()` (*sage.algebras.lie_algebras.center_uea.CenterUEA* method), 1099
`product_on_basis()` (*sage.algebras.lie_algebras.onsager.QuantumOnsagerAlgebra* method), 1209
`product_on_basis()` (*sage.algebras.lie_algebras.poincare_birkhoff_witt.PoincareBirkhoffWittBasis* method), 1216
`product_on_basis()` (*sage.algebras.orlik_solomon.OrlikSolomonAlgebra* method), 501
`product_on_basis()` (*sage.algebras.orlik_terao.OrlikTeraoAlgebra* method), 490
`product_on_basis()` (*sage.algebras.q_commuting_polynomials.qCommutingLaurentPolynomials* method), 1025
`product_on_basis()` (*sage.algebras.q_commuting_polynomials.qCommutingPolynomials* method), 1030
`product_on_basis()` (*sage.algebras.quantum_clifford.QuantumCliffordAlgebraGeneric* method), 534
`product_on_basis()` (*sage.algebras.quantum_clifford.QuantumCliffordAlgebraRootUnity* method), 537
`product_on_basis()` (*sage.algebras.quantum_groups.ace_quantum_onsager.ACEQuantumOnsagerAlgebra* method), 8
`product_on_basis()` (*sage.algebras.quantum_matrix_coordinate_algebra.QuantumGL* method), 573
`product_on_basis()` (*sage.algebras.quantum_matrix_coordinate_algebra.QuantumMatrixCoordinateAlgebra_abstract* method), 579
`product_on_basis()` (*sage.algebras.quantum_oscillator.QuantumOscillatorAlgebra* method), 586
`product_on_basis()` (*sage.algebras.rational_cherednik_algebra.RationalCherednikAlgebra* method), 660
`product_on_basis()` (*sage.algebras.schur_algebra.SchurAlgebra* method), 665
`product_on_basis()` (*sage.algebras.shuffle_algebra.ShuffleAlgebra* method), 1382
`product_on_basis()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic* method), 725
`product_on_basis()` (*sage.algebras.tensor_algebra.TensorAlgebra* method), 119
`product_on_basis()` (*sage.algebras.yangian.GradedYangianNatural* method), 789
`product_on_basis()` (*sage.algebras.yangian.Yangian* method), 797
`product_on_basis()` (*sage.algebras.yokonuma_hecke_algebra.YokonumaHeckeAlgebra* method), 861
`product_on_basis()` (*sage.combinat.descent_algebra.DescentAlgebra.B* method), 341
`product_on_basis()` (*sage.combinat.descent_algebra.DescentAlgebra.D* method), 345
`product_on_basis()` (*sage.combinat.descent_algebra.DescentAlgebra.I* method), 348
`product_on_basis()` (*sage.combinat.diagram_algebras.DiagramBasis* method), 188
`product_on_basis()` (*sage.combinat.diagram_algebras.OrbitBasis* method), 195
`product_on_basis()` (*sage.combinat.free_dendriform_algebra.FreeDendriformAlgebra* method), 1351
`product_on_basis()` (*sage.combinat.free_prelie_algebra.FreePreLieAlgebra* method), 1365
`product_on_basis()` (*sage.combinat.grossman_lar-*

- `son_algebras.GrossmanLarsonAlgebra` method), 476
- `product_on_basis()` (`sage.combinat.partition_algebra.PartitionAlgebra_generic` method), 511
- `product_on_basis()` (`sage.combinat.posets.incidence_algebras.IncidenceAlgebra` method), 463
- `product_on_basis()` (`sage.combinat.posets.moebius_algebra.MoebiusAlgebraBases.ParentMethods` method), 483
- `product_on_basis()` (`sage.combinat.posets.moebius_algebra.MoebiusAlgebra.E` method), 481
- `product_on_basis()` (`sage.combinat.posets.moebius_algebra.MoebiusAlgebra.I` method), 482
- `product_on_basis()` (`sage.combinat.posets.moebius_algebra.QuantumMoebiusAlgebra.E` method), 485
- `product_on_basis_left()` (`sage.algebras.free_zinbiel_algebra.FreeZinbielAlgebra` method), 1390
- `product_on_basis_right()` (`sage.algebras.free_zinbiel_algebra.FreeZinbielAlgebra` method), 1390
- `product_on_gens()` (`sage.algebras.yangian.Yangian` method), 797
- `product_on_gens()` (`sage.algebras.yangian.Yangian-Level` method), 800
- `profile()` (`sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic` method), 726
- `projection()` (`sage.algebras.lie_algebras.onsager.OnsagerAlgebraACE` method), 1203
- `projection_lower_half()` (in module `sage.algebras.quantum_groups.quantum_group_gap`), 569
- `propagating_number()` (in module `sage.combinat.diagram_algebras`), 254
- `propagating_number()` (in module `sage.combinat.partition_algebra`), 525
- `propagating_number()` (`sage.combinat.diagram_algebras.AbstractPartitionDiagram` method), 177
- `PropagatingIdeal` (class in `sage.combinat.diagram_algebras`), 236
- `PropagatingIdeal.Element` (class in `sage.combinat.diagram_algebras`), 237
- `pseudoscalar()` (`sage.algebras.clifford_algebra.CliffordAlgebra` method), 270
- `pst()` (`sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic` method), 727
- `pst_mono_to_string()` (in module `sage.algebras.steenrod.steenrod_algebra_misc`), 758
- `pullback()` (`sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational` method), 635
- `pushforward()` (`sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational` method), 636
- `pwitt()` (in module `sage.algebras.lie_algebras.examples`), 1123
- ## Q
- `q()` (`sage.algebras.askey_wilson.AskeyWilsonAlgebra` method), 169
- `q()` (`sage.algebras.hecke_algebras.ariki_koike_algebra.ArikiKoikeAlgebra` method), 815
- `q()` (`sage.algebras.lie_algebras.heisenberg.HeisenbergAlgebra_abstract` method), 1143
- `q()` (`sage.algebras.lie_algebras.heisenberg.HeisenbergAlgebra_matrix` method), 1149
- `q()` (`sage.algebras.lie_algebras.onsager.QuantumOnsagerAlgebra` method), 1210
- `q()` (`sage.algebras.q_commuting_polynomials.qCommutingPolynomials_generic` method), 1034
- `Q()` (`sage.algebras.q_system.QSystem` method), 1019
- `q()` (`sage.algebras.quantum_clifford.QuantumCliffordAlgebra` method), 531
- `q()` (`sage.algebras.quantum_groups.ace_quantum_onsager.ACEQuantumOnsagerAlgebra` method), 11
- `q()` (`sage.algebras.quantum_groups.fock_space.FockSpace` method), 34
- `q()` (`sage.algebras.quantum_groups.fock_space.FockSpaceBases.ParentMethods` method), 36
- `q()` (`sage.algebras.quantum_groups.quantum_group_gap.QuantumGroup` method), 563
- `q()` (`sage.algebras.quantum_matrix_coordinate_algebra.QuantumMatrixCoordinateAlgebra_abstract` method), 580
- `q()` (`sage.algebras.quantum_oscillator.QuantumOscillatorAlgebra` method), 588
- `Q()` (`sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic` method), 702
- `q1()` (`sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra` method), 848
- `q2()` (`sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra` method), 848
- `q_binomial()` (in module `sage.algebras.quantum_groups.q_numbers`), 44
- `q_dimension()` (`sage.algebras.fusion_rings.fusion_double.FusionDouble.Element` method), 424
- `q_dimension()` (`sage.algebras.fusion_rings.fusion_ring.FusionRing.Element` method), 373
- `Q_exp()` (`sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic` method), 703
- `q_factorial()` (in module `sage.algebras.quantum_groups.q_numbers`), 45
- `q_int()` (in module `sage.algebras.quantum_groups.q_numbers`), 46

