Modules

Release 10.6

The Sage Development Team

Jun 27, 2025

CONTENTS

1 Tutorial: Using free modules and vector spaces 3

2 Free modules, submodules, and quotients 11

3 Modules with basis 249

4 Finitely generated modules over a PID 333

5 Finitely presented graded modules 377

6 Special modules 477

7 Morphisms 611

8 Pseudomorphisms 697

9 Vectors 707

10 Misc 755

11 Indices and Tables 773

Python Module Index 775

Index 777

Sage provides modules of various kinds over various base rings.

CHAPTER

ONE

TUTORIAL: USING FREE MODULES AND VECTOR SPACES

AUTHOR: Jason Bandlow

In this tutorial, we show how to construct and manipulate free modules and vector spaces and their elements.

Sage currently provides two implementations of free modules: *FreeModule* and CombinatorialFreeModule. The distinction between the two is mostly an accident in history. The latter allows for the basis to be indexed by any kind of objects, instead of just $0, 1, 2, \ldots$. They also differ by feature set and efficiency. Eventually, both implementations will be merged under the name *FreeModule*. In the mean time, we focus here on CombinatorialFreeModule. We recommend to start by browsing its documentation:

```
sage: CombinatorialFreeModule?                # not tested
```

```
>>> from sage.all import *
>>> CombinatorialFreeModule?                  # not tested
```

1.1 Construction, arithmetic, and basic usage

We begin with a minimal example:

```
sage: G = Zmod(5)
sage: F = CombinatorialFreeModule(ZZ, G)
sage: F.an_element()
2*B[0] + 2*B[1] + 3*B[2]
```

```
>>> from sage.all import *
>>> G = Zmod(Integer(5))
>>> F = CombinatorialFreeModule(ZZ, G)
>>> F.an_element()
2*B[0] + 2*B[1] + 3*B[2]
```

F is the free module over the ring integers $\mathbf{Z}$ whose canonical basis is indexed by the set of integers modulo 5.

We can use any set, finite or not, to index the basis, as long as its elements are immutable. Here are some $\mathbf{Z}$-free modules; what is the indexing set for the basis in each example below?

```
sage: F = CombinatorialFreeModule(ZZ, CC); F.an_element()
B[1.00000000000000*I]
sage: F = CombinatorialFreeModule(ZZ, Partitions(NonNegativeIntegers(),             #␣
↪needs sage.combinat
....:                                           max_part=3)); F.an_element()
```

(continues on next page)

(continued from previous page)

```
2*B[[]] + 2*B[[1]] + 3*B[[2]]
sage: F = CombinatorialFreeModule(ZZ, ['spam', 'eggs', '42']); F.an_element()
3*B['42'] + 2*B['eggs'] + 2*B['spam']
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(ZZ, CC); F.an_element()
B[1.00000000000000*I]
>>> F = CombinatorialFreeModule(ZZ, Partitions(NonNegativeIntegers(),              #␣
↪needs sage.combinat
...                                                max_part=Integer(3))); F.an_element()
2*B[[]] + 2*B[[1]] + 3*B[[2]]
>>> F = CombinatorialFreeModule(ZZ, ['spam', 'eggs', '42']); F.an_element()
3*B['42'] + 2*B['eggs'] + 2*B['spam']
```

Note that we use '42' (and not the number 42) in order to ensure that all objects are comparable in a deterministic way, which allows the elements to be printed in a predictable manner. It is not mandatory that indices have such a stable ordering, but if they do not, then the elements may be displayed in some random order.

Lists are not hashable, and thus cannot be used to index the basis; instead one can use tuples:

```
sage: F = CombinatorialFreeModule(ZZ, ([1],[2],[3])); F.an_element()
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'

sage: F = CombinatorialFreeModule(ZZ, ((1,), (2,), (3,))); F.an_element()
2*B[(1,)] + 2*B[(2,)] + 3*B[(3,)]
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(ZZ, ([Integer(1)],[Integer(2)],[Integer(3)])); F.an_
↪element()
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'

>>> F = CombinatorialFreeModule(ZZ, ((Integer(1),), (Integer(2),), (Integer(3),))); F.
↪an_element()
2*B[(1,)] + 2*B[(2,)] + 3*B[(3,)]
```

The name of the basis can be customized:

```
sage: F = CombinatorialFreeModule(ZZ, Zmod(5), prefix='a'); F.an_element()
2*a[0] + 2*a[1] + 3*a[2]
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(ZZ, Zmod(Integer(5)), prefix='a'); F.an_element()
2*a[0] + 2*a[1] + 3*a[2]
```

Let us do some arithmetic with elements of A:

```
sage: f = F.an_element(); f
2*a[0] + 2*a[1] + 3*a[2]
```

(continues on next page)

(continued from previous page)

```
sage: 2*f
4*a[0] + 4*a[1] + 6*a[2]

sage: 2*f - f
2*a[0] + 2*a[1] + 3*a[2]
```

```
>>> from sage.all import *
>>> f = F.an_element(); f
2*a[0] + 2*a[1] + 3*a[2]

>>> Integer(2)*f
4*a[0] + 4*a[1] + 6*a[2]

>>> Integer(2)*f - f
2*a[0] + 2*a[1] + 3*a[2]
```

Inputing elements as they are output does not work by default:

```
sage: a[0] + 3*a[1]
Traceback (most recent call last):
...
NameError: name 'a' is not defined
```

```
>>> from sage.all import *
>>> a[Integer(0)] + Integer(3)*a[Integer(1)]
Traceback (most recent call last):
...
NameError: name 'a' is not defined
```

To enable this, we must first get the *canonical basis* for the module:

```
sage: a = F.basis(); a
Lazy family (Term map from Ring of integers modulo 5
             to Free module generated by Ring of integers modulo 5
             over Integer Ring(i))_{i in Ring of integers modulo 5}
```

```
>>> from sage.all import *
>>> a = F.basis(); a
Lazy family (Term map from Ring of integers modulo 5
             to Free module generated by Ring of integers modulo 5
             over Integer Ring(i))_{i in Ring of integers modulo 5}
```

This gadget models the `family` $(B_i)_{i \in \mathbf{Z}_5}$. In particular, one can run through its elements:

```
sage: list(a)
[a[0], a[1], a[2], a[3], a[4]]
```

```
>>> from sage.all import *
>>> list(a)
[a[0], a[1], a[2], a[3], a[4]]
```

recover its indexing set:

```
sage: a.keys()
Ring of integers modulo 5
```

```
>>> from sage.all import *
>>> a.keys()
Ring of integers modulo 5
```

or construct an element from the corresponding index:

```
sage: a[2]
a[2]
```

```
>>> from sage.all import *
>>> a[Integer(2)]
a[2]
```

So now we can do:

```
sage: a[0] + 3*a[1]
a[0] + 3*a[1]
```

```
>>> from sage.all import *
>>> a[Integer(0)] + Integer(3)*a[Integer(1)]
a[0] + 3*a[1]
```

which enables copy-pasting outputs as long as the prefix matches the name of the basis:

```
sage: 2*a[0] + 2*a[1] + 3*a[2] == f
True
```

```
>>> from sage.all import *
>>> Integer(2)*a[Integer(0)] + Integer(2)*a[Integer(1)] + Integer(3)*a[Integer(2)] ==␣
↪f
True
```

Be careful that the input is currently *not* checked:

```
sage: a['is'] + a['this'] + a['a'] + a['bug']
a['a'] + a['bug'] + a['is'] + a['this']
```

```
>>> from sage.all import *
>>> a['is'] + a['this'] + a['a'] + a['bug']
a['a'] + a['bug'] + a['is'] + a['this']
```

1.2 Manipulating free module elements

The elements of our module come with many methods for exploring and manipulating them:

```
sage: f.<tab>                                   # not tested
```

```
>>> from sage.all import *
>>> f.<tab>                                    # not tested
```

Some definitions:

- A *monomial* is an element of the basis B_i;
- A *term* is an element of the basis multiplied by a nonzero *coefficient*: cB_i;
- The support of that term is i.
- The corresponding *item* is the `tuple (i, c)`.
- The *support* of an element f is the collection of indices i such that B_i appears in f with nonzero coefficient.
- The *monomials*, *terms*, *items*, and *coefficients* of an element f are defined accordingly.
- *Leading*/*trailing* refers to the *greatest*/*least* index. Elements are printed starting with the *least* index (for lexicographic order by default).

Let us investigate those definitions on our example:

```
sage: f
2*a[0] + 2*a[1] + 3*a[2]
sage: f.leading_term()
3*a[2]
sage: f.leading_monomial()
a[2]
sage: f.leading_support()
2
sage: f.leading_coefficient()
3
sage: f.leading_item()
(2, 3)

sage: f.support()
SupportView({0: 2, 1: 2, 2: 3})
sage: f.monomials()
[a[0], a[1], a[2]]
sage: f.coefficients()
[2, 2, 3]
```

```
>>> from sage.all import *
>>> f
2*a[0] + 2*a[1] + 3*a[2]
>>> f.leading_term()
3*a[2]
>>> f.leading_monomial()
a[2]
>>> f.leading_support()
2
>>> f.leading_coefficient()
3
>>> f.leading_item()
(2, 3)
```

(continues on next page)

(continued from previous page)

```
>>> f.support()
SupportView({0: 2, 1: 2, 2: 3})
>>> f.monomials()
[a[0], a[1], a[2]]
>>> f.coefficients()
[2, 2, 3]
```

We can iterate through the items of an element:

```
sage: for index, coeff in f:
....:     print("The coefficient of a_{%s} is %s"%(index, coeff))
The coefficient of a_{0} is 2
The coefficient of a_{1} is 2
The coefficient of a_{2} is 3
```

```
>>> from sage.all import *
>>> for index, coeff in f:
...     print("The coefficient of a_{%s} is %s"%(index, coeff))
The coefficient of a_{0} is 2
The coefficient of a_{1} is 2
The coefficient of a_{2} is 3
```

This element can be thought of as a dictionary index->coefficient:

```
sage: f[0], f[1], f[2]
(2, 2, 3)
```

```
>>> from sage.all import *
>>> f[Integer(0)], f[Integer(1)], f[Integer(2)]
(2, 2, 3)
```

This dictionary can be accessed explicitly with the monomial_coefficients method:

```
sage: f.monomial_coefficients()
{0: 2, 1: 2, 2: 3}
```

```
>>> from sage.all import *
>>> f.monomial_coefficients()
{0: 2, 1: 2, 2: 3}
```

The `map` methods are useful to transform elements:

```
sage: f
2*a[0] + 2*a[1] + 3*a[2]
sage: f.map_support(lambda i: i+1)
2*a[1] + 2*a[2] + 3*a[3]
sage: f.map_coefficients(lambda c: c-3)
-a[0] - a[1]
sage: f.map_item(lambda i,c: (i+1,c-3))
-a[1] - a[2]
```

```
>>> from sage.all import *
>>> f
2*a[0] + 2*a[1] + 3*a[2]
>>> f.map_support(lambda i: i+Integer(1))
2*a[1] + 2*a[2] + 3*a[3]
>>> f.map_coefficients(lambda c: c-Integer(3))
-a[0] - a[1]
>>> f.map_item(lambda i,c: (i+Integer(1),c-Integer(3)))
-a[1] - a[2]
```

Note: this last function should be called `map_items`!

1.3 Manipulating free modules

The free module itself (A in our example) has several utility methods for constructing elements:

```
sage: F.zero()
0
sage: F.term(1)
a[1]
sage: F.sum_of_monomials(i for i in Zmod(5) if i > 2)
a[3] + a[4]
sage: F.sum_of_terms((i+1,i) for i in Zmod(5) if i > 2)
4*a[0] + 3*a[4]
sage: F.sum(ZZ(i)*a[i+1] for i in Zmod(5) if i > 2)  # Note coeff is not (currently)␣
↪implicitly coerced
4*a[0] + 3*a[4]
```

```
>>> from sage.all import *
>>> F.zero()
0
>>> F.term(Integer(1))
a[1]
>>> F.sum_of_monomials(i for i in Zmod(Integer(5)) if i > Integer(2))
a[3] + a[4]
>>> F.sum_of_terms((i+Integer(1),i) for i in Zmod(Integer(5)) if i > Integer(2))
4*a[0] + 3*a[4]
>>> F.sum(ZZ(i)*a[i+Integer(1)] for i in Zmod(Integer(5)) if i > Integer(2))  # Note␣
↪coeff is not (currently) implicitly coerced
4*a[0] + 3*a[4]
```

Is safer to use `F.sum()` than to use `sum()`: in case the input is an empty iterable, it makes sure the zero of A is returned, and not a plain 0:

```
sage: F.sum([]), parent(F.sum([]))
(0, Free module generated by Ring of integers modulo 5 over Integer Ring)
sage: sum([]),   parent(sum([]))
(0, <... 'int'>)
```

```
>>> from sage.all import *
>>> F.sum([]), parent(F.sum([]))
(0, Free module generated by Ring of integers modulo 5 over Integer Ring)
```

(continues on next page)

(continued from previous page)

```
>>> sum([]),   parent(sum([]))
(0, <... 'int'>)
```

Todo

Introduce echelon forms, submodules, quotients in the finite dimensional case

1.4 Review

In this tutorial we have seen how to construct vector spaces and free modules with a basis indexed by any kind of objects.

To learn how to endow such free modules with additional structure, define morphisms, or implement modules with several distinguished basis, see the Implementing Algebraic Structures thematic tutorial.

CHAPTER

TWO

FREE MODULES, SUBMODULES, AND QUOTIENTS

2.1 Abstract base class for modules

AUTHORS:

- William Stein: initial version
- Julian Rueth (2014-05-10): category parameter for Module, doc cleanup

EXAMPLES:

A minimal example of a module:

```
sage: from sage.structure.richcmp import richcmp
sage: class MyElement(sage.structure.element.ModuleElement):
....:     def __init__(self, parent, x):
....:         self.x = x
....:         sage.structure.element.ModuleElement.__init__(self, parent=parent)
....:     def _lmul_(self, c):
....:         return self.parent()(c*self.x)
....:     def _add_(self, other):
....:         return self.parent()(self.x + other.x)
....:     def _richcmp_(self, other, op):
....:         return richcmp(self.x, other.x, op)
....:     def __hash__(self):
....:         return hash(self.x)
....:     def _repr_(self):
....:         return repr(self.x)

sage: from sage.modules.module import Module
sage: class MyModule(Module):
....:     Element = MyElement
....:     def _element_constructor_(self, x):
....:         if isinstance(x, MyElement): x = x.x
....:         return self.element_class(self, self.base_ring()(x))
....:     def __eq__(self, other):
....:         if not isinstance(other, MyModule): return False
....:         return self.base_ring() == other.base_ring()
....:     def __hash__(self):
....:         return hash(self.base_ring())

sage: M = MyModule(QQ)
sage: M(1)
```

(continues on next page)

(continued from previous page)

```
1

sage: import __main__
sage: __main__.MyModule = MyModule
sage: __main__.MyElement = MyElement
sage: TestSuite(M).run()
```

```
>>> from sage.all import *
>>> from sage.structure.richcmp import richcmp
>>> class MyElement(sage.structure.element.ModuleElement):
...     def __init__(self, parent, x):
...         self.x = x
...         sage.structure.element.ModuleElement.__init__(self, parent=parent)
...     def _lmul_(self, c):
...         return self.parent()(c*self.x)
...     def _add_(self, other):
...         return self.parent()(self.x + other.x)
...     def _richcmp_(self, other, op):
...         return richcmp(self.x, other.x, op)
...     def __hash__(self):
...         return hash(self.x)
...     def _repr_(self):
...         return repr(self.x)

>>> from sage.modules.module import Module
>>> class MyModule(Module):
...     Element = MyElement
...     def _element_constructor_(self, x):
...         if isinstance(x, MyElement): x = x.x
...         return self.element_class(self, self.base_ring()(x))
...     def __eq__(self, other):
...         if not isinstance(other, MyModule): return False
...         return self.base_ring() == other.base_ring()
...     def __hash__(self):
...         return hash(self.base_ring())

>>> M = MyModule(QQ)
>>> M(Integer(1))
1

>>> import __main__
>>> __main__.MyModule = MyModule
>>> __main__.MyElement = MyElement
>>> TestSuite(M).run()
```

class sage.modules.module.**Module**

Bases: Parent

Generic module class.

INPUT:

- base – a ring; the base ring of the module

- `category` – a category (default: `None`), the category for this module. If `None`, then this is set to the category of modules/vector spaces over `base`.
- `names` – names of generators

EXAMPLES:

```
sage: from sage.modules.module import Module
sage: M = Module(ZZ)
sage: M.base_ring()
Integer Ring
sage: M.category()
Category of modules over Integer Ring
```

```
>>> from sage.all import *
>>> from sage.modules.module import Module
>>> M = Module(ZZ)
>>> M.base_ring()
Integer Ring
>>> M.category()
Category of modules over Integer Ring
```

Normally the category is set to the category of modules over `base`. If `base` is a field, then the category is the category of vector spaces over `base`:

```
sage: M_QQ = Module(QQ)
sage: M_QQ.category()
Category of vector spaces over Rational Field
```

```
>>> from sage.all import *
>>> M_QQ = Module(QQ)
>>> M_QQ.category()
Category of vector spaces over Rational Field
```

The `category` parameter can be used to set a more specific category:

```
sage: N = Module(ZZ, category=FiniteDimensionalModulesWithBasis(ZZ))
sage: N.category()
Category of finite dimensional modules with basis over Integer Ring
```

```
>>> from sage.all import *
>>> N = Module(ZZ, category=FiniteDimensionalModulesWithBasis(ZZ))
>>> N.category()
Category of finite dimensional modules with basis over Integer Ring
```

base_extend(*R*)

Return the base extension of `self` to R.

This is the same as `self.change_ring(R)` except that a `TypeError` is raised if there is no canonical coerce map from the base ring of `self` to R.

INPUT:

- `R` – ring

EXAMPLES:

```
sage: V = ZZ^7                                                              #␣
↪needs sage.modules
sage: V.base_extend(QQ)                                                     #␣
↪needs sage.modules
Vector space of dimension 7 over Rational Field
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(7)                                                      ␣
↪      # needs sage.modules
>>> V.base_extend(QQ)                                                     #␣
↪needs sage.modules
Vector space of dimension 7 over Rational Field
```

change_ring(*R*)

Return the base change of `self` to R.

EXAMPLES:

```
sage: from sage.modular.modform.space import ModularFormsSpace              #␣
↪needs sage.modular
sage: ModularFormsSpace(Gamma0(11), 2,                                      #␣
↪needs sage.modular sage.rings.finite_rings
....:                   DirichletGroup(1)[0], QQ).change_ring(GF(7))
Traceback (most recent call last):
...
NotImplementedError: the method change_ring() has not yet been implemented
```

```
>>> from sage.all import *
>>> from sage.modular.modform.space import ModularFormsSpace              #␣
↪needs sage.modular
>>> ModularFormsSpace(Gamma0(Integer(11)), Integer(2),                      ␣
↪              # needs sage.modular sage.rings.finite_rings
...                    DirichletGroup(Integer(1))[Integer(0)], QQ).change_
↪ring(GF(Integer(7)))
Traceback (most recent call last):
...
NotImplementedError: the method change_ring() has not yet been implemented
```

endomorphism_ring()

Return the endomorphism ring of this module in its category.