- `qCommutingLaurentPolynomials` (class in `sage.algebras.q_commuting_polynomials`), 1023
`qCommutingLaurentPolynomials.Element` (class in `sage.algebras.q_commuting_polynomials`), 1025
`qCommutingPolynomials` (class in `sage.algebras.q_commuting_polynomials`), 1028
`qCommutingPolynomials_generic` (class in `sage.algebras.q_commuting_polynomials`), 1032
`QSystem` (class in `sage.algebras.q_system`), 1017
`QSystem.Element` (class in `sage.algebras.q_system`), 1019
`quadratic_form()` (`sage.algebras.clifford_algebra.CliffordAlgebra` method), 270
`quadratic_form()` (`sage.algebras.octonion_algebra.Octonion` method), 1044
`quadratic_form()` (`sage.algebras.octonion_algebra.Octonion_generic` method), 1054
`quadratic_form()` (`sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational` method), 637
`quadratic_form()` (`sage.algebras.quatalg.quaternion_algebra.QuaternionOrder` method), 650
`QuaGroupModuleElement` (class in `sage.algebras.quantum_groups.quantum_group_gap`), 551
`QuaGroupRepresentationElement` (class in `sage.algebras.quantum_groups.quantum_group_gap`), 552
`quantum_determinant()` (`sage.algebras.quantum_matrix_coordinate_algebra.QuantumMatrixCoordinateAlgebra_abstract` method), 580
`quantum_determinant()` (`sage.algebras.yangian.YangianLevel` method), 800
`quantum_group()` (`sage.algebras.lie_algebras.onsager.OnsagerAlgebra` method), 1198
`quantum_onsager_pbw_generator()` (`sage.algebras.quantum_groups.ace_quantum_onsager.ACEQuantumOnsagerAlgebra` method), 11
`QuantumCliffordAlgebra` (class in `sage.algebras.quantum_clifford`), 526
`QuantumCliffordAlgebraGeneric` (class in `sage.algebras.quantum_clifford`), 532
`QuantumCliffordAlgebraGeneric.Element` (class in `sage.algebras.quantum_clifford`), 532
`QuantumCliffordAlgebraRootUnity` (class in `sage.algebras.quantum_clifford`), 535
`QuantumCliffordAlgebraRootUnity.Element` (class in `sage.algebras.quantum_clifford`), 535
`QuantumGL` (class in `sage.algebras.quantum_matrix_coordinate_algebra`), 570
`QuantumGroup` (class in `sage.algebras.quantum_groups.quantum_group_gap`), 553
`QuantumGroup.Element` (class in `sage.algebras.quantum_groups.quantum_group_gap`), 554
`QuantumGroupHomset` (class in `sage.algebras.quantum_groups.quantum_group_gap`), 564
`QuantumGroupModule` (class in `sage.algebras.quantum_groups.quantum_group_gap`), 564
`QuantumGroupMorphism` (class in `sage.algebras.quantum_groups.quantum_group_gap`), 566
`QuantumGroupRepresentation` (class in `sage.algebras.quantum_groups.representations`), 56
`QuantumMatrixCoordinateAlgebra` (class in `sage.algebras.quantum_matrix_coordinate_algebra`), 573
`QuantumMatrixCoordinateAlgebra_abstract` (class in `sage.algebras.quantum_matrix_coordinate_algebra`), 576
`QuantumMatrixCoordinateAlgebra_abstract.Element` (class in `sage.algebras.quantum_matrix_coordinate_algebra`), 577
`QuantumMoebiusAlgebra` (class in `sage.combinat.posets.moebius_algebra`), 484
`QuantumMoebiusAlgebra.C` (class in `sage.combinat.posets.moebius_algebra`), 484
`QuantumMoebiusAlgebra.E` (class in `sage.combinat.posets.moebius_algebra`), 485
`QuantumMoebiusAlgebra.KL` (class in `sage.combinat.posets.moebius_algebra`), 485
`QuantumOnsagerAlgebra` (class in `sage.algebras.lie_algebras.onsager`), 1204
`QuantumOscillatorAlgebra` (class in `sage.algebras.quantum_oscillator`), 582
`QuantumOscillatorAlgebra.Element` (class in `sage.algebras.quantum_oscillator`), 585
`quaternion_algebra()` (`sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational` method), 637
`quaternion_algebra()` (`sage.algebras.quatalg.quaternion_algebra.QuaternionOrder` method), 651
`quaternion_order()` (`sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab` method), 609
`QuaternionAlgebra_ab` (class in `sage.algebras.quatalg.quaternion_algebra`), 595
`QuaternionAlgebra_abstract` (class in `sage.algebras.quatalg.quaternion_algebra`), 613
`QuaternionAlgebraFactory` (class in `sage.algebras.quatalg.quaternion_algebra`), 589
`QuaternionFractionalIdeal` (class in `sage.algebras.quatalg.quaternion_algebra`), 621
`QuaternionFractionalIdeal_rational` (class in `sage.algebras.quatalg.quaternion_algebra`), 621
`QuaternionOrder` (class in `sage.algebras.quatalg.quaternion_algebra`), 642
`quo()` (`sage.algebras.free_algebra.FreeAlgebra_generic` method), 73
`quotient()` (`sage.algebras.commutative_dga.DifferentialGCAAlgebra` method), 968

- `quotient()` (*sage.algebras.commutative_dga.GCAlgebra* method), 987
- `quotient()` (*sage.algebras.commutative_dga.GCAlgebra_multigraded* method), 997
- `quotient()` (*sage.algebras.free_algebra.FreeAlgebra_generic* method), 74
- `quotient_map()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra* method), 136
- ## R
- `r_matrix()` (*sage.algebras.fusion_rings.fusion_double.FusionDouble* method), 430
- `r_matrix()` (*sage.algebras.fusion_rings.fusion_ring.FusionRing* method), 385
- `R_matrix()` (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroupModule* method), 564
- `ramified_places()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab* method), 609
- `ramified_primes()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab* method), 612
- `random_element()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra* method), 137
- `random_element()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract* method), 620
- `random_element()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational* method), 638
- `random_element()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionOrder* method), 651
- `rank` (*sage.algebras.free_algebra.AssociativeFunctor* attribute), 63
- `rank` (*sage.algebras.free_zinbiel_algebra.ZinbielFunctor* attribute), 1393
- `rank` (*sage.algebras.tensor_algebra.TensorAlgebraFunctor* attribute), 120
- `rank` (*sage.combinat.free_dendriform_algebra.DendriformFunctor* attribute), 1343
- `rank` (*sage.combinat.free_prelie_algebra.PreLieFunctor* attribute), 1369
- `rank()` (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 322
- `rank()` (*sage.algebras.free_algebra_quotient.FreeAlgebraQuotient* method), 111
- `rank()` (*sage.algebras.quantum_clifford.QuantumCliffordAlgebra* method), 531
- `RankTwoHeisenbergVirasoro` (class in *sage.algebras.lie_algebras.rank_two_heisenberg_virasoro*), 1228
- `RankTwoHeisenbergVirasoro.Element` (class in *sage.algebras.lie_algebras.rank_two_heisenberg_virasoro*), 1229
- `RationalCherednikAlgebra` (class in *sage.algebras.rational_cherednik_algebra*), 658
- `real_part()` (*sage.algebras.octonion_algebra.Octonion_generic* method), 1054
- `reduce()` (*sage.algebras.clifford_algebra.ExteriorAlgebraIdeal* method), 294
- `reduce()` (*sage.algebras.letterplace.free_algebra_element_letterplace.FreeAlgebraElement_letterplace* method), 97
- `reduce()` (*sage.algebras.letterplace.letterplace_ideal.LetterplaceIdeal* method), 105
- `reduce_to_irr_block()` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.CubicHeckeMatrixRep* method), 925
- `reduced_basis()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational* method), 638
- `reduced_subalgebra()` (*sage.combinat.posets.incidence_algebras.IncidenceAlgebra* method), 464
- `ReducedIncidenceAlgebra` (class in *sage.combinat.posets.incidence_algebras*), 465
- `ReducedIncidenceAlgebra.Element` (class in *sage.combinat.posets.incidence_algebras*), 465
- `reflection_automorphism()` (*sage.algebras.askey_wilson.AskeyWilsonAlgebra* method), 170
- `register_ring_hom()` (in module *sage.algebras.hecke_algebras.cubic_hecke_base_ring*), 916
- `regular_vector_fields()` (in module *sage.algebras.lie_algebras.examples*), 1123
- `RegularLeft` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.RepresentationType* attribute), 930
- `RegularRight` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.RepresentationType* attribute), 930
- `repr_factored()` (in module *sage.algebras.weyl_algebra*), 783
- `repr_from_monomials()` (in module *sage.algebras.weyl_algebra*), 784
- `repr_type` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* attribute), 883
- `Representation_abstract` (class in *sage.algebras.lie_algebras.representation*), 1237
- `representation_matrix()` (*sage.algebras.lie_algebras.representation.Representation_abstract* method), 1238
- `representation_matrix()` (*sage.combinat.dia-*