EXAMPLES:

```
sage: from sage.modules.module import Module
sage: M = Module(ZZ)
sage: M.endomorphism_ring()
Set of Morphisms
 from <sage.modules.module.Module object at ...>
   to <sage.modules.module.Module object at ...>
   in Category of modules over Integer Ring
```

```
>>> from sage.all import *
>>> from sage.modules.module import Module
```

(continues on next page)

(continued from previous page)

```
>>> M = Module(ZZ)
>>> M.endomorphism_ring()
Set of Morphisms
 from <sage.modules.module.Module object at ...>
   to <sage.modules.module.Module object at ...>
   in Category of modules over Integer Ring
```

sage.modules.module.**is_Module**(*x*)

Return `True` if `x` is a module, `False` otherwise.

INPUT:

- `x` – anything

EXAMPLES:

```
sage: from sage.modules.module import is_Module
sage: M = FreeModule(RationalField(),30)                                    #␣
↪needs sage.modules
sage: is_Module(M)                                                          #␣
↪needs sage.modules
doctest:warning...
DeprecationWarning: the function is_Module is deprecated;
use 'isinstance(..., Module)' instead
See https://github.com/sagemath/sage/issues/37924 for details.
True
sage: is_Module(10)
False
```

```
>>> from sage.all import *
>>> from sage.modules.module import is_Module
>>> M = FreeModule(RationalField(),Integer(30))                                ␣
↪     # needs sage.modules
>>> is_Module(M)                                                            #␣
↪needs sage.modules
doctest:warning...
DeprecationWarning: the function is_Module is deprecated;
use 'isinstance(..., Module)' instead
See https://github.com/sagemath/sage/issues/37924 for details.
True
>>> is_Module(Integer(10))
False
```

sage.modules.module.**is_VectorSpace**(*x*)

Return `True` if `x` is a vector space, `False` otherwise.

INPUT:

- `x` – anything

EXAMPLES:

```
sage: # needs sage.modules
sage: from sage.modules.module import is_Module, is_VectorSpace
sage: M = FreeModule(RationalField(),30)
```

(continues on next page)

(continued from previous page)

```
sage: is_VectorSpace(M)
doctest:warning...
DeprecationWarning: the function is_VectorSpace is deprecated;
use 'isinstance(..., Module)' and check the base ring instead
See https://github.com/sagemath/sage/issues/37924 for details.
True
sage: M = FreeModule(IntegerRing(),30)
sage: is_Module(M)
doctest:warning...
DeprecationWarning: the function is_Module is deprecated;
use 'isinstance(..., Module)' instead
See https://github.com/sagemath/sage/issues/37924 for details.
True
sage: is_VectorSpace(M)
False
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> from sage.modules.module import is_Module, is_VectorSpace
>>> M = FreeModule(RationalField(),Integer(30))
>>> is_VectorSpace(M)
doctest:warning...
DeprecationWarning: the function is_VectorSpace is deprecated;
use 'isinstance(..., Module)' and check the base ring instead
See https://github.com/sagemath/sage/issues/37924 for details.
True
>>> M = FreeModule(IntegerRing(),Integer(30))
>>> is_Module(M)
doctest:warning...
DeprecationWarning: the function is_Module is deprecated;
use 'isinstance(..., Module)' instead
See https://github.com/sagemath/sage/issues/37924 for details.
True
>>> is_VectorSpace(M)
False
```

2.2 Free modules

Sage supports computation with free modules over an arbitrary commutative ring. Nontrivial functionality is available over $\mathbf{Z}$, fields, and some principal ideal domains (e.g. $\mathbf{Q}[x]$ and rings of integers of number fields). All free modules over an integral domain are equipped with an embedding in an ambient vector space and an inner product, which you can specify and change.

Create the free module of rank n over an arbitrary commutative ring R using the command `FreeModule(R,n)`. Equivalently, `R^n` also creates that free module.

The following example illustrates the creation of both a vector space and a free module over the integers and a submodule of it. Use the functions `FreeModule`, `span` and member functions of free modules to create free modules. *Do not use the FreeModule_xxx constructors directly.*

EXAMPLES:

```
sage: V = VectorSpace(QQ, 3)
sage: W = V.subspace([[1,2,7], [1,1,0]])
sage: W
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -7]
[ 0  1  7]
sage: C = VectorSpaces(FiniteField(7))
sage: C
Category of vector spaces over Finite Field of size 7
sage: C(W)
Vector space of degree 3 and dimension 2 over Finite Field of size 7
Basis matrix:
[1 0 0]
[0 1 0]
```

```
>>> from sage.all import *
>>> V = VectorSpace(QQ, Integer(3))
>>> W = V.subspace([[Integer(1),Integer(2),Integer(7)], [Integer(1),Integer(1),
↪Integer(0)]])
>>> W
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -7]
[ 0  1  7]
>>> C = VectorSpaces(FiniteField(Integer(7)))
>>> C
Category of vector spaces over Finite Field of size 7
>>> C(W)
Vector space of degree 3 and dimension 2 over Finite Field of size 7
Basis matrix:
[1 0 0]
[0 1 0]
```

```
sage: M = ZZ^3
sage: C = VectorSpaces(FiniteField(7))
sage: C(M)
Vector space of dimension 3 over Finite Field of size 7
sage: W = M.submodule([[1,2,7], [8,8,0]])
sage: C(W)
Vector space of degree 3 and dimension 2 over Finite Field of size 7
Basis matrix:
[1 0 0]
[0 1 0]
```

```
>>> from sage.all import *
>>> M = ZZ**Integer(3)
>>> C = VectorSpaces(FiniteField(Integer(7)))
>>> C(M)
Vector space of dimension 3 over Finite Field of size 7
>>> W = M.submodule([[Integer(1),Integer(2),Integer(7)], [Integer(8),Integer(8),
↪Integer(0)]])
```

(continues on next page)

(continued from previous page)

```
>>> C(W)
Vector space of degree 3 and dimension 2 over Finite Field of size 7
Basis matrix:
[1 0 0]
[0 1 0]
```

We illustrate the exponent notation for creation of free modules.

```
sage: ZZ^4
Ambient free module of rank 4 over the principal ideal domain Integer Ring
sage: QQ^2
Vector space of dimension 2 over Rational Field
sage: RR^3
Vector space of dimension 3 over Real Field with 53 bits of precision
```

```
>>> from sage.all import *
>>> ZZ**Integer(4)
Ambient free module of rank 4 over the principal ideal domain Integer Ring
>>> QQ**Integer(2)
Vector space of dimension 2 over Rational Field
>>> RR**Integer(3)
Vector space of dimension 3 over Real Field with 53 bits of precision
```

Base ring:

```
sage: R.<x,y> = QQ[]
sage: M = FreeModule(R,2)
sage: M.base_ring()
Multivariate Polynomial Ring in x, y over Rational Field
```

```
>>> from sage.all import *
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> M = FreeModule(R,Integer(2))
>>> M.base_ring()
Multivariate Polynomial Ring in x, y over Rational Field
```

```
sage: VectorSpace(QQ, 10).base_ring()
Rational Field
```

```
>>> from sage.all import *
>>> VectorSpace(QQ, Integer(10)).base_ring()
Rational Field
```

Enumeration of $\mathbf{Z}^n$ happens in order of increasing 1-norm primarily and increasing ∞-norm secondarily:

```
sage: print([v for _,v in zip(range(31), ZZ^3)])
[(0, 0, 0),
 (1, 0, 0), (-1, 0, 0), (0, 1, 0), (0, -1, 0), (0, 0, 1), (0, 0, -1),
 (1, 1, 0), (-1, 1, 0), (1, -1, 0), (-1, -1, 0), (1, 0, 1), (-1, 0, 1), (1, 0, -1), (-
↪1, 0, -1), (0, 1, 1), (0, -1, 1), (0, 1, -1), (0, -1, -1),
 (2, 0, 0), (-2, 0, 0), (0, 2, 0), (0, -2, 0), (0, 0, 2), (0, 0, -2),
 (1, 1, 1), (-1, 1, 1), (1, -1, 1), (-1, -1, 1), (1, 1, -1), ...]
```

```
>>> from sage.all import *
>>> print([v for _,v in zip(range(Integer(31)), ZZ**Integer(3))])
[(0, 0, 0),
 (1, 0, 0), (-1, 0, 0), (0, 1, 0), (0, -1, 0), (0, 0, 1), (0, 0, -1),
 (1, 1, 0), (-1, 1, 0), (1, -1, 0), (-1, -1, 0), (1, 0, 1), (-1, 0, 1), (1, 0, -1), (-
↪1, 0, -1), (0, 1, 1), (0, -1, 1), (0, 1, -1), (0, -1, -1),
 (2, 0, 0), (-2, 0, 0), (0, 2, 0), (0, -2, 0), (0, 0, 2), (0, 0, -2),
 (1, 1, 1), (-1, 1, 1), (1, -1, 1), (-1, -1, 1), (1, 1, -1), ...]
```

For other infinite enumerated base rings (i.e., rings which are objects of the category `InfiniteEnumeratedSets`), a free module of rank r is enumerated by applying *`FreeModule_ambient.linear_combination_of_basis()`* to all vectors in $\mathbf{Z}^r$, enumerated in the way shown above.

AUTHORS:

- William Stein (2005, 2007)
- David Kohel (2007, 2008)
- Niles Johnson (2010-08): (Issue #3893) `random_element()` should pass on `*args` and `**kwds`.
- Simon King (2010-12): (Issue #8800) fixed a bug in `denominator()`.
- Simon King (2010-12), Peter Bruin (June 2014): (Issue #10513) new coercion model and category framework.

class sage.modules.free_module.**ComplexDoubleVectorSpace_class**(*n*)

Bases: *FreeModule_ambient_field*

coordinates(*v*)

class sage.modules.free_module.**EchelonMatrixKey**(*obj*)

Bases: `object`

A total ordering on free modules for sorting.

This class orders modules by their ambient spaces, then by dimension, then in order by their echelon matrices. If a function returns a list of free modules, this can be used to sort the output and thus render it deterministic.

INPUT:

- `obj` – a free module

EXAMPLES:

```
sage: V = span([[1,2,3], [5,6,7], [8,9,10]], QQ)
sage: W = span([[5,6,7], [8,9,10]], QQ)
sage: X = span([[5,6,7]], ZZ).scale(1/11)
sage: Y = CC^3
sage: Z = ZZ^2
sage: modules = [V,W,X,Y,Z]
sage: modules_sorted = [Z,X,V,W,Y]
sage: from sage.modules.free_module import EchelonMatrixKey
sage: modules.sort(key=EchelonMatrixKey)
sage: modules == modules_sorted
True
```

```
>>> from sage.all import *
>>> V = span([[Integer(1),Integer(2),Integer(3)], [Integer(5),Integer(6),
↪Integer(7)], [Integer(8),Integer(9),Integer(10)]], QQ)
```

(continues on next page)

(continued from previous page)

```
>>> W = span([[Integer(5),Integer(6),Integer(7)], [Integer(8),Integer(9),
↪Integer(10)]], QQ)
>>> X = span([[Integer(5),Integer(6),Integer(7)]], ZZ).scale(Integer(1)/
↪Integer(11))
>>> Y = CC**Integer(3)
>>> Z = ZZ**Integer(2)
>>> modules = [V,W,X,Y,Z]
>>> modules_sorted = [Z,X,V,W,Y]
>>> from sage.modules.free_module import EchelonMatrixKey
>>> modules.sort(key=EchelonMatrixKey)
>>> modules == modules_sorted
True
```

sage.modules.free_module.**FreeModule**(*base_ring*, *rank_or_basis_keys*, *sparse*, *inner_product_matrix*, *with_basis=None*, *rank=False*, *basis_keys=None*, ***args*)

Create a free module over the given commutative `base_ring`.

`FreeModule` can be called with the following positional arguments:

- `FreeModule(base_ring, rank, ...)`
- `FreeModule(base_ring, basis_keys, ...)`

INPUT:

- `base_ring` – a commutative ring
- `rank` – nonnegative integer
- `basis_keys` – a finite or enumerated family of arbitrary objects
- `sparse` – boolean (default: `False`)
- `inner_product_matrix` – the inner product matrix (default: `None`)
- `with_basis` – either `'standard'` (the default), in which case a free module with the standard basis as the distinguished basis is created; or `None`, in which case a free module without distinguished basis is created.
- further options may be accepted by various implementation classes

OUTPUT: a free module

This factory function creates instances of various specialized classes depending on the input. Not all combinations of options are implemented.

- If the parameter `basis_keys` is provided, it must be a finite or enumerated family of objects, and an instance of `CombinatorialFreeModule` is created.

  EXAMPLES:

  ```
  sage: CombinatorialFreeModule(QQ, ['a','b','c'])
  Free module generated by {'a', 'b', 'c'} over Rational Field
  ```

  ```
  >>> from sage.all import *
  >>> CombinatorialFreeModule(QQ, ['a','b','c'])
  Free module generated by {'a', 'b', 'c'} over Rational Field
  ```

  It has a distinguished standard basis that is indexed by the provided `basis_keys`. See the documentation of `CombinatorialFreeModule` for more examples and details, including its `UniqueRepresentation` semantics.

- If the parameter `with_basis` is set to `None`, then a free module of the given `rank` without distinguished basis is created. It is represented by an instance of `FiniteRankFreeModule`.

  EXAMPLES:

```
sage: FiniteRankFreeModule(ZZ, 3, name='M')
Rank-3 free module M over the Integer Ring
```

```
>>> from sage.all import *
>>> FiniteRankFreeModule(ZZ, Integer(3), name='M')
Rank-3 free module M over the Integer Ring
```

  See the documentation of `FiniteRankFreeModule` for more options, examples, and details.

- If `rank` is provided and the option `with_basis` is left at its default value, `'standard'`, then a free ambient module with distinguished standard basis indexed by `range(rank)` is created. There is only one dense and one sparse free ambient module of given `rank` over `base_ring`.

  EXAMPLES:

```
sage: FreeModule(Integers(8), 10)
Ambient free module of rank 10 over Ring of integers modulo 8
```

```
>>> from sage.all import *
>>> FreeModule(Integers(Integer(8)), Integer(10))
Ambient free module of rank 10 over Ring of integers modulo 8
```

  The remainder of this documentation discusses this case of free ambient modules.

EXAMPLES:

First we illustrate creating free modules over various base fields. The base field affects the free module that is created. For example, free modules over a field are vector spaces, and free modules over a principal ideal domain are special in that more functionality is available for them than for completely general free modules.

```
sage: FreeModule(QQ,10)
Vector space of dimension 10 over Rational Field
sage: FreeModule(ZZ,10)
Ambient free module of rank 10 over the principal ideal domain Integer Ring
sage: FreeModule(FiniteField(5), 10)
Vector space of dimension 10 over Finite Field of size 5
sage: FreeModule(Integers(7), 10)
Vector space of dimension 10 over Ring of integers modulo 7
sage: FreeModule(PolynomialRing(QQ,'x'), 5)
Ambient free module of rank 5 over the principal ideal domain
 Univariate Polynomial Ring in x over Rational Field
sage: FreeModule(PolynomialRing(ZZ,'x'), 5)
Ambient free module of rank 5 over the integral domain
 Univariate Polynomial Ring in x over Integer Ring
```

```
>>> from sage.all import *
>>> FreeModule(QQ,Integer(10))
Vector space of dimension 10 over Rational Field
>>> FreeModule(ZZ,Integer(10))
Ambient free module of rank 10 over the principal ideal domain Integer Ring
```

(continues on next page)

(continued from previous page)

```
>>> FreeModule(FiniteField(Integer(5)), Integer(10))
Vector space of dimension 10 over Finite Field of size 5
>>> FreeModule(Integers(Integer(7)), Integer(10))
Vector space of dimension 10 over Ring of integers modulo 7
>>> FreeModule(PolynomialRing(QQ,'x'), Integer(5))
Ambient free module of rank 5 over the principal ideal domain
 Univariate Polynomial Ring in x over Rational Field
>>> FreeModule(PolynomialRing(ZZ,'x'), Integer(5))
Ambient free module of rank 5 over the integral domain
 Univariate Polynomial Ring in x over Integer Ring
```

Of course we can make rank 0 free modules:

```
sage: FreeModule(RealField(100),0)
Vector space of dimension 0 over Real Field with 100 bits of precision
```

```
>>> from sage.all import *
>>> FreeModule(RealField(Integer(100)),Integer(0))
Vector space of dimension 0 over Real Field with 100 bits of precision
```

Next we create a free module with sparse representation of elements. Functionality with sparse modules is *identical* to dense modules, but they may use less memory and arithmetic may be faster (or slower!).

```
sage: M = FreeModule(ZZ,200,sparse=True)
sage: M.is_sparse()
True
sage: type(M.0)
<class 'sage.modules.free_module_element.FreeModuleElement_generic_sparse'>
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ,Integer(200),sparse=True)
>>> M.is_sparse()
True
>>> type(M.gen(0))
<class 'sage.modules.free_module_element.FreeModuleElement_generic_sparse'>
```

The default is dense.

```
sage: M = ZZ^200
sage: type(M.0)
<class 'sage.modules.vector_integer_dense.Vector_integer_dense'>
```

```
>>> from sage.all import *
>>> M = ZZ**Integer(200)
>>> type(M.gen(0))
<class 'sage.modules.vector_integer_dense.Vector_integer_dense'>
```

Note that matrices associated in some way to sparse free modules are sparse by default:

```
sage: M = FreeModule(Integers(8), 2)
sage: A = M.basis_matrix()
sage: A.is_sparse()
```

(continues on next page)

(continued from previous page)

```
False
sage: Ms = FreeModule(Integers(8), 2, sparse=True)
sage: M == Ms  # as mathematical objects they are equal
True
sage: Ms.basis_matrix().is_sparse()
True
```

```
>>> from sage.all import *
>>> M = FreeModule(Integers(Integer(8)), Integer(2))
>>> A = M.basis_matrix()
>>> A.is_sparse()
False
>>> Ms = FreeModule(Integers(Integer(8)), Integer(2), sparse=True)
>>> M == Ms  # as mathematical objects they are equal
True
>>> Ms.basis_matrix().is_sparse()
True
```

We can also specify an inner product matrix, which is used when computing inner products of elements.

```
sage: A = MatrixSpace(ZZ, 2)([[1,0], [0,-1]])
sage: M = FreeModule(ZZ, 2, inner_product_matrix=A)
sage: v, w = M.gens()
sage: v.inner_product(w)
0
sage: v.inner_product(v)
1
sage: w.inner_product(w)
-1
sage: (v+2*w).inner_product(w)
-2
```

```
>>> from sage.all import *
>>> A = MatrixSpace(ZZ, Integer(2))([[Integer(1),Integer(0)], [Integer(0),-
↪Integer(1)]])
>>> M = FreeModule(ZZ, Integer(2), inner_product_matrix=A)
>>> v, w = M.gens()
>>> v.inner_product(w)
0
>>> v.inner_product(v)
1
>>> w.inner_product(w)
-1
>>> (v+Integer(2)*w).inner_product(w)
-2
```

You can also specify the inner product matrix by giving anything that coerces to an appropriate matrix. This is only useful if the inner product matrix takes values in the base ring.

```
sage: FreeModule(ZZ, 2, inner_product_matrix=1).inner_product_matrix()
[1 0]
[0 1]
```

(continues on next page)

(continued from previous page)

```
sage: FreeModule(ZZ, 2, inner_product_matrix=[1,2,3,4]).inner_product_matrix()
[1 2]
[3 4]
sage: FreeModule(ZZ, 2, inner_product_matrix=[[1,2], [3,4]]).inner_product_
↪matrix()
[1 2]
[3 4]
```

```
>>> from sage.all import *
>>> FreeModule(ZZ, Integer(2), inner_product_matrix=Integer(1)).inner_product_
↪matrix()
[1 0]
[0 1]
>>> FreeModule(ZZ, Integer(2), inner_product_matrix=[Integer(1),Integer(2),
↪Integer(3),Integer(4)]).inner_product_matrix()
[1 2]
[3 4]
>>> FreeModule(ZZ, Integer(2), inner_product_matrix=[[Integer(1),Integer(2)],␣
↪[Integer(3),Integer(4)]]).inner_product_matrix()
[1 2]
[3 4]
```

Todo

Refactor modules such that it only counts what category the base ring belongs to, but not what is its Python class.