- `gram_algebras.PottsRepresentation` (method), 233
- `RepresentationByMorphism` (class in `sage.algebras.lie_algebras.representation`), 1235
- `RepresentationByMorphism.Element` (class in `sage.algebras.lie_algebras.representation`), 1237
- `RepresentationType` (class in `sage.algebras.hecke_algebras.cubic_hecke_matrix_rep`), 930
- `representative()` (`sage.algebras.commutative_dga.Co-homologyClass` method), 941
- `reset_current_seed()` (`sage.algebras.cluster_algebra.ClusterAlgebra` method), 322
- `reset_exploring_iterator()` (`sage.algebras.cluster_algebra.ClusterAlgebra` method), 323
- `reset_filecache()` (`sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra` method), 883
- `resize()` (in module `sage.algebras.fusion_rings.poly_tup_engine`), 438
- `restricted_partitions()` (in module `sage.algebras.steenrod.steenrod_algebra_bases`), 739
- `retract()` (`sage.algebras.cluster_algebra.ClusterAlgebra` method), 323
- `retract()` (`sage.algebras.lie_algebras.affine_lie_algebra.TwistedAffineLieAlgebra` method), 1068
- `retract()` (`sage.algebras.lie_algebras.bgg_dual_module.SimpleModule` method), 1088
- `retract()` (`sage.algebras.lie_algebras.center_uea.CenterUEA` method), 1099
- `retract()` (`sage.algebras.lie_algebras.quotient.LieQuotientFiniteDimensionalWithBasis` method), 1227
- `retract()` (`sage.algebras.lie_algebras.subalgebra.LieSubalgebraFiniteDimensionalWithBasis` method), 1256
- `retract()` (`sage.algebras.quantum_groups.quantum_group_gap.HighestWeightSubmodule` method), 543
- `retract()` (`sage.algebras.quantum_groups.quantum_group_gap.LowerHalfQuantumGroup` method), 550
- `retract()` (`sage.combinat.diagram_algebras.SubPartitionAlgebra` method), 239
- `revert_garside()` (`sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeElement` method), 894
- `revert_mirror()` (`sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeElement` method), 894
- `revert_orientation()` (`sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeElement` method), 895
- `rho()` (`sage.algebras.askey_wilson.AskeyWilsonAlgebra` method), 171
- `ribbon()` (`sage.algebras.fusion_rings.fusion_double.FusionDouble.Element` method), 424
- `ribbon()` (`sage.algebras.fusion_rings.fusion_ring.FusionRing.Element` method), 373
- `right_ideal()` (`sage.algebras.quatalg.quaternion_algebra.QuaternionOrder` method), 652
- `right_order()` (`sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdealRational` method), 639
- `root_of_unity()` (`sage.algebras.fusion_rings.fusion_double.FusionDouble` method), 430
- `root_of_unity()` (`sage.algebras.fusion_rings.fusion_ring.FusionRing` method), 385
- ## S
- `s()` (`sage.combinat.diagram_algebras.PartitionAlgebra` method), 220
- `s_ij()` (`sage.algebras.fusion_rings.fusion_double.FusionDouble` method), 431
- `s_ij()` (`sage.algebras.fusion_rings.fusion_ring.FusionRing` method), 386
- `s_ijconj()` (`sage.algebras.fusion_rings.fusion_double.FusionDouble` method), 432
- `s_ijconj()` (`sage.algebras.fusion_rings.fusion_ring.FusionRing` method), 387
- `s_matrix()` (`sage.algebras.fusion_rings.fusion_double.FusionDouble` method), 432
- `s_matrix()` (`sage.algebras.fusion_rings.fusion_ring.FusionRing` method), 387
- `sage.algebras.affine_nil_temperley_lieb` module, 151
- `sage.algebras.askey_wilson` module, 155
- `sage.algebras.associated_graded` module, 1007
- `sage.algebras.catalog` module, 1
- `sage.algebras.cellular_basis` module, 1011
- `sage.algebras.clifford_algebra` module, 257
- `sage.algebras.cluster_algebra` module, 295
- `sage.algebras.commutative_dga` module, 939
- `sage.algebras.down_up_algebra` module, 353
- `sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra` module, 121
- `sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element` module, 138

sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_ideal
 module, 145
 sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism
 module, 146
 sage.algebras.finite_gca
 module, 933
 sage.algebras.free_algebra
 module, 59
 sage.algebras.free_algebra_element
 module, 80
 sage.algebras.free_algebra_quotient
 module, 106
 sage.algebras.free_algebra_quotient_element
 module, 112
 sage.algebras.free_zinbiel_algebra
 module, 1385
 sage.algebras.fusion_rings.f_matrix
 module, 391
 sage.algebras.fusion_rings.fast_parallel_fmats_methods
 module, 433
 sage.algebras.fusion_rings.fast_parallel_fusion_ring_braid_repn
 module, 434
 sage.algebras.fusion_rings.fusion_double
 module, 419
 sage.algebras.fusion_rings.fusion_ring
 module, 365
 sage.algebras.fusion_rings.poly_tup_engine
 module, 434
 sage.algebras.fusion_rings.shm_managers
 module, 440
 sage.algebras.group_algebra
 module, 469
 sage.algebras.hall_algebra
 module, 447
 sage.algebras.hecke_algebras.ariki_koike_algebra
 module, 803
 sage.algebras.hecke_algebras.ariki_koike_specht_modules
 module, 816
 sage.algebras.hecke_algebras.cubic_hecke_algebra
 module, 862
 sage.algebras.hecke_algebras.cubic_hecke_base_ring
 module, 895
 sage.algebras.hecke_algebras.cubic_hecke_matrix_rep
 module, 917
 sage.algebras.iwahori_hecke_algebra
 module, 821
 sage.algebras.jordan_algebra
 module, 1323
 sage.algebras.letterplace.free_algebra_element_letterplace
 module, 91
 sage.algebras.letterplace.free_algebra_letterplace
 module, 82
 sage.algebras.letterplace.letterplace_ideal
 module, 98
 sage.algebras.lie_algebras.abelian
 module, 1055
 sage.algebras.lie_algebras.affine_lie_algebra
 module, 1058
 sage.algebras.lie_algebras.bch
 module, 1072
 sage.algebras.lie_algebras.bgg_dual_module
 module, 1077
 sage.algebras.lie_algebras.bgg_resolution
 module, 1074
 sage.algebras.lie_algebras.center_uea
 module, 1092
 sage.algebras.lie_algebras.classical_lie_algebra
 module, 1100
 sage.algebras.lie_algebras.examples
 module, 1121
 sage.algebras.lie_algebras.free_lie_algebra
 module, 1133
 sage.algebras.lie_algebras.heisenberg
 module, 1141
 sage.algebras.lie_algebras.lie_algebra
 module, 1151
 sage.algebras.lie_algebras.lie_algebra_element
 module, 1173
 sage.algebras.lie_algebras.morphism
 module, 1183
 sage.algebras.lie_algebras.nilpotent_lie_algebra
 module, 1191
 sage.algebras.lie_algebras.onsager
 module, 1195
 sage.algebras.lie_algebras.poincare_birkhoff_witt
 module, 1211

`sage.algebras.lie_algebras.quotient`
 module, [1221](#)

`sage.algebras.lie_algebras.rank_two_heisenberg_virasoro`
 module, [1228](#)

`sage.algebras.lie_algebras.representation`
 module, [1231](#)

`sage.algebras.lie_algebras.structure_coefficients`
 module, [1240](#)

`sage.algebras.lie_algebras.subalgebra`
 module, [1244](#)

`sage.algebras.lie_algebras.symplectic_derivation`
 module, [1258](#)

`sage.algebras.lie_algebras.verma_module`
 module, [1261](#)

`sage.algebras.lie_algebras.virasoro`
 module, [1279](#)

`sage.algebras.lie_conformal_algebras.abelian_lie_conformal_algebra`
 module, [1299](#)

`sage.algebras.lie_conformal_algebras.affine_lie_conformal_algebra`
 module, [1300](#)

`sage.algebras.lie_conformal_algebras.bosonic_ghosts_lie_conformal_algebra`
 module, [1302](#)

`sage.algebras.lie_conformal_algebras.examples`
 module, [1297](#)

`sage.algebras.lie_conformal_algebras.fermionic_ghosts_lie_conformal_algebra`
 module, [1303](#)

`sage.algebras.lie_conformal_algebras.finitely_freely_generated_lca`
 module, [1315](#)

`sage.algebras.lie_conformal_algebras.free_bosons_lie_conformal_algebra`
 module, [1305](#)

`sage.algebras.lie_conformal_algebras.free_fermions_lie_conformal_algebra`
 module, [1307](#)

`sage.algebras.lie_conformal_algebras.freely_generated_lie_conformal_algebra`
 module, [1316](#)

`sage.algebras.lie_conformal_algebras.graded_lie_conformal_algebra`
 module, [1318](#)

`sage.algebras.lie_conformal_algebras.lie_conformal_algebra`
 module, [1292](#)

`sage.algebras.lie_conformal_algebras.lie_conformal_algebra_element`
 module, [1298](#)

`sage.algebras.lie_conformal_algebras.lie_conformal_algebra_with_basis`
 module, [1320](#)

`sage.algebras.lie_conformal_algebras.lie_conformal_algebra_with_structure_coefs`
 module, [1321](#)

`sage.algebras.lie_conformal_algebras.n2_lie_conformal_algebra`
 module, [1309](#)

`sage.algebras.lie_conformal_algebras.neveu_schwarz_lie_conformal_algebra`
 module, [1310](#)

`sage.algebras.lie_conformal_algebras.virasoro_lie_conformal_algebra`
 module, [1311](#)

`sage.algebras.lie_conformal_algebras.weyl_lie_conformal_algebra`
 module, [1312](#)

`sage.algebras.nil_coxeter_algebra`
 module, [855](#)

`sage.algebras.octonion_algebra`
 module, [1043](#)

`sage.algebras.orlik_solomon`
 module, [496](#)

`sage.algebras.orlik_terao`
 module, [487](#)

`sage.algebras.q_commuting_polynomials`
 module, [1023](#)

`sage.algebras.q_system`
 module, [1017](#)

`sage.algebras.quantum_clifford`
 module, [526](#)

`sage.algebras.quantum_groups.ace_quantum_onsager`
 module, [3](#)

`sage.algebras.quantum_groups.fock_space`
 module, [13](#)

`sage.algebras.quantum_groups.q_numbers`
 module, [44](#)

`sage.algebras.quantum_groups.quantum_group_gap`
 module, [539](#)

`sage.algebras.quantum_groups.representations`
 module, [47](#)

`sage.algebras.quantum_matrix_coordinate_algebra`
 module, 570
`sage.algebras.quantum_oscillator`
 module, 581
`sage.algebras.quatalg.quaternion_algebra`
 module, 589
`sage.algebras.rational_cherednik_algebra`
 module, 658
`sage.algebras.schur_algebra`
 module, 662
`sage.algebras.shuffle_algebra`
 module, 1370
`sage.algebras.splitting_algebra`
 module, 1034
`sage.algebras.steenrod.steenrod_algebra`
 module, 669
`sage.algebras.steenrod.steenrod_algebra_bases`
 module, 729
`sage.algebras.steenrod.steenrod_algebra_misc`
 module, 745
`sage.algebras.steenrod.steenrod_algebra_mult`
 module, 762
`sage.algebras.tensor_algebra`
 module, 114
`sage.algebras.weyl_algebra`
 module, 773
`sage.algebras.yangian`
 module, 787
`sage.algebras.yokonuma_hecke_algebra`
 module, 857
`sage.combinat.descent_algebra`
 module, 338
`sage.combinat.diagram_algebras`
 module, 173
`sage.combinat.free_dendriform_algebra`
 module, 1341
`sage.combinat.free_prelie_algebra`
 module, 1354
`sage.combinat.grossman_larson_algebras`
 module, 471
`sage.combinat.partition_algebra`
 module, 509
`sage.combinat.posets.incidence_algebras`
 module, 460
`sage.combinat.posets.moebius_algebra`
 module, 480
`save_fvars()` (*sage.algebras.fusion_rings.f_matrix.FMatrix method*), 416
`scalar()` (*sage.algebras.hall_algebra.HallAlgebra.Element method*), 450
`scalar()` (*sage.algebras.hall_algebra.HallAlgebraMonomials.Element method*), 456
`scalar_base_ring()` (*sage.algebras.splitting_algebra.SplittingAlgebra method*), 1039
`scalars()` (*sage.algebras.cluster_algebra.ClusterAlgebra method*), 324
`scale()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational method*), 640
`schur_element()` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra method*), 884
`schur_elements()` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra method*), 884
`schur_representative_from_index()` (*in module sage.algebras.schur_algebra*), 668
`schur_representative_indices()` (*in module sage.algebras.schur_algebra*), 668
`SchurAlgebra` (class in *sage.algebras.schur_algebra*), 664
`SchurTensorModule` (class in *sage.algebras.schur_algebra*), 666
`SchurTensorModule.Element` (class in *sage.algebras.schur_algebra*), 667
`section()` (*sage.algebras.lie_algebras.lie_algebra.LiftMorphismToAssociative method*), 1171
`seeds()` (*sage.algebras.cluster_algebra.ClusterAlgebra method*), 324
`serre_cartan_basis()` (*in module sage.algebras.steenrod.steenrod_algebra_bases*), 741
`serre_cartan_mono_to_string()` (*in module sage.algebras.steenrod.steenrod_algebra_misc*), 759
`set_current_seed()` (*sage.algebras.cluster_algebra.ClusterAlgebra method*), 325
`set_degbound()` (*sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace method*), 89
`set_partition()` (*sage.combinat.diagram_algebras.AbstractPartitionDiagram method*), 178
`set_partition_composition()` (*in module sage.combinat.partition_algebra*), 525
`set_partitions()` (*sage.combinat.diagram_algebras.DiagramAlgebra method*), 188
`SetPartitionsAk()` (*in module sage.combinat.partition_algebra*), 512
`SetPartitionsAk_k` (class in *sage.combinat.partition_algebra*), 514
`SetPartitionsAkhalf_k` (class in *sage.combinat.partition_algebra*), 514
`SetPartitionsBk()` (*in module sage.combinat.partition_algebra*), 514
`SetPartitionsBk_k` (class in *sage.combinat.partition_algebra*), 514