EXAMPLES:

```
sage: FreeModule(QQ, ['a', 'b', 'c'])
Free module generated by {'a', 'b', 'c'} over Rational Field
sage: _.category()
Category of finite dimensional vector spaces with basis over Rational Field

sage: FreeModule(QQ, 3, with_basis=None)
3-dimensional vector space over the Rational Field
sage: _.category()
Category of finite dimensional vector spaces over Rational Field

sage: FreeModule(QQ, [1, 2, 3, 4], with_basis=None)
4-dimensional vector space over the Rational Field
sage: _.category()
Category of finite dimensional vector spaces over Rational Field
```

```
>>> from sage.all import *
>>> FreeModule(QQ, ['a', 'b', 'c'])
Free module generated by {'a', 'b', 'c'} over Rational Field
>>> _.category()
Category of finite dimensional vector spaces with basis over Rational Field
```

(continues on next page)

(continued from previous page)

```
>>> FreeModule(QQ, Integer(3), with_basis=None)
3-dimensional vector space over the Rational Field
>>> _.category()
Category of finite dimensional vector spaces over Rational Field

>>> FreeModule(QQ, [Integer(1), Integer(2), Integer(3), Integer(4)], with_
↪basis=None)
4-dimensional vector space over the Rational Field
>>> _.category()
Category of finite dimensional vector spaces over Rational Field
```

class sage.modules.free_module.**FreeModuleFactory**

Bases: UniqueFactory

Factory class for the finite-dimensional free modules with standard basis

create_key(*base_ring*, *rank*, *sparse=False*, *inner_product_matrix=None*)

create_object(*version*, *key*)

class sage.modules.free_module.**FreeModule_ambient**(*base_ring*, *rank*, *sparse=False*, *coordinate_ring=None*, *category=None*)

Bases: *FreeModule_generic*

Ambient free module over a commutative ring.

ambient_module()

Return `self`, since `self` is ambient.

EXAMPLES:

```
sage: A = QQ^5; A.ambient_module()
Vector space of dimension 5 over Rational Field
sage: A = ZZ^5; A.ambient_module()
Ambient free module of rank 5 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> A = QQ**Integer(5); A.ambient_module()
Vector space of dimension 5 over Rational Field
>>> A = ZZ**Integer(5); A.ambient_module()
Ambient free module of rank 5 over the principal ideal domain Integer Ring
```

basis()

Return a basis for this ambient free module.

OUTPUT:

- `Sequence` – an immutable sequence with universe this ambient free module

EXAMPLES:

```
sage: A = ZZ^3; B = A.basis(); B
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
sage: B.universe()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> A = ZZ**Integer(3); B = A.basis(); B
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
>>> B.universe()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

change_ring(*R*)

Return the ambient free module over `R` of the same rank as `self`.

This also preserves the sparsity.

EXAMPLES:

```
sage: A = ZZ^3; A.change_ring(QQ)
Vector space of dimension 3 over Rational Field
sage: A = ZZ^3; A.change_ring(GF(5))
Vector space of dimension 3 over Finite Field of size 5
```

```
>>> from sage.all import *
>>> A = ZZ**Integer(3); A.change_ring(QQ)
Vector space of dimension 3 over Rational Field
>>> A = ZZ**Integer(3); A.change_ring(GF(Integer(5)))
Vector space of dimension 3 over Finite Field of size 5
```

For ambient modules any change of rings is defined:

```
sage: A = GF(5)**3; A.change_ring(QQ)
Vector space of dimension 3 over Rational Field
```

```
>>> from sage.all import *
>>> A = GF(Integer(5))**Integer(3); A.change_ring(QQ)
Vector space of dimension 3 over Rational Field
```

coordinate_vector(*v*, *check=True*)

Write v in terms of the standard basis for `self` and return the resulting coefficients in a vector over the fraction field of the base ring.

Returns a vector c such that if B is the basis for self, then

$$\sum c_i B_i = v.$$

If v is not in self, raise an `ArithmeticError` exception.

EXAMPLES:

```
sage: V = Integers(16)^3
sage: v = V.coordinate_vector([1,5,9]); v
(1, 5, 9)
sage: v.parent()
Ambient free module of rank 3 over Ring of integers modulo 16
```

```
>>> from sage.all import *
>>> V = Integers(Integer(16))**Integer(3)
>>> v = V.coordinate_vector([Integer(1),Integer(5),Integer(9)]); v
```

(continues on next page)

(continued from previous page)

```
(1, 5, 9)
>>> v.parent()
Ambient free module of rank 3 over Ring of integers modulo 16
```

echelon_coordinate_vector(*v*, *check=True*)

Same as `self.coordinate_vector(v)`, since `self` is an ambient free module.

INPUT:

- `v` – vector
- `check` – boolean (default: `True`); if `True`, also verify that v is really in `self`

OUTPUT: list

EXAMPLES:

```
sage: V = QQ^4
sage: v = V([-1/2,1/2,-1/2,1/2])
sage: v
(-1/2, 1/2, -1/2, 1/2)
sage: V.coordinate_vector(v)
(-1/2, 1/2, -1/2, 1/2)
sage: V.echelon_coordinate_vector(v)
(-1/2, 1/2, -1/2, 1/2)
sage: W = V.submodule_with_basis([[1/2,1/2,1/2,1/2],[1,0,1,0]])
sage: W.coordinate_vector(v)
(1, -1)
sage: W.echelon_coordinate_vector(v)
(-1/2, 1/2)
```

```
>>> from sage.all import *
>>> V = QQ**Integer(4)
>>> v = V([-Integer(1)/Integer(2),Integer(1)/Integer(2),-Integer(1)/
↪Integer(2),Integer(1)/Integer(2)])
>>> v
(-1/2, 1/2, -1/2, 1/2)
>>> V.coordinate_vector(v)
(-1/2, 1/2, -1/2, 1/2)
>>> V.echelon_coordinate_vector(v)
(-1/2, 1/2, -1/2, 1/2)
>>> W = V.submodule_with_basis([[Integer(1)/Integer(2),Integer(1)/Integer(2),
↪Integer(1)/Integer(2),Integer(1)/Integer(2)],[Integer(1),Integer(0),
↪Integer(1),Integer(0)]])
>>> W.coordinate_vector(v)
(1, -1)
>>> W.echelon_coordinate_vector(v)
(-1/2, 1/2)
```

echelon_coordinates(*v*, *check=True*)

Return the coordinate vector of v in terms of the echelon basis for `self`.

EXAMPLES:

```
sage: U = VectorSpace(QQ,3)
sage: [ U.coordinates(v) for v in U.basis() ]
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
sage: [ U.echelon_coordinates(v) for v in U.basis() ]
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
sage: V = U.submodule([[1,1,0],[0,1,1]])
sage: V
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  1]
sage: [ V.coordinates(v) for v in V.basis() ]
[[1, 0], [0, 1]]
sage: [ V.echelon_coordinates(v) for v in V.basis() ]
[[1, 0], [0, 1]]
sage: W = U.submodule_with_basis([[1,1,0],[0,1,1]])
sage: W
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[1 1 0]
[0 1 1]
sage: [ W.coordinates(v) for v in W.basis() ]
[[1, 0], [0, 1]]
sage: [ W.echelon_coordinates(v) for v in W.basis() ]
[[1, 1], [0, 1]]
```

```
>>> from sage.all import *
>>> U = VectorSpace(QQ,Integer(3))
>>> [ U.coordinates(v) for v in U.basis() ]
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
>>> [ U.echelon_coordinates(v) for v in U.basis() ]
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
>>> V = U.submodule([[Integer(1),Integer(1),Integer(0)],[Integer(0),
↪Integer(1),Integer(1)]])
>>> V
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  1]
>>> [ V.coordinates(v) for v in V.basis() ]
[[1, 0], [0, 1]]
>>> [ V.echelon_coordinates(v) for v in V.basis() ]
[[1, 0], [0, 1]]
>>> W = U.submodule_with_basis([[Integer(1),Integer(1),Integer(0)],
↪[Integer(0),Integer(1),Integer(1)]])
>>> W
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[1 1 0]
[0 1 1]
>>> [ W.coordinates(v) for v in W.basis() ]
[[1, 0], [0, 1]]
```

(continues on next page)

(continued from previous page)

```
>>> [ W.echelon_coordinates(v) for v in W.basis() ]
[[1, 1], [0, 1]]
```

echelonized_basis()

Return a basis for this ambient free module in echelon form.

EXAMPLES:

```
sage: A = ZZ^3; A.echelonized_basis()
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
```

```
>>> from sage.all import *
>>> A = ZZ**Integer(3); A.echelonized_basis()
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
```

echelonized_basis_matrix()

The echelonized basis matrix of `self`.

EXAMPLES:

```
sage: V = ZZ^4
sage: W = V.submodule([ V.gen(i)-V.gen(0) for i in range(1,4) ])
sage: W.basis_matrix()
[ 1  0  0 -1]
[ 0  1  0 -1]
[ 0  0  1 -1]
sage: W.echelonized_basis_matrix()
[ 1  0  0 -1]
[ 0  1  0 -1]
[ 0  0  1 -1]
sage: U = V.submodule_with_basis([ V.gen(i)-V.gen(0) for i in range(1,4) ])
sage: U.basis_matrix()
[-1  1  0  0]
[-1  0  1  0]
[-1  0  0  1]
sage: U.echelonized_basis_matrix()
[ 1  0  0 -1]
[ 0  1  0 -1]
[ 0  0  1 -1]
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(4)
>>> W = V.submodule([ V.gen(i)-V.gen(Integer(0)) for i in range(Integer(1),
↪Integer(4)) ])
>>> W.basis_matrix()
[ 1  0  0 -1]
[ 0  1  0 -1]
[ 0  0  1 -1]
>>> W.echelonized_basis_matrix()
[ 1  0  0 -1]
[ 0  1  0 -1]
[ 0  0  1 -1]
```

(continues on next page)

(continued from previous page)

```
>>> U = V.submodule_with_basis([ V.gen(i)-V.gen(Integer(0)) for i in␣
↪range(Integer(1),Integer(4)) ])
>>> U.basis_matrix()
[-1  1  0  0]
[-1  0  1  0]
[-1  0  0  1]
>>> U.echelonized_basis_matrix()
[ 1  0  0 -1]
[ 0  1  0 -1]
[ 0  0  1 -1]
```

gen(*i=0*)

Return the i-th generator for `self`.

Here i is between 0 and rank - 1, inclusive.

INPUT:

- `i` – integer (default: 0)

OUTPUT: i-th basis vector for `self`

EXAMPLES:

```
sage: n = 5
sage: V = QQ^n
sage: B = [V.gen(i) for i in range(n)]
sage: B
[(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: V.gens() == tuple(B)
True
```

```
>>> from sage.all import *
>>> n = Integer(5)
>>> V = QQ**n
>>> B = [V.gen(i) for i in range(n)]
>>> B
[(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)]
>>> V.gens() == tuple(B)
True
```

is_ambient()

Return `True` since this module is an ambient module.

EXAMPLES:

```
sage: A = QQ^5; A.is_ambient()
True
sage: A = (QQ^5).span([[1,2,3,4,5]]); A.is_ambient()
False
```

```
>>> from sage.all import *
>>> A = QQ**Integer(5); A.is_ambient()
True
>>> A = (QQ**Integer(5)).span([[Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5)]]); A.is_ambient()
False
```

linear_combination_of_basis(*v*)

Return the linear combination of the basis for `self` obtained from the elements of the list v.

INPUT:

- `v` – list

EXAMPLES:

```
sage: V = span([[1,2,3], [4,5,6]], ZZ)
sage: V
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[1 2 3]
[0 3 6]
sage: V.linear_combination_of_basis([1,1])
(1, 5, 9)
```

```
>>> from sage.all import *
>>> V = span([[Integer(1),Integer(2),Integer(3)], [Integer(4),Integer(5),
↪Integer(6)]], ZZ)
>>> V
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[1 2 3]
[0 3 6]
>>> V.linear_combination_of_basis([Integer(1),Integer(1)])
(1, 5, 9)
```

This should raise an error if the resulting element is not in self:

```
sage: W = span([[2,4]], ZZ)
sage: W.linear_combination_of_basis([1/2])
Traceback (most recent call last):
...
TypeError: element [1, 2] is not in free module
```

```
>>> from sage.all import *
>>> W = span([[Integer(2),Integer(4)]], ZZ)
>>> W.linear_combination_of_basis([Integer(1)/Integer(2)])
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
TypeError: element [1, 2] is not in free module
```

random_element(*prob=1.0*, **args*, ***kwds*)

Return a random element of `self`.

INPUT:

- `prob` – float. Each coefficient will be set to zero with probability $1 - prob$. Otherwise coefficients will be chosen randomly from base ring (and may be zero).
- `*args`, `**kwds` – passed on to random_element function of base ring

EXAMPLES:

```
sage: M = FreeModule(ZZ, 3)
sage: M.random_element().parent() is M
True
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ, Integer(3))
>>> M.random_element().parent() is M
True
```

Passes extra positional or keyword arguments through:

```
sage: all(i in range(5, 10) for i in M.random_element(1.0, 5, 10))
True
```

```
>>> from sage.all import *
>>> all(i in range(Integer(5), Integer(10)) for i in M.random_
↪element(RealNumber('1.0'), Integer(5), Integer(10)))
True
```

```
sage: M = FreeModule(ZZ, 16)
sage: M.random_element().parent() is M
True

sage: def add_sample(**kwds):
....:     global total, zeros
....:     v = M.random_element(**kwds)
....:     total += M.rank()
....:     zeros += sum(i == 0 for i in v)

sage: total = 0
sage: zeros = 0
sage: add_sample()
sage: expected = 1/5
sage: while abs(zeros/total - expected) > 0.01:
....:     add_sample()

sage: total = 0
sage: zeros = 0
```

(continues on next page)

(continued from previous page)

```
sage: add_sample(prob=0.3)
sage: expected = 1/5 * 3/10 + 7/10
sage: while abs(zeros/total - expected) > 0.01:
....:     add_sample(prob=0.3)

sage: total = 0
sage: zeros = 0
sage: add_sample(prob=0.7)
sage: expected = 1/5 * 7/10 + 3/10
sage: while abs(zeros/total - expected) > 0.01:
....:     add_sample(prob=0.7)
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ, Integer(16))
>>> M.random_element().parent() is M
True

>>> def add_sample(**kwds):
...     global total, zeros
...     v = M.random_element(**kwds)
...     total += M.rank()
...     zeros += sum(i == Integer(0) for i in v)

>>> total = Integer(0)
>>> zeros = Integer(0)
>>> add_sample()
>>> expected = Integer(1)/Integer(5)
>>> while abs(zeros/total - expected) > RealNumber('0.01'):
...     add_sample()

>>> total = Integer(0)
>>> zeros = Integer(0)
>>> add_sample(prob=RealNumber('0.3'))
>>> expected = Integer(1)/Integer(5) * Integer(3)/Integer(10) + Integer(7)/
↪Integer(10)
>>> while abs(zeros/total - expected) > RealNumber('0.01'):
...     add_sample(prob=RealNumber('0.3'))

>>> total = Integer(0)
>>> zeros = Integer(0)
>>> add_sample(prob=RealNumber('0.7'))
>>> expected = Integer(1)/Integer(5) * Integer(7)/Integer(10) + Integer(3)/
↪Integer(10)
>>> while abs(zeros/total - expected) > RealNumber('0.01'):
...     add_sample(prob=RealNumber('0.7'))
```

class sage.modules.free_module.**FreeModule_ambient_domain**(*base_ring*, *rank*, *sparse=False*, *coordinate_ring=None*, *category=None*)

Bases: *FreeModule_generic_domain*, *FreeModule_ambient*

Ambient free module over an integral domain.

EXAMPLES:

```
sage: FreeModule(PolynomialRing(GF(5), 'x'), 3)
Ambient free module of rank 3 over the principal ideal domain
 Univariate Polynomial Ring in x over Finite Field of size 5
```

```
>>> from sage.all import *
>>> FreeModule(PolynomialRing(GF(Integer(5)), 'x'), Integer(3))
Ambient free module of rank 3 over the principal ideal domain
 Univariate Polynomial Ring in x over Finite Field of size 5
```

ambient_vector_space()

Return the ambient vector space, which is this free module tensored with its fraction field.

EXAMPLES:

```
sage: M = ZZ^3
sage: V = M.ambient_vector_space(); V
Vector space of dimension 3 over Rational Field
```

```
>>> from sage.all import *
>>> M = ZZ**Integer(3)
>>> V = M.ambient_vector_space(); V
Vector space of dimension 3 over Rational Field
```

If an inner product on the module is specified, then this is preserved on the ambient vector space.