- tion_algebra), 515
- SetPartitionsBkhalf_k (class in sage.combinat.partition_algebra), 516
- SetPartitionsIk() (in module sage.combinat.partition_algebra), 516
- SetPartitionsIk_k (class in sage.combinat.partition_algebra), 517
- SetPartitionsIkhalf_k (class in sage.combinat.partition_algebra), 517
- SetPartitionsPk() (in module sage.combinat.partition_algebra), 517
- SetPartitionsPk_k (class in sage.combinat.partition_algebra), 518
- SetPartitionsPkhalf_k (class in sage.combinat.partition_algebra), 519
- SetPartitionsPRk() (in module sage.combinat.partition_algebra), 517
- SetPartitionsPRk_k (class in sage.combinat.partition_algebra), 517
- SetPartitionsPRkhalf_k (class in sage.combinat.partition_algebra), 517
- SetPartitionsRk() (in module sage.combinat.partition_algebra), 519
- SetPartitionsRk_k (class in sage.combinat.partition_algebra), 519
- SetPartitionsRkhalf_k (class in sage.combinat.partition_algebra), 519
- SetPartitionsSk() (in module sage.combinat.partition_algebra), 519
- SetPartitionsSk_k (class in sage.combinat.partition_algebra), 521
- SetPartitionsSkhalf_k (class in sage.combinat.partition_algebra), 521
- SetPartitionsTk() (in module sage.combinat.partition_algebra), 521
- SetPartitionsTk_k (class in sage.combinat.partition_algebra), 522
- SetPartitionsTkhalf_k (class in sage.combinat.partition_algebra), 522
- SetPartitionsXkElement (class in sage.combinat.partition_algebra), 522
- shm (sage.algebras.fusion_rings.shm_managers.FvarsHandler attribute), 443
- shm (sage.algebras.fusion_rings.shm_managers.KSHandler attribute), 445
- shuffle_algebra() (sage.algebras.shuffle_algebra.DualPBWBasis method), 1376
- ShuffleAlgebra (class in sage.algebras.shuffle_algebra), 1376
- shutdown_worker_pool() (sage.algebras.fusion_rings.f_matrix.FMatrix method), 417
- side() (sage.algebras.free_zinbiel_algebra.FreeZinbielAlgebra method), 1391
- side() (sage.algebras.lie_algebras.representation.RepresentationAbstract method), 1239
- sigma() (sage.algebras.askey_wilson.AskeyWilsonAlgebra method), 172
- sigma() (sage.algebras.quantum_groups.ace_quantum_onsager.ACEQuantumOnsagerAlgebra method), 12
- sigma() (sage.combinat.diagram_algebras.PartitionAlgebra method), 221
- simple_module() (sage.algebras.lie_algebras.bgg_resolution.BGGResolution method), 1076
- simple_root() (sage.algebras.lie_algebras.classical_lie_algebra.ClassicalMatrixLieAlgebra method), 1105
- simple_root() (sage.algebras.lie_algebras.classical_lie_algebra.sl method), 1118
- simple_root() (sage.algebras.lie_algebras.classical_lie_algebra.so method), 1119
- simple_root() (sage.algebras.lie_algebras.classical_lie_algebra.sp method), 1121
- SimpleLieCenterIndices (class in sage.algebras.lie_algebras.center_uea), 1100
- SimpleModule (class in sage.algebras.lie_algebras.bgg_dual_module), 1082
- SimpleModule.Element (class in sage.algebras.lie_algebras.bgg_dual_module), 1082
- SimpleModuleIndices (class in sage.algebras.lie_algebras.bgg_dual_module), 1090
- single_vertex() (sage.combinat.grossman_larson_algebras.GrossmanLarsonAlgebra method), 477
- single_vertex_all() (sage.combinat.grossman_larson_algebras.GrossmanLarsonAlgebra method), 478
- singular_vector() (sage.algebras.lie_algebras.verma_module.VermaModuleHomset method), 1274
- sl (class in sage.algebras.lie_algebras.classical_lie_algebra), 1118
- sl() (in module sage.algebras.lie_algebras.examples), 1124
- so (class in sage.algebras.lie_algebras.classical_lie_algebra), 1119
- so() (in module sage.algebras.lie_algebras.examples), 1125
- solve_with_extension() (in module sage.algebras.splitting_algebra), 1041
- some_elements() (sage.algebras.askey_wilson.AskeyWilsonAlgebra method), 173
- some_elements() (sage.algebras.fusion_rings.fusion_ring.FusionRing method), 388
- some_elements() (sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.CubicHeckeMatrixSpace method), 928
- some_elements() (sage.algebras.jordan_algebra.ExceptionalJordanAlgebra method), 1328

- `some_elements()` (*sage.algebras.lie_algebras.center_uea.CenterIndices* method), 1093
- `some_elements()` (*sage.algebras.lie_algebras.onsager.OnsagerAlgebra* method), 1199
- `some_elements()` (*sage.algebras.lie_algebras.onsager.OnsagerAlgebraACE* method), 1204
- `some_elements()` (*sage.algebras.lie_algebras.onsager.QuantumOnsagerAlgebra* method), 1211
- `some_elements()` (*sage.algebras.lie_algebras.rank_two_heisenberg_virasoro.RankTwoHeisenbergVirasoro* method), 1230
- `some_elements()` (*sage.algebras.lie_algebras.structure_coefficients.LieAlgebraWithStructureCoefficients* method), 1243
- `some_elements()` (*sage.algebras.lie_algebras.symplectic_derivation.SymplecticDerivationLieAlgebra* method), 1260
- `some_elements()` (*sage.algebras.lie_algebras.virasoro.LieAlgebraRegularVectorFields* method), 1283
- `some_elements()` (*sage.algebras.lie_algebras.virasoro.VirasoroAlgebra* method), 1290
- `some_elements()` (*sage.algebras.lie_algebras.virasoro.WittLieAlgebra_charp* method), 1292
- `some_elements()` (*sage.algebras.octonion_algebra.OctonionAlgebra* method), 1049
- `some_elements()` (*sage.algebras.quantum_groups.ace_quantum_onsager.ACEQuantumOnsagerAlgebra* method), 13
- `some_elements()` (*sage.algebras.quantum_groups.fock_space.FockSpaceBases.ParentMethods* method), 36
- `some_elements()` (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroup* method), 563
- `some_elements()` (*sage.algebras.quantum_groups.quantum_group_gap.TensorProductOfHighestWeightModules* method), 568
- `some_elements()` (*sage.algebras.quantum_oscillator.FockSpaceRepresentation* method), 581
- `some_elements()` (*sage.algebras.quantum_oscillator.QuantumOscillatorAlgebra* method), 588
- `some_elements()` (*sage.algebras.rational_cherednik_algebra.RationalCherednikAlgebra* method), 662
- `some_elements()` (*sage.algebras.shuffle_algebra.DualPBWBasis* method), 1376
- `some_elements()` (*sage.algebras.shuffle_algebra.ShuffleAlgebra* method), 1383
- `some_elements()` (*sage.combinat.free_dendriform_algebra.FreeDendriformAlgebra* method), 1351
- `some_elements()` (*sage.combinat.free_prelie_algebra.FreePreLieAlgebra* method), 1366
- `some_elements()` (*sage.combinat.grossman_larson_algebras.GrossmanLarsonAlgebra* method), 478
- `some_elements()` (*sage.combinat.posets.incidence_algebras.IncidenceAlgebra* method), 464
- `some_elements()` (*sage.combinat.posets.incidence_algebras.ReducedIncidenceAlgebra* method), 468
- `sorting_keys()` (in module *sage.algebras.commutative_dga*), 1003
- `sp` (class in *sage.algebras.lie_algebras.classical_lie_algebra*), 1120
- `sp()` (in module *sage.algebras.lie_algebras.examples*), 1127
- `specht_module()` (*sage.algebras.hecke_algebras.ariki_koike_algebra.ArikiKoikeAlgebra* method), 816
- `SpechtModule` (class in *sage.algebras.hecke_algebras.ariki_koike_specht_modules*), 816
- `SpechtModule.Element` (class in *sage.algebras.hecke_algebras.ariki_koike_specht_modules*), 818
- `specialize_homfly()` (*sage.algebras.hecke_algebras.cubic_hecke_base_ring.CubicHeckeRingOfDefinition* method), 911
- `specialize_kauffman()` (*sage.algebras.hecke_algebras.cubic_hecke_base_ring.CubicHeckeRingOfDefinition* method), 912
- `specialize_links_gould()` (*sage.algebras.hecke_algebras.cubic_hecke_base_ring.CubicHeckeRingOfDefinition* method), 914
- `SpecialJordanAlgebra` (class in *sage.algebras.jordan_algebra*), 1338
- `SpecialJordanAlgebra.Element` (class in *sage.algebras.jordan_algebra*), 1338
- `SplitIrredChevie` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.RepresentationType* attribute), 930
- `SplitIrredMarin` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.RepresentationType* attribute), 930
- `splitting_roots()` (*sage.algebras.splitting_algebra.SplittingAlgebra* method), 1039
- `SplittingAlgebra` (class in *sage.algebras.splitting_algebra*), 1034
- `SplittingAlgebraElement` (class in *sage.algebras.splitting_algebra*), 1040
- `Sq()` (in module *sage.algebras.steenrod.steenrod_algebra*), 679
- `Sq()` (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_mod_two* method), 729
- `standard` (*sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra* attribute), 848
- `standard` (*sage.combinat.descent_algebra.DescentAlgebra* attribute), 350
- `start_worker_pool()` (*sage.algebras.fusion_rings.f_matrix.FMatrix* method), 417
- `steenrod_algebra_basis()` (in module *sage.alge-*

- bras.steenrod.steenrod_algebra_bases*), 741
- steenrod_basis_error_check()* (in module *sage.algebras.steenrod.steenrod_algebra_bases*), 743
- SteenrodAlgebra()* (in module *sage.algebras.steenrod.steenrod_algebra*), 680
- SteenrodAlgebra_generic* (class in *sage.algebras.steenrod.steenrod_algebra*), 689
- SteenrodAlgebra_generic.Element* (class in *sage.algebras.steenrod.steenrod_algebra*), 689
- SteenrodAlgebra_mod_two* (class in *sage.algebras.steenrod.steenrod_algebra*), 728
- step()* (*sage.algebras.lie_algebras.heisenberg.HeisenbergAlgebra_abstract* method), 1143
- step()* (*sage.algebras.lie_algebras.heisenberg.HeisenbergAlgebra_matrix* method), 1150
- strands()* (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeAlgebra* method), 885
- strictly_upper_triangular_matrices()* (in module *sage.algebras.lie_algebras.examples*), 1128
- structure_coefficients()* (*sage.algebras.lie_algebras.structure_coefficients.LieAlgebraWithStructureCoefficients* method), 1243
- structure_coefficients()* (*sage.algebras.lie_conformal_algebras.lie_conformal_algebra_with_structure_coefs.LieConformalAlgebraWithStructureCoefficients* method), 1323
- StructureCoefficientsElement* (class in *sage.algebras.lie_algebras.lie_algebra_element*), 1178
- su()* (in module *sage.algebras.lie_algebras.examples*), 1129
- SubPartitionAlgebra* (class in *sage.combinat.diagram_algebras*), 237
- SubPartitionAlgebra.Element* (class in *sage.combinat.diagram_algebras*), 237
- subset* (*sage.combinat.descent_algebra.DescentAlgebra* attribute), 350
- subset_image()* (*sage.algebras.orlik_solomon.OrlikSolomonAlgebra* method), 502
- subset_image()* (*sage.algebras.orlik_terao.OrlikTeraoAlgebra* method), 491
- succ()* (*sage.combinat.free_dendriform_algebra.FreeDendriformAlgebra* method), 1352
- succ_product_on_basis()* (*sage.combinat.free_dendriform_algebra.FreeDendriformAlgebra* method), 1352
- super_categories()* (*sage.algebras.lie_algebras.free_lie_algebra.FreeLieAlgebraBases* method), 1138
- super_categories()* (*sage.algebras.quantum_groups.fock_space.FockSpaceBases* method), 37
- super_categories()* (*sage.combinat.descent_algebra.DescentAlgebraBases* method), 352
- super_categories()* (*sage.combinat.posets.moebius_algebra.MoebiusAlgebraBases* method), 484
- supercenter_basis()* (*sage.algebras.clifford_algebra.CliffordAlgebra* method), 271
- symmetric_diagrams()* (*sage.combinat.diagram_algebras.BrauerDiagrams* method), 186
- symmetric_group()* (*sage.combinat.diagram_algebras.PottsRepresentation* method), 235
- SymplecticDerivationLieAlgebra* (class in *sage.algebras.lie_algebras.symplectic_derivation*), 1258
- SymplecticDerivationLieAlgebra.Element* (class in *sage.algebras.lie_algebras.symplectic_derivation*), 1259
- ## T
- T()* (*sage.algebras.hecke_algebras.ariki_koike_algebra.ArikiKoikeAlgebra.LT* method), 808
- T()* (*sage.algebras.hecke_algebras.ariki_koike_algebra.ArikiKoikeAlgebra.T* method), 812
- T()* (*sage.algebras.hecke_algebras.ariki_koike_specht_modules.SpechtModule.Element* method), 819
- t()* (*sage.algebras.lie_algebras.rank_two_heisenberg_virasoro.RankTwoHeisenbergVirasoro* method), 1231
- T()* (*sage.algebras.lie_conformal_algebras.lie_conformal_algebra_element.LCAWithGeneratorsElement* method), 1298
- t()* (*sage.algebras.yokonuma_hecke_algebra.YokonumaHeckeAlgebra* method), 862
- t_dict()* (*sage.algebras.lie_algebras.lie_algebra_element.UntwistedAffineLieAlgebraElement* method), 1182
- table()* (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra* method), 138
- tau()* (*sage.algebras.quantum_groups.quantum_group_gap.LowerHalfQuantumGroup.Element* method), 546
- tau()* (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroup.Element* method), 557
- temperley_lieb_diagrams()* (in module *sage.combinat.diagram_algebras*), 255
- TemperleyLiebAlgebra* (class in *sage.combinat.diagram_algebras*), 242
- TemperleyLiebDiagram* (class in *sage.combinat.diagram_algebras*), 246
- TemperleyLiebDiagrams* (class in *sage.combinat.diagram_algebras*), 247
- tensor()* (*sage.algebras.quantum_groups.quantum_group_gap.HighestWeightModule* method), 540