```
sage: N = FreeModule(ZZ,4,inner_product_matrix=1)
sage: U = N.ambient_vector_space()
sage: U
Ambient quadratic space of dimension 4 over Rational Field
Inner product matrix:
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
sage: P = N.submodule_with_basis([[1,-1,0,0],[0,1,-1,0],[0,0,1,-1]])
sage: P.gram_matrix()
[ 2 -1  0]
[-1  2 -1]
[ 0 -1  2]
sage: U == N.ambient_vector_space()
True
sage: U == V
False
```

```
>>> from sage.all import *
>>> N = FreeModule(ZZ,Integer(4),inner_product_matrix=Integer(1))
>>> U = N.ambient_vector_space()
>>> U
Ambient quadratic space of dimension 4 over Rational Field
Inner product matrix:
[1 0 0 0]
[0 1 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 0 1 0]
[0 0 0 1]
>>> P = N.submodule_with_basis([[Integer(1),-Integer(1),Integer(0),
↪Integer(0)],[Integer(0),Integer(1),-Integer(1),Integer(0)],[Integer(0),
↪Integer(0),Integer(1),-Integer(1)]])
>>> P.gram_matrix()
[ 2 -1  0]
[-1  2 -1]
[ 0 -1  2]
>>> U == N.ambient_vector_space()
True
>>> U == V
False
```

coordinate_vector(*v*, *check=True*)

Write v in terms of the standard basis for `self` and return the resulting coefficients in a vector over the fraction field of the base ring.

INPUT:

- `v` – vector
- `check` – boolean (default: `True`); if `True`, also verify that v is really in `self`

OUTPUT: list

The output is a vector c such that if B is the basis for `self`, then

$$\sum c_i B_i = v.$$

If v is not in self, raise an `ArithmeticError` exception.

EXAMPLES:

```
sage: V = ZZ^3
sage: v = V.coordinate_vector([1,5,9]); v
(1, 5, 9)
sage: v.parent()
Vector space of dimension 3 over Rational Field
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(3)
>>> v = V.coordinate_vector([Integer(1),Integer(5),Integer(9)]); v
(1, 5, 9)
>>> v.parent()
Vector space of dimension 3 over Rational Field
```

vector_space(*base_field=None*)

Return the vector space obtained from `self` by tensoring with the fraction field of the base ring and extending to the field.

EXAMPLES:

```
sage: M = ZZ^3;  M.vector_space()
Vector space of dimension 3 over Rational Field
```

```
>>> from sage.all import *
>>> M = ZZ**Integer(3);  M.vector_space()
Vector space of dimension 3 over Rational Field
```

class sage.modules.free_module.**FreeModule_ambient_field**(*base_field*, *dimension*, *sparse=False*, *category=None*)

Bases: *FreeModule_generic_field*, *FreeModule_ambient_pid*

Create the ambient vector space of given dimension over the given field.

INPUT:

- `base_field` – a field
- `dimension` – nonnegative integer
- `sparse` – boolean (default: `False`)

EXAMPLES:

```
sage: QQ^3
Vector space of dimension 3 over Rational Field
```

```
>>> from sage.all import *
>>> QQ**Integer(3)
Vector space of dimension 3 over Rational Field
```

ambient_vector_space()

Return `self` as the ambient vector space.

EXAMPLES:

```
sage: M = QQ^3
sage: M.ambient_vector_space()
Vector space of dimension 3 over Rational Field
```

```
>>> from sage.all import *
>>> M = QQ**Integer(3)
>>> M.ambient_vector_space()
Vector space of dimension 3 over Rational Field
```

base_field()

Return the base field of this vector space.

EXAMPLES:

```
sage: M = QQ^3
sage: M.base_field()
Rational Field
```

```
>>> from sage.all import *
>>> M = QQ**Integer(3)
>>> M.base_field()
Rational Field
```

class sage.modules.free_module.**FreeModule_ambient_pid**(*base_ring*, *rank*, *sparse=False*, *coordinate_ring=None*, *category=None*)

Bases: *FreeModule_generic_pid*, *FreeModule_ambient_domain*

Ambient free module over a principal ideal domain.

class sage.modules.free_module.**FreeModule_generic**(*base_ring*, *rank*, *degree*, *sparse=False*, *coordinate_ring=None*, *category=None*)

Bases: *Module_free_ambient*

Base class for all free modules.

INPUT:

- `base_ring` – a commutative ring
- `rank` – nonnegative integer
- `degree` – nonnegative integer
- `sparse` – boolean (default: `False`)
- `coordinate_ring` – a ring containing `base_ring` (default: equal to `base_ring`)
- `category` – category (default: `None`)

If `base_ring` is a field, then the default category is the category of finite-dimensional vector spaces over that field; otherwise it is the category of finite-dimensional free modules over that ring. In addition, the category is intersected with the category of finite enumerated sets if the ring is finite or the rank is 0.

EXAMPLES:

```
sage: PolynomialRing(QQ,3,'x')^3
Ambient free module of rank 3 over the integral domain
 Multivariate Polynomial Ring in x0, x1, x2 over Rational Field

sage: FreeModule(GF(7), 3).category()
Category of enumerated finite dimensional vector spaces with basis over
 (finite enumerated fields and subquotients of monoids and quotients of␣
↪semigroups)
sage: V = QQ^4; V.category()
Category of finite dimensional vector spaces with basis over
 (number fields and quotient fields and metric spaces)
sage: V = GF(5)**20; V.category()
Category of enumerated finite dimensional vector spaces with basis over
 (finite enumerated fields and subquotients of monoids and quotients of␣
↪semigroups)
sage: FreeModule(ZZ,3).category()
Category of finite dimensional modules with basis over
 (Dedekind domains and euclidean domains
  and noetherian rings and infinite enumerated sets
  and metric spaces)
sage: (QQ^0).category()
Category of finite enumerated finite dimensional vector spaces with basis
 over (number fields and quotient fields and metric spaces)
```

```
>>> from sage.all import *
>>> PolynomialRing(QQ,Integer(3),'x')**Integer(3)
```

(continues on next page)

(continued from previous page)

```
Ambient free module of rank 3 over the integral domain
 Multivariate Polynomial Ring in x0, x1, x2 over Rational Field

>>> FreeModule(GF(Integer(7)), Integer(3)).category()
Category of enumerated finite dimensional vector spaces with basis over
 (finite enumerated fields and subquotients of monoids and quotients of␣
↪semigroups)
>>> V = QQ**Integer(4); V.category()
Category of finite dimensional vector spaces with basis over
 (number fields and quotient fields and metric spaces)
>>> V = GF(Integer(5))**Integer(20); V.category()
Category of enumerated finite dimensional vector spaces with basis over
 (finite enumerated fields and subquotients of monoids and quotients of␣
↪semigroups)
>>> FreeModule(ZZ,Integer(3)).category()
Category of finite dimensional modules with basis over
 (Dedekind domains and euclidean domains
  and noetherian rings and infinite enumerated sets
  and metric spaces)
>>> (QQ**Integer(0)).category()
Category of finite enumerated finite dimensional vector spaces with basis
 over (number fields and quotient fields and metric spaces)
```

are_linearly_dependent(*vecs*)

Return `True` if the vectors `vecs` are linearly dependent and `False` otherwise.

EXAMPLES:

```
sage: M = QQ^3
sage: vecs = [M([1,2,3]), M([4,5,6])]
sage: M.are_linearly_dependent(vecs)
False
sage: vecs.append(M([3,3,3]))
sage: M.are_linearly_dependent(vecs)
True

sage: R.<x> = QQ[]
sage: M = FreeModule(R, 2)
sage: vecs = [M([x^2+1, x+1]), M([x+2, 2*x+1])]
sage: M.are_linearly_dependent(vecs)
False
sage: vecs.append(M([-2*x+1, -2*x^2+1]))
sage: M.are_linearly_dependent(vecs)
True
```

```
>>> from sage.all import *
>>> M = QQ**Integer(3)
>>> vecs = [M([Integer(1),Integer(2),Integer(3)]), M([Integer(4),Integer(5),
↪Integer(6)])]
>>> M.are_linearly_dependent(vecs)
False
>>> vecs.append(M([Integer(3),Integer(3),Integer(3)]))
```

(continues on next page)

(continued from previous page)

```
>>> M.are_linearly_dependent(vecs)
True

>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> M = FreeModule(R, Integer(2))
>>> vecs = [M([x**Integer(2)+Integer(1), x+Integer(1)]), M([x+Integer(2),␣
↪Integer(2)*x+Integer(1)])]
>>> M.are_linearly_dependent(vecs)
False
>>> vecs.append(M([-Integer(2)*x+Integer(1), -
↪Integer(2)*x**Integer(2)+Integer(1)]))
>>> M.are_linearly_dependent(vecs)
True
```

base_field()

Return the base field, which is the fraction field of the base ring of this module.

EXAMPLES:

```
sage: FreeModule(GF(3), 2).base_field()
Finite Field of size 3
sage: FreeModule(ZZ, 2).base_field()
Rational Field
sage: FreeModule(PolynomialRing(GF(7), 'x'), 2).base_field()
Fraction Field of Univariate Polynomial Ring in x
 over Finite Field of size 7
```

```
>>> from sage.all import *
>>> FreeModule(GF(Integer(3)), Integer(2)).base_field()
Finite Field of size 3
>>> FreeModule(ZZ, Integer(2)).base_field()
Rational Field
>>> FreeModule(PolynomialRing(GF(Integer(7)), 'x'), Integer(2)).base_field()
Fraction Field of Univariate Polynomial Ring in x
 over Finite Field of size 7
```

basis()

Return the basis of this module.

EXAMPLES:

```
sage: FreeModule(Integers(12),3).basis()
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
```

```
>>> from sage.all import *
>>> FreeModule(Integers(Integer(12)),Integer(3)).basis()
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
```

basis_matrix(*ring=None*)

Return the matrix whose rows are the basis for this free module.

INPUT:

- `ring` – (default: `self.coordinate_ring()`) a ring over which the matrix is defined

EXAMPLES:

```
sage: FreeModule(Integers(12), 3).basis_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
```

```
>>> from sage.all import *
>>> FreeModule(Integers(Integer(12)), Integer(3)).basis_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
```

```
sage: M = FreeModule(GF(7), 3).span([[2,3,4], [1,1,1]]); M
Vector space of degree 3 and dimension 2 over Finite Field of size 7
Basis matrix:
[1 0 6]
[0 1 2]
sage: M.basis_matrix()
[1 0 6]
[0 1 2]
```

```
>>> from sage.all import *
>>> M = FreeModule(GF(Integer(7)), Integer(3)).span([[Integer(2),Integer(3),
↪Integer(4)], [Integer(1),Integer(1),Integer(1)]]); M
Vector space of degree 3 and dimension 2 over Finite Field of size 7
Basis matrix:
[1 0 6]
[0 1 2]
>>> M.basis_matrix()
[1 0 6]
[0 1 2]
```

```
sage: M = FreeModule(GF(7), 3).span_of_basis([[2,3,4], [1,1,1]])
sage: M.basis_matrix()
[2 3 4]
[1 1 1]
```

```
>>> from sage.all import *
>>> M = FreeModule(GF(Integer(7)), Integer(3)).span_of_basis([[Integer(2),
↪Integer(3),Integer(4)], [Integer(1),Integer(1),Integer(1)]])
>>> M.basis_matrix()
[2 3 4]
[1 1 1]
```

```
sage: M = FreeModule(QQ, 2).span_of_basis([[1,-1], [1,0]]); M
Vector space of degree 2 and dimension 2 over Rational Field
User basis matrix:
[ 1 -1]
[ 1  0]
sage: M.basis_matrix()
[ 1 -1]
```

(continues on next page)

(continued from previous page)

```
[ 1  0]
```

```
>>> from sage.all import *
>>> M = FreeModule(QQ, Integer(2)).span_of_basis([[Integer(1),-Integer(1)],␣
↪[Integer(1),Integer(0)]]); M
Vector space of degree 2 and dimension 2 over Rational Field
User basis matrix:
[ 1 -1]
[ 1  0]
>>> M.basis_matrix()
[ 1 -1]
[ 1  0]
```

cardinality()

Return the cardinality of the free module.

OUTPUT: either an integer or `+Infinity`

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: k.<a> = FiniteField(9)
sage: V = VectorSpace(k, 3)
sage: V.cardinality()
729
sage: W = V.span([[1,2,1], [0,1,1]])
sage: W.cardinality()
81

sage: R = IntegerModRing(12)
sage: M = FreeModule(R, 2)
sage: M.cardinality()
144

sage: (QQ^3).cardinality()
+Infinity
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> k = FiniteField(Integer(9), names=('a',)); (a,) = k._first_ngens(1)
>>> V = VectorSpace(k, Integer(3))
>>> V.cardinality()
729
>>> W = V.span([[Integer(1),Integer(2),Integer(1)], [Integer(0),Integer(1),
↪Integer(1)]])
>>> W.cardinality()
81

>>> R = IntegerModRing(Integer(12))
>>> M = FreeModule(R, Integer(2))
>>> M.cardinality()
144
```

(continues on next page)

(continued from previous page)

```
>>> (QQ**Integer(3)).cardinality()
+Infinity
```

codimension()

Return the codimension of this free module, which is the dimension of the ambient space minus the dimension of this free module.

EXAMPLES:

```
sage: M = Matrix(3, 4, range(12))
sage: V = M.left_kernel(); V
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[ 1 -2  1]
sage: V.dimension()
1
sage: V.codimension()
2
```

```
>>> from sage.all import *
>>> M = Matrix(Integer(3), Integer(4), range(Integer(12)))
>>> V = M.left_kernel(); V
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[ 1 -2  1]
>>> V.dimension()
1
>>> V.codimension()
2
```

The codimension of an ambient space is always zero:

```
sage: (QQ^10).codimension()
0
```

```
>>> from sage.all import *
>>> (QQ**Integer(10)).codimension()
0
```

construction()

The construction functor and base ring for `self`.

EXAMPLES:

```
sage: R = PolynomialRing(QQ, 3, 'x')
sage: V = R^5
sage: V.construction()
(VectorFunctor, Multivariate Polynomial Ring in x0, x1, x2 over Rational␣
↪Field)
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ, Integer(3), 'x')
>>> V = R**Integer(5)
>>> V.construction()
(VectorFunctor, Multivariate Polynomial Ring in x0, x1, x2 over Rational␣
↪Field)
```

coordinate_module(*V*)

Suppose `V` is a submodule of `self` (or a module commensurable with `self`), and that `self` is a free module over R of rank n. Let ϕ be the map from `self` to R^n that sends the basis vectors of `self` in order to the standard basis of R^n. This function returns the image $\phi(V)$.

Warning

If there is no integer d such that dV is a submodule of `self`, then this function will give total nonsense.

EXAMPLES:

We illustrate this function with some $\mathbf{Z}$-submodules of $\mathbf{Q}^3$:

```
sage: V = (ZZ^3).span([[1/2,3,5], [0,1,-3]])
sage: W = (ZZ^3).span([[1/2,4,2]])
sage: V.coordinate_module(W)
Free module of degree 2 and rank 1 over Integer Ring
User basis matrix:
[1 4]
sage: V.0 + 4*V.1
(1/2, 4, 2)
```

```
>>> from sage.all import *
>>> V = (ZZ**Integer(3)).span([[Integer(1)/Integer(2),Integer(3),Integer(5)],␣
↪[Integer(0),Integer(1),-Integer(3)]])
>>> W = (ZZ**Integer(3)).span([[Integer(1)/Integer(2),Integer(4),Integer(2)]])
>>> V.coordinate_module(W)
Free module of degree 2 and rank 1 over Integer Ring
User basis matrix:
[1 4]
>>> V.gen(0) + Integer(4)*V.gen(1)
(1/2, 4, 2)
```

In this example, the coordinate module isn't even in $\mathbf{Z}^3$:

```
sage: W = (ZZ^3).span([[1/4,2,1]])
sage: V.coordinate_module(W)
Free module of degree 2 and rank 1 over Integer Ring
User basis matrix:
[1/2   2]
```

```
>>> from sage.all import *
>>> W = (ZZ**Integer(3)).span([[Integer(1)/Integer(4),Integer(2),Integer(1)]])
>>> V.coordinate_module(W)
Free module of degree 2 and rank 1 over Integer Ring
```

(continues on next page)

(continued from previous page)

```
User basis matrix:
[1/2   2]
```

The following more elaborate example illustrates using this function to write a submodule in terms of integral cuspidal modular symbols:

```
sage: # needs sage.modular
sage: M = ModularSymbols(54)
sage: S = M.cuspidal_subspace()
sage: K = S.integral_structure(); K
Free module of degree 19 and rank 8 over Integer Ring
Echelon basis matrix:
[ 0  1  0  0 -1  0  0  0  0  0  0  0  0  0  0  0  0  0]
...
sage: L = M[0].integral_structure(); L
Free module of degree 19 and rank 2 over Integer Ring
Echelon basis matrix:
[ 0  1  1  0 -2  1 -1  1 -1 -2  2  0  0  0  0  0  0  0]
[ 0  0  3  0 -3  2 -1  2 -1 -4  2 -1 -2  1  2  0  0 -1  1]
sage: K.coordinate_module(L)
Free module of degree 8 and rank 2 over Integer Ring
User basis matrix:
[ 1  1  1 -1  1 -1  0  0]
[ 0  3  2 -1  2 -1 -1 -2]
sage: K.coordinate_module(L).basis_matrix() * K.basis_matrix()
[ 0  1  1  0 -2  1 -1  1 -1 -2  2  0  0  0  0  0  0  0]
[ 0  0  3  0 -3  2 -1  2 -1 -4  2 -1 -2  1  2  0  0 -1  1]
```

```
>>> from sage.all import *
>>> # needs sage.modular
>>> M = ModularSymbols(Integer(54))
>>> S = M.cuspidal_subspace()
>>> K = S.integral_structure(); K
Free module of degree 19 and rank 8 over Integer Ring
Echelon basis matrix:
[ 0  1  0  0 -1  0  0  0  0  0  0  0  0  0  0  0  0  0]
...
>>> L = M[Integer(0)].integral_structure(); L
Free module of degree 19 and rank 2 over Integer Ring
Echelon basis matrix:
[ 0  1  1  0 -2  1 -1  1 -1 -2  2  0  0  0  0  0  0  0]
[ 0  0  3  0 -3  2 -1  2 -1 -4  2 -1 -2  1  2  0  0 -1  1]
>>> K.coordinate_module(L)
Free module of degree 8 and rank 2 over Integer Ring
User basis matrix:
[ 1  1  1 -1  1 -1  0  0]
[ 0  3  2 -1  2 -1 -1 -2]
>>> K.coordinate_module(L).basis_matrix() * K.basis_matrix()
[ 0  1  1  0 -2  1 -1  1 -1 -2  2  0  0  0  0  0  0  0]
[ 0  0  3  0 -3  2 -1  2 -1 -4  2 -1 -2  1  2  0  0 -1  1]
```

coordinate_ring()

Return the ring over which the entries of the vectors are defined.

This is the same as `base_ring()` unless an explicit basis was given over the fraction field.