- `tensor_factors()` (*sage.algebras.quantum_groups.quantum_group_gap.TensorProductOfHighestWeightModules* method), 568
`TensorAlgebra` (class in *sage.algebras.tensor_algebra*), 114
`TensorAlgebraFunctor` (class in *sage.algebras.tensor_algebra*), 120
`TensorProductOfHighestWeightModules` (class in *sage.algebras.quantum_groups.quantum_group_gap*), 567
`term()` (*sage.algebras.lie_algebras.classical_lie_algebra.MatrixCompactRealForm* method), 1114
`term()` (*sage.algebras.lie_algebras.lie_algebra.LieAlgebra* method), 1162
`term()` (*sage.algebras.lie_algebras.lie_algebra.LieAlgebraFromAssociative* method), 1169
`term()` (*sage.algebras.lie_algebras.structure_coefficients.LieAlgebraWithStructureCoefficients* method), 1243
`term_order_of_block()` (*sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebraLetterplace* method), 90
`ternary_quadratic_form()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionOrder* method), 652
`test_braid_representation()` (*sage.algebras.fusion_rings.fusion_ring.FusionRing* method), 388
`theta_basis_decomposition()` (*sage.algebras.cluster_algebra.PrincipalClusterAlgebraElement* method), 338
`theta_basis_element()` (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 326
`theta_basis_F_polynomial()` (*sage.algebras.cluster_algebra.ClusterAlgebra* method), 325
`theta_series()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational* method), 641
`theta_series_vector()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational* method), 642
`three_dimensional()` (in module *sage.algebras.lie_algebras.examples*), 1130
`three_dimensional_by_rank()` (in module *sage.algebras.lie_algebras.examples*), 1131
`Tietze()` (*sage.algebras.hecke_algebras.cubic_hecke_algebra.CubicHeckeElement* method), 886
`TL_diagram_ascii_art()` (in module *sage.combinat.diagram_algebras*), 239
`to_ambient()` (*sage.algebras.lie_algebras.affine_lie_algebra.TwistedAffineLieAlgebra* method), 1069
`to_B_basis()` (*sage.combinat.descent_algebra.DescentAlgebra.D* method), 345
`to_B_basis()` (*sage.combinat.descent_algebra.DescentAlgebra.I* method), 349
`to_Brauer_partition()` (in module *sage.combinat.diagram_algebras*), 255
`to_C_basis()` (*sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_nonstandard.T* method), 851
`to_C_basis()` (*sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T* method), 845
`to_Cp_basis()` (*sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_nonstandard.T* method), 852
`to_Cp_basis()` (*sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T* method), 846
`to_D_basis()` (*sage.combinat.descent_algebra.DescentAlgebra.B* method), 341
`to_diagram_basis()` (*sage.combinat.diagram_algebras.OrbitBasis.Element* method), 193
`to_dual_pbw_element()` (*sage.algebras.shuffle_algebra.ShuffleAlgebra* method), 1383
`to_graph()` (in module *sage.combinat.diagram_algebras*), 256
`to_graph()` (in module *sage.combinat.partition_algebra*), 525
`to_I_basis()` (*sage.combinat.descent_algebra.DescentAlgebra.B* method), 342
`to_matrix()` (*sage.combinat.posets.incidence_algebras.IncidenceAlgebra.Element* method), 461
`to_matrix()` (*sage.combinat.posets.incidence_algebras.ReducedIncidenceAlgebra.Element* method), 466
`to_nsym()` (*sage.combinat.descent_algebra.DescentAlgebra.B* method), 343
`to_orbit_basis()` (*sage.combinat.diagram_algebras.PartitionAlgebra.Element* method), 205
`to_orbit_basis()` (*sage.combinat.diagram_algebras.SubPartitionAlgebra.Element* method), 238
`to_pbw_basis()` (*sage.algebras.free_algebra_element.FreeAlgebraElement* method), 81
`to_set_partition()` (in module *sage.combinat.diagram_algebras*), 256
`to_set_partition()` (in module *sage.combinat.partition_algebra*), 526
`to_symmetric_group_algebra()` (*sage.combinat.descent_algebra.DescentAlgebraBases.ElementMethods* method), 350
`to_symmetric_group_algebra()` (*sage.combinat.descent_algebra.DescentAlgebraBases.ParentMethods* method), 351
`to_symmetric_group_algebra_on_basis()` (*sage.combinat.descent_algebra.DescentAlgebraBases.ParentMethods* method), 351
`to_symmetric_group_algebra_on_basis()` (*sage.combinat.descent_algebra.DescentAl-*

gebra.D method), 346

to_T_basis() (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_nonstandard.C method), 849

to_T_basis() (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_nonstandard.Cp method), 850

to_T_basis() (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.A method), 829

to_T_basis() (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.B method), 831

to_vector() (sage.algebras.lie_algebras.lie_algebra_element.LieSubalgebraElementWrapper method), 1177

to_vector() (sage.algebras.lie_algebras.lie_algebra_element.StructureCoefficientsElement method), 1180

to_word() (sage.algebras.lie_algebras.lie_algebra_element.LieBracket method), 1175

to_word() (sage.algebras.lie_algebras.lie_algebra_element.LieGenerator method), 1176

to_word() (sage.algebras.lie_algebras.lie_algebra_element.LieObject method), 1177

top_class() (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 728

total_degree() (in module sage.algebras.commutative_dga), 1004

total_q_order() (sage.algebras.fusion_rings.fusion_double.FusionDouble method), 433

total_q_order() (sage.algebras.fusion_rings.fusion_ring.FusionRing method), 389

trace() (sage.algebras.jordan_algebra.JordanAlgebraSymmetricBilinear.Element method), 1336

transpose() (sage.algebras.lie_algebras.poincare_birkhoff_witt.PoincareBirkhoffWittBasisSemisimpleLieAlgebra method), 1220

transpose() (sage.algebras.lie_algebras.poincare_birkhoff_witt.PoincareBirkhoffWittBasisSemisimpleLieAlgebra.Element method), 1218

transpose_cmp() (in module sage.algebras.hall_algebra), 459

tree_from_sortkey() (in module sage.combinat.free_prelie_algebra), 1370

trivial_idempotent() (sage.algebras.rational_cherednik_algebra.RationalCherednikAlgebra method), 662

TrivialRepresentation (class in sage.algebras.lie_algebras.representation), 1239

TrivialRepresentation.Element (class in sage.algebras.lie_algebras.representation), 1240

tup_to_univ_poly() (in module sage.algebras.fusion_rings.poly_tup_engine), 439

twist() (sage.algebras.fusion_rings.fusion_double.FusionDouble.Element method), 424

twist() (sage.algebras.fusion_rings.fusion_ring.FusionRing.Element method), 374

twist() (sage.algebras.quantum_clifford.QuantumCliffordAlgebra method), 531

TwistedAffineIndices (class in sage.algebras.lie_algebras.affine_lie_algebra), 1065

TwistedAffineLieAlgebra (class in sage.algebras.lie_algebras.affine_lie_algebra), 1067

TwistedAffineLieAlgebra.Element (class in sage.algebras.lie_algebras.affine_lie_algebra), 1067

twists_matrix() (sage.algebras.fusion_rings.fusion_ring.FusionRing method), 389

U

u() (sage.algebras.hecke_algebras.ariki_koike_algebra.ArikiKoikeAlgebra method), 816

under() (sage.combinat.free_dendriform_algebra.FreeDendriformAlgebra method), 1353

unit_ideal() (sage.algebras.quatalg.quaternion_algebra.QuaternionOrder method), 653

UnitDiagramMixin (class in sage.combinat.diagram_algebras), 248

unpickle_FiniteDimensionalAlgebraElement() (in module sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element), 144

unpickle_QuaternionAlgebra_v0() (in module sage.algebras.quatalg.quaternion_algebra), 658

UntwistedAffineLieAlgebra (class in sage.algebras.lie_algebras.affine_lie_algebra), 1069

UntwistedAffineLieAlgebraElement (class in sage.algebras.lie_algebras.lie_algebra_element), 1180

update() (sage.algebras.fusion_rings.shm_managers.KSHandler method), 445

upper_bound() (sage.algebras.cluster_algebra.ClusterAlgebra method), 328

upper_cluster_algebra() (sage.algebras.cluster_algebra.ClusterAlgebra method), 328

upper_triangular_matrices() (in module sage.algebras.lie_algebras.examples), 1132

V

vacuum() (sage.algebras.quantum_oscillator.FockSpaceRepresentation method), 582

valuation() (sage.combinat.free_prelie_algebra.FreePreLieAlgebra.Element method), 1358

variable_names() (sage.algebras.shuffle_algebra.ShuffleAlgebra method), 1384

variable_names() (sage.combinat.free_dendriform_algebra.FreeDendriformAlgebra method), 1353

variable_names() (sage.combinat.free_prelie_algebra.FreePreLieAlgebra method), 1366