EXAMPLES:

```
sage: M = ZZ^2
sage: M.coordinate_ring()
Integer Ring
```

```
>>> from sage.all import *
>>> M = ZZ**Integer(2)
>>> M.coordinate_ring()
Integer Ring
```

```
sage: M = (ZZ^2) * (1/2)
sage: M.base_ring()
Integer Ring
sage: M.coordinate_ring()
Rational Field
```

```
>>> from sage.all import *
>>> M = (ZZ**Integer(2)) * (Integer(1)/Integer(2))
>>> M.base_ring()
Integer Ring
>>> M.coordinate_ring()
Rational Field
```

```
sage: R.<x> = QQ[]
sage: L = R^2
sage: L.coordinate_ring()
Univariate Polynomial Ring in x over Rational Field
sage: L.span([(x,0), (1,x)]).coordinate_ring()
Univariate Polynomial Ring in x over Rational Field
sage: L.span([(x,0), (1,1/x)]).coordinate_ring()
Fraction Field of Univariate Polynomial Ring in x over Rational Field
sage: L.span([]).coordinate_ring()
Univariate Polynomial Ring in x over Rational Field
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> L = R**Integer(2)
>>> L.coordinate_ring()
Univariate Polynomial Ring in x over Rational Field
>>> L.span([(x,Integer(0)), (Integer(1),x)]).coordinate_ring()
Univariate Polynomial Ring in x over Rational Field
>>> L.span([(x,Integer(0)), (Integer(1),Integer(1)/x)]).coordinate_ring()
Fraction Field of Univariate Polynomial Ring in x over Rational Field
>>> L.span([]).coordinate_ring()
Univariate Polynomial Ring in x over Rational Field
```

coordinate_vector(*v*, *check=True*)

Return the vector whose coefficients give v as a linear combination of the basis for `self`.

INPUT:

- v – vector
- check – boolean (default: True); if True, also verify that v is really in self

OUTPUT: list

EXAMPLES:

```
sage: M = FreeModule(ZZ, 2); M0, M1 = M.gens()
sage: W = M.submodule([M0 + M1, M0 - 2*M1])
sage: W.coordinate_vector(2*M0 - M1)
(2, -1)
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ, Integer(2)); M0, M1 = M.gens()
>>> W = M.submodule([M0 + M1, M0 - Integer(2)*M1])
>>> W.coordinate_vector(Integer(2)*M0 - M1)
(2, -1)
```

coordinates(*v*, *check=True*)

Write v in terms of the basis for self.

INPUT:

- v – vector
- check – boolean (default: True); if True, also verify that v is really in self

OUTPUT: list

Returns a list c such that if B is the basis for self, then

$$\sum c_i B_i = v.$$

If v is not in self, raise an ArithmeticError exception.

EXAMPLES:

```
sage: M = FreeModule(ZZ, 2); M0, M1 = M.gens()
sage: W = M.submodule([M0 + M1, M0 - 2*M1])
sage: W.coordinates(2*M0 - M1)
[2, -1]
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ, Integer(2)); M0, M1 = M.gens()
>>> W = M.submodule([M0 + M1, M0 - Integer(2)*M1])
>>> W.coordinates(Integer(2)*M0 - M1)
[2, -1]
```

dense_module()

Return corresponding dense module.

EXAMPLES:

We first illustrate conversion with ambient spaces:

```
sage: M = FreeModule(QQ, 3)
sage: S = FreeModule(QQ, 3, sparse=True)
```

(continues on next page)

(continued from previous page)

```
sage: M.sparse_module()
Sparse vector space of dimension 3 over Rational Field
sage: S.dense_module()
Vector space of dimension 3 over Rational Field
sage: M.sparse_module() == S
True
sage: S.dense_module() == M
True
sage: M.dense_module() == M
True
sage: S.sparse_module() == S
True
```

```
>>> from sage.all import *
>>> M = FreeModule(QQ, Integer(3))
>>> S = FreeModule(QQ, Integer(3), sparse=True)
>>> M.sparse_module()
Sparse vector space of dimension 3 over Rational Field
>>> S.dense_module()
Vector space of dimension 3 over Rational Field
>>> M.sparse_module() == S
True
>>> S.dense_module() == M
True
>>> M.dense_module() == M
True
>>> S.sparse_module() == S
True
```

Next we create a subspace:

```
sage: M = FreeModule(QQ, 3, sparse=True)
sage: V = M.span([ [1,2,3] ] ); V
Sparse vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 2 3]
sage: V.sparse_module()
Sparse vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 2 3]
```

```
>>> from sage.all import *
>>> M = FreeModule(QQ, Integer(3), sparse=True)
>>> V = M.span([ [Integer(1),Integer(2),Integer(3)] ] ); V
Sparse vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 2 3]
>>> V.sparse_module()
Sparse vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 2 3]
```

dimension()

Return the dimension of this free module.

EXAMPLES:

```
sage: M = FreeModule(FiniteField(19), 100)
sage: W = M.submodule([M.gen(50)])
sage: W.dimension()
1
```

```
>>> from sage.all import *
>>> M = FreeModule(FiniteField(Integer(19)), Integer(100))
>>> W = M.submodule([M.gen(Integer(50))])
>>> W.dimension()
1
```

direct_sum(*other*)

Return the direct sum of `self` and `other` as a free module.

EXAMPLES:

```
sage: V = (ZZ^3).span([[1/2,3,5], [0,1,-3]]); V
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[1/2   0  14]
[  0   1  -3]
sage: W = (ZZ^3).span([[1/2,4,2]]); W
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[1/2   4   2]
sage: V.direct_sum(W)
Free module of degree 6 and rank 3 over Integer Ring
Echelon basis matrix:
[1/2   0  14   0   0   0]
[  0   1  -3   0   0   0]
[  0   0   0 1/2   4   2]
```

```
>>> from sage.all import *
>>> V = (ZZ**Integer(3)).span([[Integer(1)/Integer(2),Integer(3),Integer(5)],␣
↪[Integer(0),Integer(1),-Integer(3)]]); V
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[1/2   0  14]
[  0   1  -3]
>>> W = (ZZ**Integer(3)).span([[Integer(1)/Integer(2),Integer(4),
↪Integer(2)]]); W
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[1/2   4   2]
>>> V.direct_sum(W)
Free module of degree 6 and rank 3 over Integer Ring
Echelon basis matrix:
[1/2   0  14   0   0   0]
[  0   1  -3   0   0   0]
```

(continues on next page)

(continued from previous page)

```
[  0   0   0 1/2   4   2]
```

discriminant()

Return the discriminant of this free module.

EXAMPLES:

```
sage: M = FreeModule(ZZ, 3)
sage: M.discriminant()
1
sage: W = M.span([[1,2,3]])
sage: W.discriminant()
14
sage: W2 = M.span([[1,2,3], [1,1,1]])
sage: W2.discriminant()
6
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ, Integer(3))
>>> M.discriminant()
1
>>> W = M.span([[Integer(1),Integer(2),Integer(3)]])
>>> W.discriminant()
14
>>> W2 = M.span([[Integer(1),Integer(2),Integer(3)], [Integer(1),Integer(1),
↪Integer(1)]])
>>> W2.discriminant()
6
```

echelonized_basis_matrix()

The echelonized basis matrix (not implemented for this module).

This example works because M is an ambient module. Submodule creation should exist for generic modules.

EXAMPLES:

```
sage: R = IntegerModRing(12)
sage: S.<x,y> = R[]
sage: M = FreeModule(S, 3)
sage: M.echelonized_basis_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
```

```
>>> from sage.all import *
>>> R = IntegerModRing(Integer(12))
>>> S = R['x, y']; (x, y,) = S._first_ngens(2)
>>> M = FreeModule(S, Integer(3))
>>> M.echelonized_basis_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
```

free_module()

Return this free module. (This is used by the `FreeModule` functor, and simply returns self.)

EXAMPLES:

```
sage: M = FreeModule(ZZ, 3)
sage: M.free_module()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ, Integer(3))
>>> M.free_module()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

gen(*i=0*)

Return the i-th generator for `self`.

Here i is between 0 and rank - 1, inclusive.

INPUT:

- `i` – integer (default: 0)

OUTPUT: i-th basis vector for `self`

EXAMPLES:

```
sage: n = 5
sage: V = QQ^n
sage: B = [V.gen(i) for i in range(n)]
sage: B
[(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: V.gens() == tuple(B)
True
```

```
>>> from sage.all import *
>>> n = Integer(5)
>>> V = QQ**n
>>> B = [V.gen(i) for i in range(n)]
>>> B
[(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)]
>>> V.gens() == tuple(B)
True
```

gens()

Return a tuple of basis elements of `self`.

EXAMPLES:

```
sage: FreeModule(Integers(12),3).gens()
((1, 0, 0), (0, 1, 0), (0, 0, 1))
```

```
>>> from sage.all import *
>>> FreeModule(Integers(Integer(12)),Integer(3)).gens()
((1, 0, 0), (0, 1, 0), (0, 0, 1))
```

gram_matrix()

Return the gram matrix associated to this free module, defined to be $G = B * A * B.transpose()$, where A is the inner product matrix (induced from the ambient space), and B the basis matrix.

EXAMPLES:

```
sage: V = VectorSpace(QQ,4)
sage: u = V([1/2,1/2,1/2,1/2])
sage: v = V([0,1,1,0])
sage: w = V([0,0,1,1])
sage: M = span([u,v,w], ZZ)
sage: M.inner_product_matrix() == V.inner_product_matrix()
True
sage: L = M.submodule_with_basis([u,v,w])
sage: L.inner_product_matrix() == M.inner_product_matrix()
True
sage: L.gram_matrix()
[1 1 1]
[1 2 1]
[1 1 2]
```

```
>>> from sage.all import *
>>> V = VectorSpace(QQ,Integer(4))
>>> u = V([Integer(1)/Integer(2),Integer(1)/Integer(2),Integer(1)/Integer(2),
↪Integer(1)/Integer(2)])
>>> v = V([Integer(0),Integer(1),Integer(1),Integer(0)])
>>> w = V([Integer(0),Integer(0),Integer(1),Integer(1)])
>>> M = span([u,v,w], ZZ)
>>> M.inner_product_matrix() == V.inner_product_matrix()
True
>>> L = M.submodule_with_basis([u,v,w])
>>> L.inner_product_matrix() == M.inner_product_matrix()
True
>>> L.gram_matrix()
[1 1 1]
[1 2 1]
[1 1 2]
```

has_user_basis()

Return `True` if the basis of this free module is specified by the user, as opposed to being the default echelon form.

EXAMPLES:

```
sage: V = QQ^3
sage: W = V.subspace([[2,'1/2', 1]])
```

(continues on next page)

(continued from previous page)

```
sage: W.has_user_basis()
False
sage: W = V.subspace_with_basis([[2,'1/2',1]])
sage: W.has_user_basis()
True
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3)
>>> W = V.subspace([[Integer(2),'1/2', Integer(1)]])
>>> W.has_user_basis()
False
>>> W = V.subspace_with_basis([[Integer(2),'1/2',Integer(1)]])
>>> W.has_user_basis()
True
```

hom(*im_gens*, *codomain=None*, ***kwds*)

Override the hom method to handle the case of morphisms given by left-multiplication of a matrix and the codomain is not given.

EXAMPLES:

```
sage: W = ZZ^2; W.hom(matrix(1, [1, 2]), side='right')
Free module morphism defined as left-multiplication by the matrix
[1 2]
Domain: Ambient free module of rank 2 over the principal ideal domain Integer␣
↪Ring
Codomain: Ambient free module of rank 1 over the principal ideal domain␣
↪Integer Ring
sage: V = QQ^2; V.hom(identity_matrix(2), side='right')
Vector space morphism represented as left-multiplication by the matrix:
[1 0]
[0 1]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
```

```
>>> from sage.all import *
>>> W = ZZ**Integer(2); W.hom(matrix(Integer(1), [Integer(1), Integer(2)]),␣
↪side='right')
Free module morphism defined as left-multiplication by the matrix
[1 2]
Domain: Ambient free module of rank 2 over the principal ideal domain Integer␣
↪Ring
Codomain: Ambient free module of rank 1 over the principal ideal domain␣
↪Integer Ring
>>> V = QQ**Integer(2); V.hom(identity_matrix(Integer(2)), side='right')
Vector space morphism represented as left-multiplication by the matrix:
[1 0]
[0 1]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
```

inner_product_matrix()

Return the default identity inner product matrix associated to this module.

By definition this is the inner product matrix of the ambient space, hence may be of degree greater than the rank of the module.

TODO: Differentiate the image ring of the inner product from the base ring of the module and/or ambient space. E.g. On an integral module over ZZ the inner product pairing could naturally take values in ZZ, QQ, RR, or CC.

EXAMPLES:

```
sage: M = FreeModule(ZZ, 3)
sage: M.inner_product_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ, Integer(3))
>>> M.inner_product_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
```

is_ambient()

Return `False` since this is not an ambient free module.

EXAMPLES:

```
sage: M = FreeModule(ZZ, 3).span([[1,2,3]]); M
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[1 2 3]
sage: M.is_ambient()
False
sage: M = (ZZ^2).span([[1,0], [0,1]])
sage: M
Free module of degree 2 and rank 2 over Integer Ring
Echelon basis matrix:
[1 0]
[0 1]
sage: M.is_ambient()
False
sage: M == M.ambient_module()
True
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ, Integer(3)).span([[Integer(1),Integer(2),Integer(3)]]);
↪ M
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[1 2 3]
>>> M.is_ambient()
False
>>> M = (ZZ**Integer(2)).span([[Integer(1),Integer(0)], [Integer(0),
↪Integer(1)]])
```

(continues on next page)

(continued from previous page)

```
>>> M
Free module of degree 2 and rank 2 over Integer Ring
Echelon basis matrix:
[1 0]
[0 1]
>>> M.is_ambient()
False
>>> M == M.ambient_module()
True
```

is_dense()

Return `True` if the underlying representation of this module uses dense vectors, and `False` otherwise.

EXAMPLES:

```
sage: FreeModule(ZZ, 2).is_dense()
True
sage: FreeModule(ZZ, 2, sparse=True).is_dense()
False
```

```
>>> from sage.all import *
>>> FreeModule(ZZ, Integer(2)).is_dense()
True
>>> FreeModule(ZZ, Integer(2), sparse=True).is_dense()
False
```

is_finite()

Return `True` if the underlying set of this free module is finite.

EXAMPLES:

```
sage: FreeModule(ZZ, 2).is_finite()
False
sage: FreeModule(Integers(8), 2).is_finite()
True
sage: FreeModule(ZZ, 0).is_finite()
True
```

```
>>> from sage.all import *
>>> FreeModule(ZZ, Integer(2)).is_finite()
False
>>> FreeModule(Integers(Integer(8)), Integer(2)).is_finite()
True
>>> FreeModule(ZZ, Integer(0)).is_finite()
True
```

is_full()

Return `True` if the rank of this module equals its degree.

EXAMPLES:

```
sage: FreeModule(ZZ, 2).is_full()
True
```

(continues on next page)

(continued from previous page)

```
sage: M = FreeModule(ZZ, 2).span([[1,2]])
sage: M.is_full()
False
```

```
>>> from sage.all import *
>>> FreeModule(ZZ, Integer(2)).is_full()
True
>>> M = FreeModule(ZZ, Integer(2)).span([[Integer(1),Integer(2)]])
>>> M.is_full()
False
```

is_submodule(*other*)

Return `True` if `self` is a submodule of `other`.

EXAMPLES:

```
sage: M = FreeModule(ZZ, 3)
sage: V = M.ambient_vector_space()
sage: X = V.span([[1/2,1/2,0], [1/2,0,1/2]], ZZ)
sage: Y = V.span([[1,1,1]], ZZ)
sage: N = X + Y
sage: M.is_submodule(X)
False
sage: M.is_submodule(Y)
False
sage: Y.is_submodule(M)
True
sage: N.is_submodule(M)
False
sage: M.is_submodule(N)
True

sage: M = FreeModule(ZZ, 2)
sage: M.is_submodule(M)
True
sage: N = M.scale(2)
sage: N.is_submodule(M)
True
sage: M.is_submodule(N)
False
sage: N = M.scale(1/2)
sage: N.is_submodule(M)
False
sage: M.is_submodule(N)
True
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ, Integer(3))
>>> V = M.ambient_vector_space()
>>> X = V.span([[Integer(1)/Integer(2),Integer(1)/Integer(2),Integer(0)],␣
↪[Integer(1)/Integer(2),Integer(0),Integer(1)/Integer(2)]], ZZ)
>>> Y = V.span([[Integer(1),Integer(1),Integer(1)]], ZZ)
```

(continues on next page)

(continued from previous page)

```
>>> N = X + Y
>>> M.is_submodule(X)
False
>>> M.is_submodule(Y)
False
>>> Y.is_submodule(M)
True
>>> N.is_submodule(M)
False
>>> M.is_submodule(N)
True

>>> M = FreeModule(ZZ, Integer(2))
>>> M.is_submodule(M)
True
>>> N = M.scale(Integer(2))
>>> N.is_submodule(M)
True
>>> M.is_submodule(N)
False
>>> N = M.scale(Integer(1)/Integer(2))
>>> N.is_submodule(M)
False
>>> M.is_submodule(N)
True
```

Since *basis()* is not implemented in general, submodule testing does not work for all PID's. However, trivial cases are already used (and useful) for coercion, e.g.:

```
sage: QQ(1/2) * vector(ZZ['x']['y'], [1,2,3,4])
(1/2, 1, 3/2, 2)
sage: vector(ZZ['x']['y'], [1,2,3,4]) * QQ(1/2)
(1/2, 1, 3/2, 2)
```

```
>>> from sage.all import *
>>> QQ(Integer(1)/Integer(2)) * vector(ZZ['x']['y'], [Integer(1),Integer(2),
↪Integer(3),Integer(4)])
(1/2, 1, 3/2, 2)
>>> vector(ZZ['x']['y'], [Integer(1),Integer(2),Integer(3),Integer(4)]) *␣
↪QQ(Integer(1)/Integer(2))
(1/2, 1, 3/2, 2)
```

matrix()

Return the basis matrix of this module, which is the matrix whose rows are a basis for this module.

EXAMPLES:

```
sage: M = FreeModule(ZZ, 2)
sage: M.matrix()
[1 0]
[0 1]
sage: M.submodule([M.gen(0) + M.gen(1), M.gen(0) - 2*M.gen(1)]).matrix()
```

(continues on next page)

(continued from previous page)

```
[1 1]
[0 3]
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ, Integer(2))
>>> M.matrix()
[1 0]
[0 1]
>>> M.submodule([M.gen(Integer(0)) + M.gen(Integer(1)), M.gen(Integer(0)) -␣
↪Integer(2)*M.gen(Integer(1))]).matrix()
[1 1]
[0 3]
```

ngens()

Return the number of basis elements of this free module.

EXAMPLES:

```
sage: FreeModule(ZZ, 2).ngens()
2
sage: FreeModule(ZZ, 0).ngens()
0
sage: FreeModule(ZZ, 2).span([[1,1]]).ngens()
1
```

```
>>> from sage.all import *
>>> FreeModule(ZZ, Integer(2)).ngens()
2
>>> FreeModule(ZZ, Integer(0)).ngens()
0
>>> FreeModule(ZZ, Integer(2)).span([[Integer(1),Integer(1)]]).ngens()
1
```

nonembedded_free_module()

Return an ambient free module that is isomorphic to this free module.

Thus if this free module is of rank n over a ring R, then this function returns R^n, as an ambient free module.

EXAMPLES:

```
sage: FreeModule(ZZ, 2).span([[1,1]]).nonembedded_free_module()
Ambient free module of rank 1 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> FreeModule(ZZ, Integer(2)).span([[Integer(1),Integer(1)]]).nonembedded_
↪free_module()
Ambient free module of rank 1 over the principal ideal domain Integer Ring
```

pseudoHom(*twist*, *codomain=None*)

Return the pseudo-Hom space corresponding to given data.

INPUT:

- `twist` – the twisting morphism or the twisting derivation

- `codomain` – (default: `None`) the codomain of the pseudo morphisms; if `None`, the codomain is the same as the domain

EXAMPLES:

```
sage: F = GF(25)
sage: Frob = F.frobenius_endomorphism()
sage: M = F^2
sage: M.pseudoHom(Frob)
Set of Pseudoendomorphisms (twisted by z2 |--> z2^5) of Vector space of␣
↪dimension 2 over Finite Field in z2 of size 5^2
```

```
>>> from sage.all import *
>>> F = GF(Integer(25))
>>> Frob = F.frobenius_endomorphism()
>>> M = F**Integer(2)
>>> M.pseudoHom(Frob)
Set of Pseudoendomorphisms (twisted by z2 |--> z2^5) of Vector space of␣
↪dimension 2 over Finite Field in z2 of size 5^2
```

See also

pseudohom()

pseudohom(*f*, *twist*, *codomain=None*, *side='left'*)

Return the pseudohomomorphism corresponding to the given data.

We recall that, given two R-modules M and M' together with a ring homomorphism $\theta : R \to R$ and a θ-derivation, $\delta : R \to R$ (that is, a map such that $\delta(xy) = \theta(x)\delta(y) + \delta(x)y$), a pseudomorphism $f : M \to M'$ is an additive map such that

$$f(\lambda x) = \theta(\lambda)f(x) + \delta(\lambda)x$$

When δ is nonzero, this requires that M coerces into M'.

Note

Internally, pseudomorphisms are represented by matrices with coefficient in the base ring R. See class *sage.modules.free_module_pseudomorphism.FreeModulePseudoMorphism* for details.

INPUT:

- `f` – a matrix (or any other data) defining the pseudomorphism
- `twist` – the twisting morphism or the twisting derivation (if a derivation is given, the corresponding morphism θ is automatically infered; see also `sage.rings.polynomial.ore_polynomial_ring.OrePolynomialRing`)
- `codomain` – (default: `None`) the codomain of the pseudo morphisms; if `None`, the codomain is the same than the domain
- `side` – (default: `left`) side of the vectors acted on by the matrix

EXAMPLES:

```
sage: K.<z> = GF(5^5)
sage: Frob = K.frobenius_endomorphism()
sage: V = K^2; W = K^3
sage: f = V.pseudohom([[1, z, z^2], [1, z^2, z^4]], Frob, codomain=W)
sage: f
Free module pseudomorphism (twisted by z |--> z^5) defined by the matrix
[  1   z z^2]
[  1 z^2 z^4]
Domain: Vector space of dimension 2 over Finite Field in z of size 5^5
Codomain: Vector space of dimension 3 over Finite Field in z of size 5^5
```

```
>>> from sage.all import *
>>> K = GF(Integer(5)**Integer(5), names=('z',)); (z,) = K._first_ngens(1)
>>> Frob = K.frobenius_endomorphism()
>>> V = K**Integer(2); W = K**Integer(3)
>>> f = V.pseudohom([[Integer(1), z, z**Integer(2)], [Integer(1),␣
↪z**Integer(2), z**Integer(4)]], Frob, codomain=W)
>>> f
Free module pseudomorphism (twisted by z |--> z^5) defined by the matrix
[  1   z z^2]
[  1 z^2 z^4]
Domain: Vector space of dimension 2 over Finite Field in z of size 5^5
Codomain: Vector space of dimension 3 over Finite Field in z of size 5^5
```

We check that f is indeed semi-linear:

```
sage: v = V([z+1, z+2])
sage: f(z*v)
(2*z^2 + z + 4, z^4 + 2*z^3 + 3*z^2 + z, 4*z^4 + 2*z^2 + 3*z + 2)
sage: z^5 * f(v)
(2*z^2 + z + 4, z^4 + 2*z^3 + 3*z^2 + z, 4*z^4 + 2*z^2 + 3*z + 2)
```

```
>>> from sage.all import *
>>> v = V([z+Integer(1), z+Integer(2)])
>>> f(z*v)
(2*z^2 + z + 4, z^4 + 2*z^3 + 3*z^2 + z, 4*z^4 + 2*z^2 + 3*z + 2)
>>> z**Integer(5) * f(v)
(2*z^2 + z + 4, z^4 + 2*z^3 + 3*z^2 + z, 4*z^4 + 2*z^2 + 3*z + 2)
```

An example with a derivation:

```
sage: R.<t> = ZZ[]
sage: d = R.derivation()
sage: M = R^2
sage: Nabla = M.pseudohom([[1, t], [t^2, t^3]], d, side='right')
sage: Nabla
Free module pseudomorphism (twisted by d/dt) defined as left-multiplication␣
↪by the matrix
[  1   t]
[t^2 t^3]
Domain: Ambient free module of rank 2 over the integral domain Univariate␣
↪Polynomial Ring in t over Integer Ring
Codomain: Ambient free module of rank 2 over the integral domain Univariate␣
```

(continues on next page)

(continued from previous page)

```
↪Polynomial Ring in t over Integer Ring

sage: v = M([1,1])
sage: Nabla(v)
(t + 1, t^3 + t^2)
sage: Nabla(t*v)
(t^2 + t + 1, t^4 + t^3 + 1)
sage: Nabla(t*v) == t * Nabla(v) + v
True
```

```
>>> from sage.all import *
>>> R = ZZ['t']; (t,) = R._first_ngens(1)
>>> d = R.derivation()
>>> M = R**Integer(2)
>>> Nabla = M.pseudohom([[Integer(1), t], [t**Integer(2), t**Integer(3)]], d,␣
↪side='right')
>>> Nabla
Free module pseudomorphism (twisted by d/dt) defined as left-multiplication␣
↪by the matrix
[  1   t]
[t^2 t^3]
Domain: Ambient free module of rank 2 over the integral domain Univariate␣
↪Polynomial Ring in t over Integer Ring
Codomain: Ambient free module of rank 2 over the integral domain Univariate␣
↪Polynomial Ring in t over Integer Ring

>>> v = M([Integer(1),Integer(1)])
>>> Nabla(v)
(t + 1, t^3 + t^2)
>>> Nabla(t*v)
(t^2 + t + 1, t^4 + t^3 + 1)
>>> Nabla(t*v) == t * Nabla(v) + v
True
```

If the twisting derivation is not zero, the domain must coerce into the codomain:

```
sage: N = R^3
sage: M.pseudohom([[1, t, t^2], [1, t^2, t^4]], d, codomain=N)
Traceback (most recent call last):
...
ValueError: the domain does not coerce into the codomain
```

```
>>> from sage.all import *
>>> N = R**Integer(3)
>>> M.pseudohom([[Integer(1), t, t**Integer(2)], [Integer(1), t**Integer(2),␣
↪t**Integer(4)]], d, codomain=N)
Traceback (most recent call last):
...
ValueError: the domain does not coerce into the codomain
```

See also

pseudoHom()

random_element (*prob=1.0*, **args*, ***kwds*)

Return a random element of `self`.

INPUT:

- `prob` – float. Each coefficient will be set to zero with probability $1 - prob$. Otherwise coefficients will be chosen randomly from base ring (and may be zero).
- `*args`, `**kwds` – passed on to the *random_element()* function of the base ring

EXAMPLES:

```
sage: M = FreeModule(ZZ, 2).span([[1, 1]])
sage: v = M.random_element()
sage: v.parent() is M
True
sage: v in M
True
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ, Integer(2)).span([[Integer(1), Integer(1)]])
>>> v = M.random_element()
>>> v.parent() is M
True
>>> v in M
True
```

Small entries are likely:

```
sage: for i in [-2, -1, 0, 1, 2]:
....:     while vector([i, i]) != M.random_element():
....:         pass
```

```
>>> from sage.all import *
>>> for i in [-Integer(2), -Integer(1), Integer(0), Integer(1), Integer(2)]:
...     while vector([i, i]) != M.random_element():
...         pass
```

Large entries appear as well:

```
sage: while abs(M.random_element()[0]) < 100:
....:     pass
```

```
>>> from sage.all import *
>>> while abs(M.random_element()[Integer(0)]) < Integer(100):
...     pass
```

Passes extra positional or keyword arguments through:

```
sage: all(i in range(5, 10) for i in M.random_element(1.0, 5, 10))
True
```

```
>>> from sage.all import *
>>> all(i in range(Integer(5), Integer(10)) for i in M.random_
↪element(RealNumber('1.0'), Integer(5), Integer(10)))
True
```

rank()

Return the rank of this free module.

EXAMPLES:

```
sage: FreeModule(Integers(6), 10000000).rank()
10000000
sage: FreeModule(ZZ, 2).span([[1,1], [2,2], [3,4]]).rank()
2
```

```
>>> from sage.all import *
>>> FreeModule(Integers(Integer(6)), Integer(10000000)).rank()
10000000
>>> FreeModule(ZZ, Integer(2)).span([[Integer(1),Integer(1)], [Integer(2),
↪Integer(2)], [Integer(3),Integer(4)]]).rank()
2
```

relations()

Return the module of relations of `self`.

EXAMPLES:

```
sage: V = GF(2)^2
sage: V.relations() == V.zero_submodule()
True
sage: W = V.subspace([[1, 0]])
sage: W.relations() == V.zero_submodule()
True

sage: Q = V / W
sage: Q.relations() == W
True
```

```
>>> from sage.all import *
>>> V = GF(Integer(2))**Integer(2)
>>> V.relations() == V.zero_submodule()
True
>>> W = V.subspace([[Integer(1), Integer(0)]])
>>> W.relations() == V.zero_submodule()
True

>>> Q = V / W
>>> Q.relations() == W
True
```

scale(*other*)

Return the product of this module by the number other, which is the module spanned by other times each basis vector.

EXAMPLES:

```
sage: M = FreeModule(ZZ, 3)
sage: M.scale(2)
Free module of degree 3 and rank 3 over Integer Ring
Echelon basis matrix:
[2 0 0]
[0 2 0]
[0 0 2]
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ, Integer(3))
>>> M.scale(Integer(2))
Free module of degree 3 and rank 3 over Integer Ring
Echelon basis matrix:
[2 0 0]
[0 2 0]
[0 0 2]
```

```
sage: a = QQ('1/3')
sage: M.scale(a)
Free module of degree 3 and rank 3 over Integer Ring
Echelon basis matrix:
[1/3   0   0]
[  0 1/3   0]
[  0   0 1/3]
```

```
>>> from sage.all import *
>>> a = QQ('1/3')
>>> M.scale(a)
Free module of degree 3 and rank 3 over Integer Ring
Echelon basis matrix:
[1/3   0   0]
[  0 1/3   0]
[  0   0 1/3]
```

sparse_module()

Return the corresponding sparse module with the same defining data.

EXAMPLES:

We first illustrate conversion with ambient spaces:

```
sage: M = FreeModule(Integers(8), 3)
sage: S = FreeModule(Integers(8), 3, sparse=True)
sage: M.sparse_module()
Ambient sparse free module of rank 3 over Ring of integers modulo 8
sage: S.dense_module()
Ambient free module of rank 3 over Ring of integers modulo 8
sage: M.sparse_module() is S
True
sage: S.dense_module() is M
True
sage: M.dense_module() is M
```

(continues on next page)

(continued from previous page)

```
True
sage: S.sparse_module() is S
True
```

```
>>> from sage.all import *
>>> M = FreeModule(Integers(Integer(8)), Integer(3))
>>> S = FreeModule(Integers(Integer(8)), Integer(3), sparse=True)
>>> M.sparse_module()
Ambient sparse free module of rank 3 over Ring of integers modulo 8
>>> S.dense_module()
Ambient free module of rank 3 over Ring of integers modulo 8
>>> M.sparse_module() is S
True
>>> S.dense_module() is M
True
>>> M.dense_module() is M
True
>>> S.sparse_module() is S
True
```

Next we convert a subspace:

```
sage: M = FreeModule(QQ,3)
sage: V = M.span([ [1,2,3] ] ); V
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 2 3]
sage: V.sparse_module()
Sparse vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 2 3]
```

```
>>> from sage.all import *
>>> M = FreeModule(QQ,Integer(3))
>>> V = M.span([ [Integer(1),Integer(2),Integer(3)] ] ); V
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 2 3]
>>> V.sparse_module()
Sparse vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 2 3]
```

uses_ambient_inner_product()

Return `True` if the inner product on this module is the one induced by the ambient inner product.

EXAMPLES:

```
sage: M = FreeModule(ZZ, 2)
sage: W = M.submodule([[1,2]])
sage: W.uses_ambient_inner_product()
True
```

(continues on next page)

(continued from previous page)

```
sage: W.inner_product_matrix()
[1 0]
[0 1]
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ, Integer(2))
>>> W = M.submodule([[Integer(1),Integer(2)]])
>>> W.uses_ambient_inner_product()
True
>>> W.inner_product_matrix()
[1 0]
[0 1]
```

```
sage: W.gram_matrix()
[5]
```

```
>>> from sage.all import *
>>> W.gram_matrix()
[5]
```

class sage.modules.free_module.**FreeModule_generic_domain**(*base_ring*, *rank*, *degree*, *sparse=False*, *coordinate_ring=None*, *category=None*)

Bases: *FreeModule_generic*

Base class for free modules over an integral domain.

class sage.modules.free_module.**FreeModule_generic_field**(*base_field*, *dimension*, *degree*, *sparse=False*, *category=None*)

Bases: *FreeModule_generic_pid*

Base class for all free modules over fields.

complement()

Return the complement of `self` in the *ambient_vector_space()*.

EXAMPLES:

```
sage: V = QQ^3
sage: V.complement()
Vector space of degree 3 and dimension 0 over Rational Field
Basis matrix:
[]
sage: V == V.complement().complement()
True
sage: W = V.span([[1, 0, 1]])
sage: X = W.complement(); X
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  0]
sage: X.complement() == W
True
```

(continues on next page)

(continued from previous page)

```
sage: X + W == V
True
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3)
>>> V.complement()
Vector space of degree 3 and dimension 0 over Rational Field
Basis matrix:
[]
>>> V == V.complement().complement()
True
>>> W = V.span([[Integer(1), Integer(0), Integer(1)]])
>>> X = W.complement(); X
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  0]
>>> X.complement() == W
True
>>> X + W == V
True
```

Even though we construct a subspace of a subspace, the orthogonal complement is still done in the ambient vector space $\mathbf{Q}^3$:

```
sage: V = QQ^3
sage: W = V.subspace_with_basis([[1,0,1],[-1,1,0]])
sage: X = W.subspace_with_basis([[1,0,1]])
sage: X.complement()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  0]
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3)
>>> W = V.subspace_with_basis([[Integer(1),Integer(0),Integer(1)],[-
↪Integer(1),Integer(1),Integer(0)]])
>>> X = W.subspace_with_basis([[Integer(1),Integer(0),Integer(1)]])
>>> X.complement()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  0]
```

All these complements are only done with respect to the inner product in the usual basis. Over finite fields, this means we can get complements which are only isomorphic to a vector space decomposition complement.