`variable_names()` (*sage.combinat.grossman_larson_algebras.GrossmanLarsonAlgebra* method), 479
`variables()` (in module *sage.algebras.fusion_rings.poly_tup_engine*), 439
`variables()` (*sage.algebras.free_algebra_element.FreeAlgebraElement* method), 81
`variables()` (*sage.algebras.weyl_algebra.DifferentialWeylAlgebra* method), 779
`vector()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement* method), 144
`vector()` (*sage.algebras.free_algebra_quotient_element.FreeAlgebraQuotientElement* method), 113
`vector()` (*sage.algebras.octonion_algebra.Octonion_generic* method), 1055
`vector_space()` (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_ideal.FiniteDimensionalAlgebraIdeal* method), 146
`vector_space()` (*sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract* method), 621
`verma_module()` (*sage.algebras.down_up_algebra.DownUpAlgebra* method), 359
`verma_module()` (*sage.algebras.lie_algebras.virasoro.VirasoroAlgebra* method), 1290
`VermaModule` (class in *sage.algebras.down_up_algebra*), 359
`VermaModule` (class in *sage.algebras.lie_algebras.verma_module*), 1261
`VermaModule` (class in *sage.algebras.lie_algebras.virasoro*), 1284
`VermaModule.Element` (class in *sage.algebras.down_up_algebra*), 362
`VermaModule.Element` (class in *sage.algebras.lie_algebras.verma_module*), 1262
`VermaModule.Element` (class in *sage.algebras.lie_algebras.virasoro*), 1285
`VermaModuleHomset` (class in *sage.algebras.lie_algebras.verma_module*), 1271
`VermaModuleMorphism` (class in *sage.algebras.lie_algebras.verma_module*), 1277
`virasoro_algebra()` (*sage.algebras.lie_algebras.virasoro.ChargelessRepresentation* method), 1281
`virasoro_algebra()` (*sage.algebras.lie_algebras.virasoro.VermaModule* method), 1286
`virasoro_central_charge()` (*sage.algebras.fusion_rings.fusion_ring.FusionRing* method), 390
`VirasoroAlgebra` (class in *sage.algebras.lie_algebras.virasoro*), 1287
`VirasoroAlgebra.Element` (class in *sage.algebras.lie_algebras.virasoro*), 1287
`VirasoroLieConformalAlgebra` (class in *sage.alge-*

bras.lie_conformal_algebras.virasoro_lie_conformal_algebra), 1311

`volume_form()` (*sage.algebras.clifford_algebra.ExteriorAlgebra* method), 282

W

`W2_001` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 918
`W2_010` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 918
`W2_100` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 918
`W3_001` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 918
`W3_010` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 918
`W3_011` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 918
`W3_100` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 918
`W3_101` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 918
`W3_110` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 918
`W3_111` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 919
`W4_001` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 919
`W4_010` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 919
`W4_011` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 919
`W4_012` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 919
`W4_021` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 919
`W4_100` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 919
`W4_101` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 919
`W4_102` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 919
`W4_110` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 919
`W4_111` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 919
`W4_120` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 919
`W4_123` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 919
`W4_132` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 919
`W4_201` (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 919

- W4_210 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 919
- W4_213 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 919
- W4_224 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 919
- W4_231 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 919
- W4_242 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 920
- W4_312 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 920
- W4_321 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 920
- W4_333 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 920
- W4_333bar (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 920
- W4_422 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 920
- W5_001 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 920
- W5_010 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 920
- W5_013 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 920
- W5_023 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 920
- W5_031 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 920
- W5_032 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 920
- W5_033 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 920
- W5_100 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 920
- W5_103 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 920
- W5_130 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 920
- W5_136 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 920
- W5_163 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 920
- W5_203 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 920
- W5_230 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 921
- W5_301 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 921
- W5_302 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 921
- W5_303 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 921
- W5_310 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 921
- W5_316 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 921
- W5_320 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 921
- W5_330 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 921
- W5_339 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 921
- W5_361 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 921
- W5_366 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 921
- W5_393 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 921
- W5_613 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 921
- W5_631 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 921
- W5_636 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 921
- W5_663 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 921
- W5_933 (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.AbsIrreducibleRep* attribute), 921
- wall_height() (*sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element* method), 701
- wall_long_mono_to_string() (in module *sage.algebras.steenrod.steenrod_algebra_misc*), 760
- wall_mono_to_string() (in module *sage.algebras.steenrod.steenrod_algebra_misc*), 761
- weight() (*sage.algebras.down_up_algebra.VermaModule.Element* method), 363
- weight() (*sage.algebras.fusion_rings.fusion_ring.FusionRing.Element* method), 375
- weight_space_basis() (*sage.algebras.lie_algebras.bgg_dual_module.SimpleModule* method), 1088
- weight_space_basis() (*sage.algebras.lie_algebras.bgg_dual_module.SimpleModuleIndices* method), 1091
- weight_space_basis() (*sage.algebras.lie_algebras.verma_module.VermaModule* method), 1270
- weights() (*sage.algebras.down_up_algebra.VermaModule* method), 364
- weyl_group() (*sage.algebras.affine_nil_temperley_lieb.AffineNilTemperleyLiebTypeA* method), 155
- WeylLieConformalAlgebra (class in *sage.algebras.lie_conformal_algebras.weyl_lie_confor-*

mal_algebra), 1312
witt() (in module *sage.algebras.lie_algebras.examples*), 1133
WittLieAlgebra_charp (class in *sage.algebras.lie_algebras.virasoro*), 1290
WittLieAlgebra_charp.Element (class in *sage.algebras.lie_algebras.virasoro*), 1291
wood_mono_to_string() (in module *sage.algebras.steenrod.steenrod_algebra_misc*), 762

X

xi_degrees() (in module *sage.algebras.steenrod.steenrod_algebra_bases*), 744

Y

Yangian (class in *sage.algebras.yangian*), 790
YangianLevel (class in *sage.algebras.yangian*), 798
YokonumaHeckeAlgebra (class in *sage.algebras.yokonuma_hecke_algebra*), 857
YokonumaHeckeAlgebra.Element (class in *sage.algebras.yokonuma_hecke_algebra*), 859

Z

z() (*sage.algebras.lie_algebras.heisenberg.HeisenbergAlgebra_abstract* method), 1143
z() (*sage.algebras.lie_algebras.heisenberg.HeisenbergAlgebra_matrix* method), 1150
zero() (*sage.algebras.commutative_dga.GCAlgebraHomset* method), 990
zero() (*sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism.FiniteDimensionalAlgebraHomset* method), 146
zero() (*sage.algebras.hecke_algebras.cubic_hecke_matrix_rep.CubicHeckeMatrixSpace* method), 928
zero() (*sage.algebras.jordan_algebra.ExceptionalJordanAlgebra* method), 1331
zero() (*sage.algebras.jordan_algebra.JordanAlgebraSymmetricBilinear* method), 1338
zero() (*sage.algebras.jordan_algebra.SpecialJordanAlgebra* method), 1341
zero() (*sage.algebras.lie_algebras.affine_lie_algebra.AffineLieAlgebra* method), 1065
zero() (*sage.algebras.lie_algebras.classical_lie_algebra.MatrixCompactRealForm* method), 1114
zero() (*sage.algebras.lie_algebras.lie_algebra.LieAlgebra* method), 1163
zero() (*sage.algebras.lie_algebras.lie_algebra.LieAlgebraFromAssociative* method), 1169
zero() (*sage.algebras.lie_algebras.morphism.LieAlgebraHomset* method), 1186
zero() (*sage.algebras.lie_algebras.structure_coefficients.LieAlgebraWithStructureCoefficients* method), 1244

zero() (*sage.algebras.lie_algebras.subalgebra.LieSubalgebra_finite_dimensional_with_basis* method), 1257
zero() (*sage.algebras.lie_algebras.verma_module.VermaModuleHomset* method), 1276
zero() (*sage.algebras.quantum_groups.quantum_group_gap.LowerHalfQuantumGroup* method), 550
zero() (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroup* method), 564
zero() (*sage.algebras.quantum_groups.quantum_group_gap.QuantumGroupModule* method), 566
zero() (*sage.algebras.weyl_algebra.DifferentialWeylAlgebra* method), 779
zeta() (*sage.combinat.posets.incidence_algebras.IncidenceAlgebra* method), 465
zeta() (*sage.combinat.posets.incidence_algebras.ReducedIncidenceAlgebra* method), 469
ZinbielFunctor (class in *sage.algebras.free_zinbiel_algebra*), 1391