```
sage: F2 = GF(2, 'x')
sage: V = F2^6
sage: W = V.span([[1,1,0,0,0,0]]); W
Vector space of degree 6 and dimension 1 over Finite Field of size 2
```

(continues on next page)

(continued from previous page)

```
Basis matrix:
[1 1 0 0 0 0]
sage: W.complement()
Vector space of degree 6 and dimension 5 over Finite Field of size 2
Basis matrix:
[1 1 0 0 0 0]
[0 0 1 0 0 0]
[0 0 0 1 0 0]
[0 0 0 0 1 0]
[0 0 0 0 0 1]
sage: W.intersection(W.complement())
Vector space of degree 6 and dimension 1 over Finite Field of size 2
Basis matrix:
[1 1 0 0 0 0]
```

```
>>> from sage.all import *
>>> F2 = GF(Integer(2), 'x')
>>> V = F2**Integer(6)
>>> W = V.span([[Integer(1),Integer(1),Integer(0),Integer(0),Integer(0),
↪Integer(0)]]); W
Vector space of degree 6 and dimension 1 over Finite Field of size 2
Basis matrix:
[1 1 0 0 0 0]
>>> W.complement()
Vector space of degree 6 and dimension 5 over Finite Field of size 2
Basis matrix:
[1 1 0 0 0 0]
[0 0 1 0 0 0]
[0 0 0 1 0 0]
[0 0 0 0 1 0]
[0 0 0 0 0 1]
>>> W.intersection(W.complement())
Vector space of degree 6 and dimension 1 over Finite Field of size 2
Basis matrix:
[1 1 0 0 0 0]
```

echelonized_basis_matrix()

Return basis matrix for `self` in row echelon form.

EXAMPLES:

```
sage: V = FreeModule(QQ, 3).span_of_basis([[1,2,3],[4,5,6]])
sage: V.basis_matrix()
[1 2 3]
[4 5 6]
sage: V.echelonized_basis_matrix()
[ 1  0 -1]
[ 0  1  2]
```

```
>>> from sage.all import *
>>> V = FreeModule(QQ, Integer(3)).span_of_basis([[Integer(1),Integer(2),
↪Integer(3)],[Integer(4),Integer(5),Integer(6)]])
```

(continues on next page)

(continued from previous page)

```
>>> V.basis_matrix()
[1 2 3]
[4 5 6]
>>> V.echelonized_basis_matrix()
[ 1  0 -1]
[ 0  1  2]
```

intersection(*other*)

Return the intersection of `self` and `other`, which must be R-submodules of a common ambient vector space.

EXAMPLES:

```
sage: V  = VectorSpace(QQ,3)
sage: W1 = V.submodule([V.gen(0), V.gen(0) + V.gen(1)])
sage: W2 = V.submodule([V.gen(1), V.gen(2)])
sage: W1.intersection(W2)
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[0 1 0]
sage: W2.intersection(W1)
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[0 1 0]
sage: V.intersection(W1)
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[1 0 0]
[0 1 0]
sage: W1.intersection(V)
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[1 0 0]
[0 1 0]
sage: Z = V.submodule([])
sage: W1.intersection(Z)
Vector space of degree 3 and dimension 0 over Rational Field
Basis matrix:
[]
```

```
>>> from sage.all import *
>>> V  = VectorSpace(QQ,Integer(3))
>>> W1 = V.submodule([V.gen(Integer(0)), V.gen(Integer(0)) + V.
↪gen(Integer(1))])
>>> W2 = V.submodule([V.gen(Integer(1)), V.gen(Integer(2))])
>>> W1.intersection(W2)
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[0 1 0]
>>> W2.intersection(W1)
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
```

(continues on next page)

(continued from previous page)

```
[0 1 0]
>>> V.intersection(W1)
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[1 0 0]
[0 1 0]
>>> W1.intersection(V)
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[1 0 0]
[0 1 0]
>>> Z = V.submodule([])
>>> W1.intersection(Z)
Vector space of degree 3 and dimension 0 over Rational Field
Basis matrix:
[]
```

is_subspace(*other*)

`True` if this vector space is a subspace of `other`.

EXAMPLES:

```
sage: V = VectorSpace(QQ,3)
sage: W = V.subspace([V.gen(0), V.gen(0) + V.gen(1)])
sage: W2 = V.subspace([V.gen(1)])
sage: W.is_subspace(V)
True
sage: W2.is_subspace(V)
True
sage: W.is_subspace(W2)
False
sage: W2.is_subspace(W)
True
```

```
>>> from sage.all import *
>>> V = VectorSpace(QQ,Integer(3))
>>> W = V.subspace([V.gen(Integer(0)), V.gen(Integer(0)) + V.gen(Integer(1))])
>>> W2 = V.subspace([V.gen(Integer(1))])
>>> W.is_subspace(V)
True
>>> W2.is_subspace(V)
True
>>> W.is_subspace(W2)
False
>>> W2.is_subspace(W)
True
```

linear_dependence(*vectors*, *zeros='left'*, *check=True*)

Return a list of vectors giving relations of linear dependence for the input list of vectors.

Can be used to check linear independence of a set of vectors.

INPUT:

- `vectors` – list of vectors; all from the same vector space
- `zeros` – (default: `'left'`) `'left'` or `'right'` as a general preference for where zeros are located in the returned coefficients
- `check` – (default: `True`) if `True` each item in the list `vectors` is checked for membership in `self`. Set to `False` if you can be certain the vectors come from the vector space.

OUTPUT:

Returns a list of vectors. The scalar entries of each vector provide the coefficients for a linear combination of the input vectors that will equal the zero vector in `self`. Furthermore, the returned list is linearly independent in the vector space over the same base field with degree equal to the length of the list `vectors`.

The linear independence of `vectors` is equivalent to the returned list being empty, so this provides a test - see the examples below.

The returned vectors are always independent, and with `zeros` set to `'left'` they have 1s in their first nonzero entries and a qualitative disposition to having zeros in the low-index entries. With `zeros` set to `'right'` the situation is reversed with a qualitative disposition for zeros in the high-index entries.

If the vectors in `vectors` are made the rows of a matrix V and the returned vectors are made the rows of a matrix R, then the matrix product RV is a zero matrix of the proper size. And R is a matrix of full rank. This routine uses kernels of matrices to compute these relations of linear dependence, but handles all the conversions between sets of vectors and matrices. If speed is important, consider working with the appropriate matrices and kernels instead.

EXAMPLES:

We begin with two linearly independent vectors, and add three non-trivial linear combinations to the set. We illustrate both types of output and check a selected relation of linear dependence.

```
sage: v1 = vector(QQ, [2, 1, -4, 3])
sage: v2 = vector(QQ, [1, 5, 2, -2])
sage: V = QQ^4
sage: V.linear_dependence([v1,v2])
[]

sage: v3 = v1 + v2
sage: v4 = 3*v1 - 4*v2
sage: v5 = -v1 + 2*v2
sage: L = [v1, v2, v3, v4, v5]

sage: relations = V.linear_dependence(L, zeros='left')
sage: relations
[(1, 0, 0, -1, -2), (0, 1, 0, -1/2, -3/2), (0, 0, 1, -3/2, -7/2)]
sage: v2 + (-1/2)*v4 + (-3/2)*v5
(0, 0, 0, 0)

sage: relations = V.linear_dependence(L, zeros='right')
sage: relations
[(-1, -1, 1, 0, 0), (-3, 4, 0, 1, 0), (1, -2, 0, 0, 1)]
sage: z = sum([relations[2][i]*L[i] for i in range(len(L))])
sage: z == zero_vector(QQ, 4)
True
```

```
>>> from sage.all import *
>>> v1 = vector(QQ, [Integer(2), Integer(1), -Integer(4), Integer(3)])
```

(continues on next page)

(continued from previous page)

```
>>> v2 = vector(QQ, [Integer(1), Integer(5), Integer(2), -Integer(2)])
>>> V = QQ**Integer(4)
>>> V.linear_dependence([v1,v2])
[]

>>> v3 = v1 + v2
>>> v4 = Integer(3)*v1 - Integer(4)*v2
>>> v5 = -v1 + Integer(2)*v2
>>> L = [v1, v2, v3, v4, v5]

>>> relations = V.linear_dependence(L, zeros='left')
>>> relations
[(1, 0, 0, -1, -2), (0, 1, 0, -1/2, -3/2), (0, 0, 1, -3/2, -7/2)]
>>> v2 + (-Integer(1)/Integer(2))*v4 + (-Integer(3)/Integer(2))*v5
(0, 0, 0, 0)

>>> relations = V.linear_dependence(L, zeros='right')
>>> relations
[(-1, -1, 1, 0, 0), (-3, 4, 0, 1, 0), (1, -2, 0, 0, 1)]
>>> z = sum([relations[Integer(2)][i]*L[i] for i in range(len(L))])
>>> z == zero_vector(QQ, Integer(4))
True
```

A linearly independent set returns an empty list, a result that can be tested.

```
sage: v1 = vector(QQ, [0,1,-3])
sage: v2 = vector(QQ, [4,1,0])
sage: V = QQ^3
sage: relations = V.linear_dependence([v1, v2]); relations
[]
sage: relations == []
True
```

```
>>> from sage.all import *
>>> v1 = vector(QQ, [Integer(0),Integer(1),-Integer(3)])
>>> v2 = vector(QQ, [Integer(4),Integer(1),Integer(0)])
>>> V = QQ**Integer(3)
>>> relations = V.linear_dependence([v1, v2]); relations
[]
>>> relations == []
True
```

Exact results result from exact fields. We start with three linearly independent vectors and add in two linear combinations to make a linearly dependent set of five vectors.

```
sage: F = FiniteField(17)
sage: v1 = vector(F, [1, 2, 3, 4, 5])
sage: v2 = vector(F, [2, 4, 8, 16, 15])
sage: v3 = vector(F, [1, 0, 0, 0, 1])
sage: (F^5).linear_dependence([v1, v2, v3]) == []
True
sage: L = [v1, v2, v3, 2*v1+v2, 3*v2+6*v3]
```

(continues on next page)

(continued from previous page)

```
sage: (F^5).linear_dependence(L)
[(1, 0, 16, 8, 3), (0, 1, 2, 0, 11)]
sage: v1 + 16*v3 + 8*(2*v1+v2) + 3*(3*v2+6*v3)
(0, 0, 0, 0, 0)
sage: v2 + 2*v3 + 11*(3*v2+6*v3)
(0, 0, 0, 0, 0)
sage: (F^5).linear_dependence(L, zeros='right')
[(15, 16, 0, 1, 0), (0, 14, 11, 0, 1)]
```

```
>>> from sage.all import *
>>> F = FiniteField(Integer(17))
>>> v1 = vector(F, [Integer(1), Integer(2), Integer(3), Integer(4),␣
↪Integer(5)])
>>> v2 = vector(F, [Integer(2), Integer(4), Integer(8), Integer(16),␣
↪Integer(15)])
>>> v3 = vector(F, [Integer(1), Integer(0), Integer(0), Integer(0),␣
↪Integer(1)])
>>> (F**Integer(5)).linear_dependence([v1, v2, v3]) == []
True
>>> L = [v1, v2, v3, Integer(2)*v1+v2, Integer(3)*v2+Integer(6)*v3]
>>> (F**Integer(5)).linear_dependence(L)
[(1, 0, 16, 8, 3), (0, 1, 2, 0, 11)]
>>> v1 + Integer(16)*v3 + Integer(8)*(Integer(2)*v1+v2) +␣
↪Integer(3)*(Integer(3)*v2+Integer(6)*v3)
(0, 0, 0, 0, 0)
>>> v2 + Integer(2)*v3 + Integer(11)*(Integer(3)*v2+Integer(6)*v3)
(0, 0, 0, 0, 0)
>>> (F**Integer(5)).linear_dependence(L, zeros='right')
[(15, 16, 0, 1, 0), (0, 14, 11, 0, 1)]
```

quotient_abstract(*sub*, *check=True*, ***kwds*)

Return an ambient free module isomorphic to the quotient space of `self` modulo `sub`, together with maps from `self` to the quotient, and a lifting map in the other direction.

Use `self.quotient(sub)` to obtain the quotient module as an object equipped with natural maps in both directions, and a canonical coercion.

INPUT:

- `sub` – a submodule of `self` or something that can be turned into one via `self.submodule(sub)`
- `check` – boolean (default: `True`); whether or not to check that sub is a submodule
- further named arguments, that are currently ignored.

OUTPUT:

- `U`; the quotient as an abstract *ambient* free module
- `pi`; projection map to the quotient
- `lift`; lifting map back from quotient

EXAMPLES:

```
sage: V = GF(19)^3
sage: W = V.span_of_basis([[1,2,3], [1,0,1]])
```

(continues on next page)

(continued from previous page)

```
sage: U, pi, lift = V.quotient_abstract(W)
sage: pi(V.2)
(18)
sage: pi(V.0)
(1)
sage: pi(V.0 + V.2)
(0)
```

```
>>> from sage.all import *
>>> V = GF(Integer(19))**Integer(3)
>>> W = V.span_of_basis([[Integer(1),Integer(2),Integer(3)], [Integer(1),
↪Integer(0),Integer(1)]])
>>> U, pi, lift = V.quotient_abstract(W)
>>> pi(V.gen(2))
(18)
>>> pi(V.gen(0))
(1)
>>> pi(V.gen(0) + V.gen(2))
(0)
```

Another example involving a quotient of one subspace by another:

```
sage: A = matrix(QQ, 4,4, [0,1,0,0, 0,0,1,0, 0,0,0,1, 0,0,0,0])
sage: V = (A^3).kernel()
sage: W = A.kernel()
sage: U, pi, lift = V.quotient_abstract(W)
sage: [pi(v) == 0 for v in W.gens()]
[True]
sage: [pi(lift(b)) == b for b in U.basis()]
[True, True]
```

```
>>> from sage.all import *
>>> A = matrix(QQ, Integer(4),Integer(4), [Integer(0),Integer(1),Integer(0),
↪Integer(0), Integer(0),Integer(0),Integer(1),Integer(0), Integer(0),
↪Integer(0),Integer(0),Integer(1), Integer(0),Integer(0),Integer(0),
↪Integer(0)])
>>> V = (A**Integer(3)).kernel()
>>> W = A.kernel()
>>> U, pi, lift = V.quotient_abstract(W)
>>> [pi(v) == Integer(0) for v in W.gens()]
[True]
>>> [pi(lift(b)) == b for b in U.basis()]
[True, True]
```

quotient_module(*sub*, *check=True*)

Return the quotient of `self` by the given subspace sub.

INPUT:

- `sub` – a submodule of `self`, or something that can be turned into one via `self.submodule(sub)`
- `check` – boolean (default: `True`); whether or not to check that `sub` is a submodule

EXAMPLES:

```
sage: A = QQ^3; V = A.span([[1,2,3], [4,5,6]])
sage: Q = V.quotient( [V.0 + V.1] ); Q
Vector space quotient V/W of dimension 1 over Rational Field where
V: Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
W: Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 1 1]
sage: Q(V.0 + V.1)
(0)
```

```
>>> from sage.all import *
>>> A = QQ**Integer(3); V = A.span([[Integer(1),Integer(2),Integer(3)],␣
↪[Integer(4),Integer(5),Integer(6)]])
>>> Q = V.quotient( [V.gen(0) + V.gen(1)] ); Q
Vector space quotient V/W of dimension 1 over Rational Field where
V: Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
W: Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 1 1]
>>> Q(V.gen(0) + V.gen(1))
(0)
```

We illustrate that the base rings must be the same:

```
sage: (QQ^2)/(ZZ^2)
Traceback (most recent call last):
...
ValueError: base rings must be the same
```

```
>>> from sage.all import *
>>> (QQ**Integer(2))/(ZZ**Integer(2))
Traceback (most recent call last):
...
ValueError: base rings must be the same
```

scale(*other*)

Return the product of `self` by the number other, which is the module spanned by `other` times each basis vector. Since `self` is a vector space this product equals `self` if `other` is nonzero, and is the zero vector space if `other` is 0.

EXAMPLES:

```
sage: V = QQ^4
sage: V.scale(5)
Vector space of dimension 4 over Rational Field
sage: V.scale(0)
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 *
>>> V = QQ**Integer(4)
>>> V.scale(Integer(5))
Vector space of dimension 4 over Rational Field
>>> V.scale(Integer(0))
Vector space of degree 4 and dimension 0 over Rational Field
Basis matrix:
[]
```

```
sage: W = V.span([[1,1,1,1]])
sage: W.scale(2)
Vector space of degree 4 and dimension 1 over Rational Field
Basis matrix:
[1 1 1 1]
sage: W.scale(0)
Vector space of degree 4 and dimension 0 over Rational Field
Basis matrix:
[]
```

```
>>> from sage.all import *
>>> W = V.span([[Integer(1),Integer(1),Integer(1),Integer(1)]])
>>> W.scale(Integer(2))
Vector space of degree 4 and dimension 1 over Rational Field
Basis matrix:
[1 1 1 1]
>>> W.scale(Integer(0))
Vector space of degree 4 and dimension 0 over Rational Field
Basis matrix:
[]
```

```
sage: V = QQ^4; V
Vector space of dimension 4 over Rational Field
sage: V.scale(3)
Vector space of dimension 4 over Rational Field
sage: V.scale(0)
Vector space of degree 4 and dimension 0 over Rational Field
Basis matrix:
[]
```

```
>>> from sage.all import *
>>> V = QQ**Integer(4); V
Vector space of dimension 4 over Rational Field
>>> V.scale(Integer(3))
Vector space of dimension 4 over Rational Field
>>> V.scale(Integer(0))
Vector space of degree 4 and dimension 0 over Rational Field
Basis matrix:
[]
```

span_of_basis(*basis*, *base_ring=None*, *check=True*, *already_echelonized=False*)

Return the free K-module with the given basis, where K is the base field of `self` or user specified base_ring.

Note that this span is a subspace of the ambient vector space, but need not be a subspace of `self`.

INPUT:

- `basis` – list of vectors
- `check` – boolean (default: `True`); whether or not to coerce entries of gens into base field
- `already_echelonized` – boolean (default: `False`); set this if you know the gens are already in echelon form

EXAMPLES:

```
sage: V = VectorSpace(GF(7), 3)
sage: W = V.subspace([[2,3,4]]); W
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 5 2]
sage: W.span_of_basis([[2,2,2], [3,3,0]])
Vector space of degree 3 and dimension 2 over Finite Field of size 7
User basis matrix:
[2 2 2]
[3 3 0]
```

```
>>> from sage.all import *
>>> V = VectorSpace(GF(Integer(7)), Integer(3))
>>> W = V.subspace([[Integer(2),Integer(3),Integer(4)]]); W
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 5 2]
>>> W.span_of_basis([[Integer(2),Integer(2),Integer(2)], [Integer(3),
↪Integer(3),Integer(0)]])
Vector space of degree 3 and dimension 2 over Finite Field of size 7
User basis matrix:
[2 2 2]
[3 3 0]
```

The basis vectors must be linearly independent or a `ValueError` exception is raised:

```
sage: W.span_of_basis([[2,2,2], [3,3,3]])
Traceback (most recent call last):
...
ValueError: The given basis vectors must be linearly independent.
```

```
>>> from sage.all import *
>>> W.span_of_basis([[Integer(2),Integer(2),Integer(2)], [Integer(3),
↪Integer(3),Integer(3)]])
Traceback (most recent call last):
...
ValueError: The given basis vectors must be linearly independent.
```

subspace(*gens*, *check=True*, *already_echelonized=False*)

Return the subspace of `self` spanned by the elements of gens.

INPUT:

- `gens` – list of vectors
- `check` – boolean (default: `True`); verify that gens are all in `self`
- `already_echelonized` – boolean (default: `False`); set to `True` if you know the gens are in Echelon form

EXAMPLES:

First we create a 1-dimensional vector subspace of an ambient 3-dimensional space over the finite field of order 7:

```
sage: V = VectorSpace(GF(7), 3)
sage: W = V.subspace([[2,3,4]]); W
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 5 2]
```

```
>>> from sage.all import *
>>> V = VectorSpace(GF(Integer(7)), Integer(3))
>>> W = V.subspace([[Integer(2),Integer(3),Integer(4)]]); W
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 5 2]
```

Next we create an invalid subspace, but it's allowed since `check=False`. This is just equivalent to computing the span of the element:

```
sage: W.subspace([[1,1,0]], check=False)
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 1 0]
```

```
>>> from sage.all import *
>>> W.subspace([[Integer(1),Integer(1),Integer(0)]], check=False)
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 1 0]
```

With `check=True` (the default) the mistake is correctly detected and reported with an `ArithmeticError` exception:

```
sage: W.subspace([[1,1,0]], check=True)
Traceback (most recent call last):
...
ArithmeticError: argument gens (= [[1, 1, 0]]) does not generate a submodule␣
↪of self
```

```
>>> from sage.all import *
>>> W.subspace([[Integer(1),Integer(1),Integer(0)]], check=True)
Traceback (most recent call last):
...
ArithmeticError: argument gens (= [[1, 1, 0]]) does not generate a submodule␣
↪of self
```

subspace_with_basis(*gens*, *check=True*, *already_echelonized=False*)

Same as `self.submodule_with_basis(...)`.

EXAMPLES:

We create a subspace with a user-defined basis.

```
sage: V = VectorSpace(GF(7), 3)
sage: W = V.subspace_with_basis([[2,2,2], [1,2,3]]); W
Vector space of degree 3 and dimension 2 over Finite Field of size 7
User basis matrix:
[2 2 2]
[1 2 3]
```

```
>>> from sage.all import *
>>> V = VectorSpace(GF(Integer(7)), Integer(3))
>>> W = V.subspace_with_basis([[Integer(2),Integer(2),Integer(2)],␣
↪[Integer(1),Integer(2),Integer(3)]]); W
Vector space of degree 3 and dimension 2 over Finite Field of size 7
User basis matrix:
[2 2 2]
[1 2 3]
```

We then create a subspace of the subspace with user-defined basis.

```
sage: W1 = W.subspace_with_basis([[3,4,5]]); W1
Vector space of degree 3 and dimension 1 over Finite Field of size 7
User basis matrix:
[3 4 5]
```

```
>>> from sage.all import *
>>> W1 = W.subspace_with_basis([[Integer(3),Integer(4),Integer(5)]]); W1
Vector space of degree 3 and dimension 1 over Finite Field of size 7
User basis matrix:
[3 4 5]
```

Notice how the basis for the same subspace is different if we merely use the `subspace` command.

```
sage: W2 = W.subspace([[3,4,5]]); W2
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 6 4]
```

```
>>> from sage.all import *
>>> W2 = W.subspace([[Integer(3),Integer(4),Integer(5)]]); W2
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 6 4]
```

Nonetheless the two subspaces are equal (as mathematical objects):

```
sage: W1 == W2
True
```

```
>>> from sage.all import *
>>> W1 == W2
True
```

subspaces(*dim*)

Iterate over all subspaces of dimension dim.

INPUT:

- `dim` – integer; dimension of subspaces to be generated

EXAMPLES:

```
sage: V = VectorSpace(GF(3), 5)
sage: len(list(V.subspaces(0)))
1
sage: len(list(V.subspaces(1)))
121
sage: len(list(V.subspaces(2)))
1210
sage: len(list(V.subspaces(3)))
1210
sage: len(list(V.subspaces(4)))
121
sage: len(list(V.subspaces(5)))
1
```

```
>>> from sage.all import *
>>> V = VectorSpace(GF(Integer(3)), Integer(5))
>>> len(list(V.subspaces(Integer(0))))
1
>>> len(list(V.subspaces(Integer(1))))
121
>>> len(list(V.subspaces(Integer(2))))
1210
>>> len(list(V.subspaces(Integer(3))))
1210
>>> len(list(V.subspaces(Integer(4))))
121
>>> len(list(V.subspaces(Integer(5))))
1
```

```
sage: V = VectorSpace(GF(3), 5)
sage: V = V.subspace([V([1,1,0,0,0]), V([0,0,1,1,0])])
sage: list(V.subspaces(1))
[Vector space of degree 5 and dimension 1 over Finite Field of size 3
Basis matrix:
[1 1 0 0 0],
 Vector space of degree 5 and dimension 1 over Finite Field of size 3
Basis matrix:
[1 1 1 1 0],
 Vector space of degree 5 and dimension 1 over Finite Field of size 3
Basis matrix:
[1 1 2 2 0],
```

(continues on next page)

(continued from previous page)

```
 Vector space of degree 5 and dimension 1 over Finite Field of size 3
Basis matrix:
[0 0 1 1 0]]
```

```
>>> from sage.all import *
>>> V = VectorSpace(GF(Integer(3)), Integer(5))
>>> V = V.subspace([V([Integer(1),Integer(1),Integer(0),Integer(0),
↪Integer(0)]), V([Integer(0),Integer(0),Integer(1),Integer(1),Integer(0)])])
>>> list(V.subspaces(Integer(1)))
[Vector space of degree 5 and dimension 1 over Finite Field of size 3
Basis matrix:
[1 1 0 0 0],
 Vector space of degree 5 and dimension 1 over Finite Field of size 3
Basis matrix:
[1 1 1 1 0],
 Vector space of degree 5 and dimension 1 over Finite Field of size 3
Basis matrix:
[1 1 2 2 0],
 Vector space of degree 5 and dimension 1 over Finite Field of size 3
Basis matrix:
[0 0 1 1 0]]
```

vector_space(*base_field=None*)

Return the vector space associated to `self`. Since `self` is a vector space this function simply returns `self`, unless the base field is different.

EXAMPLES:

```
sage: V = span([[1,2,3]],QQ); V
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 2 3]
sage: V.vector_space()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 2 3]
```

```
>>> from sage.all import *
>>> V = span([[Integer(1),Integer(2),Integer(3)]],QQ); V
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 2 3]
>>> V.vector_space()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 2 3]
```

zero_submodule()

Return the zero submodule of `self`.

EXAMPLES:

```
sage: (QQ^4).zero_submodule()
Vector space of degree 4 and dimension 0 over Rational Field
Basis matrix:
[]
```

```
>>> from sage.all import *
>>> (QQ**Integer(4)).zero_submodule()
Vector space of degree 4 and dimension 0 over Rational Field
Basis matrix:
[]
```

zero_subspace()

Return the zero subspace of `self`.

EXAMPLES:

```
sage: (QQ^4).zero_subspace()
Vector space of degree 4 and dimension 0 over Rational Field
Basis matrix:
[]
```

```
>>> from sage.all import *
>>> (QQ**Integer(4)).zero_subspace()
Vector space of degree 4 and dimension 0 over Rational Field
Basis matrix:
[]
```

class sage.modules.free_module.**FreeModule_generic_pid**(*base_ring*, *rank*, *degree*, *sparse=False*, *coordinate_ring=None*, *category=None*)

Bases: *FreeModule_generic_domain*

Base class for all free modules over a PID.

denominator()

The denominator of the basis matrix of `self` (i.e. the LCM of the coordinate entries with respect to the basis of the ambient space).

EXAMPLES:

```
sage: V = QQ^3
sage: L = V.span([[1,1/2,1/3], [-1/5,2/3,3]],ZZ)
sage: L
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1/5 19/6 37/3]
[   0 23/6 46/3]
sage: L.denominator()
30
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3)
>>> L = V.span([[Integer(1),Integer(1)/Integer(2),Integer(1)/Integer(3)], [-
↪Integer(1)/Integer(5),Integer(2)/Integer(3),Integer(3)]],ZZ)
```

(continues on next page)

(continued from previous page)

```
>>> L
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1/5 19/6 37/3]
[   0 23/6 46/3]
>>> L.denominator()
30
```

index_in(*other*)

Return the lattice index [other:self] of `self` in other, as an element of the base field. When `self` is contained in other, the lattice index is the usual index. If the index is infinite, then this function returns infinity.

EXAMPLES:

```
sage: L1 = span([[1,2]], ZZ)
sage: L2 = span([[3,6]], ZZ)
sage: L2.index_in(L1)
3
```

```
>>> from sage.all import *
>>> L1 = span([[Integer(1),Integer(2)]], ZZ)
>>> L2 = span([[Integer(3),Integer(6)]], ZZ)
>>> L2.index_in(L1)
3
```

Note that the free modules being compared need not be integral.

```
sage: L1 = span([['1/2','1/3'], [4,5]], ZZ)
sage: L2 = span([[1,2], [3,4]], ZZ)
sage: L2.index_in(L1)
12/7
sage: L1.index_in(L2)
7/12
sage: L1.discriminant() / L2.discriminant()
49/144
```

```
>>> from sage.all import *
>>> L1 = span([['1/2','1/3'], [Integer(4),Integer(5)]], ZZ)
>>> L2 = span([[Integer(1),Integer(2)], [Integer(3),Integer(4)]], ZZ)
>>> L2.index_in(L1)
12/7
>>> L1.index_in(L2)
7/12
>>> L1.discriminant() / L2.discriminant()
49/144
```

The index of a lattice of infinite index is infinite.

```
sage: L1 = FreeModule(ZZ, 2)
sage: L2 = span([[1,2]], ZZ)
sage: L2.index_in(L1)
+Infinity
```

```
>>> from sage.all import *
>>> L1 = FreeModule(ZZ, Integer(2))
>>> L2 = span([[Integer(1),Integer(2)]], ZZ)
>>> L2.index_in(L1)
+Infinity
```

index_in_saturation()

Return the index of this module in its saturation, i.e., its intersection with R^n.

EXAMPLES:

```
sage: W = span([[2,4,6]], ZZ)
sage: W.index_in_saturation()
2
sage: W = span([[1/2,1/3]], ZZ)
sage: W.index_in_saturation()
1/6
```

```
>>> from sage.all import *
>>> W = span([[Integer(2),Integer(4),Integer(6)]], ZZ)
>>> W.index_in_saturation()
2
>>> W = span([[Integer(1)/Integer(2),Integer(1)/Integer(3)]], ZZ)
>>> W.index_in_saturation()
1/6
```

intersection(*other*)

Return the intersection of `self` and `other`.

EXAMPLES:

We intersect two submodules one of which is clearly contained in the other:

```
sage: A = ZZ^2
sage: M1 = A.span([[1,1]])
sage: M2 = A.span([[3,3]])
sage: M1.intersection(M2)
Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[3 3]
sage: M1.intersection(M2) is M2
True
```

```
>>> from sage.all import *
>>> A = ZZ**Integer(2)
>>> M1 = A.span([[Integer(1),Integer(1)]])
>>> M2 = A.span([[Integer(3),Integer(3)]])
>>> M1.intersection(M2)
Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[3 3]
>>> M1.intersection(M2) is M2
True
```

We intersection two submodules of $\mathbf{Z}^3$ of rank 2, whose intersection has rank 1:

```
sage: A = ZZ^3
sage: M1 = A.span([[1,1,1], [1,2,3]])
sage: M2 = A.span([[2,2,2], [1,0,0]])
sage: M1.intersection(M2)
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[2 2 2]
```

```
>>> from sage.all import *
>>> A = ZZ**Integer(3)
>>> M1 = A.span([[Integer(1),Integer(1),Integer(1)], [Integer(1),Integer(2),
↪Integer(3)]])
>>> M2 = A.span([[Integer(2),Integer(2),Integer(2)], [Integer(1),Integer(0),
↪Integer(0)]])
>>> M1.intersection(M2)
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[2 2 2]
```

We compute an intersection of two $\mathbf{Z}$-modules that are not submodules of $\mathbf{Z}^2$:

```
sage: A = ZZ^2
sage: M1 = A.span([[1,2]]).scale(1/6)
sage: M2 = A.span([[1,2]]).scale(1/15)
sage: M1.intersection(M2)
Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[1/3 2/3]
```

```
>>> from sage.all import *
>>> A = ZZ**Integer(2)
>>> M1 = A.span([[Integer(1),Integer(2)]]).scale(Integer(1)/Integer(6))
>>> M2 = A.span([[Integer(1),Integer(2)]]).scale(Integer(1)/Integer(15))
>>> M1.intersection(M2)
Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[1/3 2/3]
```

We intersect a $\mathbf{Z}$-module with a $\mathbf{Q}$-vector space:

```
sage: A = ZZ^3
sage: L = ZZ^3
sage: V = QQ^3
sage: W = L.span([[1/2,0,1/2]])
sage: K = V.span([[1,0,1], [0,0,1]])
sage: W.intersection(K)
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[1/2   0 1/2]
sage: K.intersection(W)
Free module of degree 3 and rank 1 over Integer Ring
```

(continues on next page)

(continued from previous page)

```
Echelon basis matrix:
[1/2   0 1/2]
```

```
>>> from sage.all import *
>>> A = ZZ**Integer(3)
>>> L = ZZ**Integer(3)
>>> V = QQ**Integer(3)
>>> W = L.span([[Integer(1)/Integer(2),Integer(0),Integer(1)/Integer(2)]])
>>> K = V.span([[Integer(1),Integer(0),Integer(1)], [Integer(0),Integer(0),
↪Integer(1)]])
>>> W.intersection(K)
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[1/2   0 1/2]
>>> K.intersection(W)
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[1/2   0 1/2]
```

We intersect two modules over the ring of integers of a number field:

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: L.<w> = NumberField(x^2 - x + 2)
sage: OL = L.ring_of_integers()
sage: V = L**3
sage: W1 = V.span([[0,w/5,0], [1,0,-1/17]], OL)
sage: W2 = V.span([[0,(1-w)/5,0]], OL)
sage: W1.intersection(W2)
Free module of degree 3 and rank 1 over Maximal Order generated by w
 in Number Field in w with defining polynomial x^2 - x + 2
Echelon basis matrix:
[  0 2/5   0]
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> x = polygen(ZZ, 'x')
>>> L = NumberField(x**Integer(2) - x + Integer(2), names=('w',)); (w,) = L._
↪first_ngens(1)
>>> OL = L.ring_of_integers()
>>> V = L**Integer(3)
>>> W1 = V.span([[Integer(0),w/Integer(5),Integer(0)], [Integer(1),Integer(0),
↪-Integer(1)/Integer(17)]], OL)
>>> W2 = V.span([[Integer(0),(Integer(1)-w)/Integer(5),Integer(0)]], OL)
>>> W1.intersection(W2)
Free module of degree 3 and rank 1 over Maximal Order generated by w
 in Number Field in w with defining polynomial x^2 - x + 2
Echelon basis matrix:
[  0 2/5   0]
```

quotient_module(*sub*, *check=True*, ***kwds*)

Return the quotient of `self` by the given submodule sub.

INPUT:

- sub – a submodule of self, or something that can be turned into one via self.submodule(sub)
- check – boolean (default: True); whether or not to check that sub is a submodule
- further named arguments, that are passed to the constructor of the quotient space

EXAMPLES:

```
sage: A = ZZ^3; V = A.span([[1,2,3], [4,5,6]])
sage: Q = V.quotient( [V.0 + V.1] ); Q
Finitely generated module V/W over Integer Ring with invariants (0)
```

```
>>> from sage.all import *
>>> A = ZZ**Integer(3); V = A.span([[Integer(1),Integer(2),Integer(3)],␣
↪[Integer(4),Integer(5),Integer(6)]])
>>> Q = V.quotient( [V.gen(0) + V.gen(1)] ); Q
Finitely generated module V/W over Integer Ring with invariants (0)
```

saturation()

Return the saturated submodule of R^n that spans the same vector space as self.

EXAMPLES:

We create a 1-dimensional lattice that is obviously not saturated and saturate it.

```
sage: L = span([[9,9,6]], ZZ); L
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[9 9 6]
sage: L.saturation()
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[3 3 2]
```

```
>>> from sage.all import *
>>> L = span([[Integer(9),Integer(9),Integer(6)]], ZZ); L
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[9 9 6]
>>> L.saturation()
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[3 3 2]
```

We create a lattice spanned by two vectors, and saturate. Computation of discriminants shows that the index of lattice in its saturation is 3, which is a prime of congruence between the two generating vectors.

```
sage: L = span([[1,2,3], [4,5,6]], ZZ)
sage: L.saturation()
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1  0 -1]
[ 0  1  2]
sage: L.discriminant()
```

(continues on next page)

(continued from previous page)

```
54
sage: L.saturation().discriminant()
6
```

```
>>> from sage.all import *
>>> L = span([[Integer(1),Integer(2),Integer(3)], [Integer(4),Integer(5),
↪Integer(6)]], ZZ)
>>> L.saturation()
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1  0 -1]
[ 0  1  2]
>>> L.discriminant()
54
>>> L.saturation().discriminant()
6
```

Notice that the saturation of a non-integral lattice L is defined, but the result is integral hence does not contain L:

```
sage: L = span([['1/2',1,3]], ZZ)
sage: L.saturation()
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[1 2 6]
```

```
>>> from sage.all import *
>>> L = span([['1/2',Integer(1),Integer(3)]], ZZ)
>>> L.saturation()
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[1 2 6]
```

span_of_basis(*basis*, *base_ring=None*, *check=True*, *already_echelonized=False*)

Return the free R-module with the given basis, where R is the base ring of `self` or user specified base_ring.

Note that this R-module need not be a submodule of self, nor even of the ambient space. It must, however, be contained in the ambient vector space, i.e., the ambient space tensored with the fraction field of R.

EXAMPLES:

```
sage: M = FreeModule(ZZ,3)
sage: W = M.span_of_basis([M([1,2,3])])
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ,Integer(3))
>>> W = M.span_of_basis([M([Integer(1),Integer(2),Integer(3)])])
```

Next we create two free **Z**-modules, neither of which is a submodule of W.

```
sage: W.span_of_basis([M([2,4,0])])
Free module of degree 3 and rank 1 over Integer Ring
```

(continues on next page)

(continued from previous page)

```
User basis matrix:
[2 4 0]
```

```
>>> from sage.all import *
>>> W.span_of_basis([M([Integer(2),Integer(4),Integer(0)])])
Free module of degree 3 and rank 1 over Integer Ring
User basis matrix:
[2 4 0]
```

The following module isn't in the ambient module $\mathbf{Z}^3$ but is contained in the ambient vector space $\mathbf{Q}^3$:

```
sage: V = M.ambient_vector_space()
sage: W.span_of_basis([ V([1/5,2/5,0]), V([1/7,1/7,0]) ])
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1/5 2/5   0]
[1/7 1/7   0]
```

```
>>> from sage.all import *
>>> V = M.ambient_vector_space()
>>> W.span_of_basis([ V([Integer(1)/Integer(5),Integer(2)/Integer(5),
↪Integer(0)]), V([Integer(1)/Integer(7),Integer(1)/Integer(7),Integer(0)]) ])
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1/5 2/5   0]
[1/7 1/7   0]
```

Of course the input basis vectors must be linearly independent:

```
sage: W.span_of_basis([ [1,2,0], [2,4,0] ])
Traceback (most recent call last):
...
ValueError: The given basis vectors must be linearly independent.
```

```
>>> from sage.all import *
>>> W.span_of_basis([ [Integer(1),Integer(2),Integer(0)], [Integer(2),
↪Integer(4),Integer(0)] ])
Traceback (most recent call last):
...
ValueError: The given basis vectors must be linearly independent.
```

submodule_with_basis(*basis*, *check=True*, *already_echelonized=False*)

Create the R-submodule of the ambient vector space with given basis, where R is the base ring of `self`.

INPUT:

- `basis` – list of linearly independent vectors
- `check` – whether or not to verify that each gen is in the ambient vector space

OUTPUT: `FreeModule`; the R-submodule with given basis

EXAMPLES:

First we create a submodule of ZZ^3:

```
sage: M = FreeModule(ZZ, 3)
sage: B = M.basis()
sage: N = M.submodule_with_basis([B[0]+B[1], 2*B[1]-B[2]])
sage: N
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[ 1  1  0]
[ 0  2 -1]
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ, Integer(3))
>>> B = M.basis()
>>> N = M.submodule_with_basis([B[Integer(0)]+B[Integer(1)],␣
↪Integer(2)*B[Integer(1)]-B[Integer(2)]])
>>> N
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[ 1  1  0]
[ 0  2 -1]
```

A list of vectors in the ambient vector space may fail to generate a submodule.

```
sage: V = M.ambient_vector_space()
sage: X = M.submodule_with_basis([ V(B[0]+B[1])/2, V(B[1]-B[2])/2])
Traceback (most recent call last):
...
ArithmeticError: The given basis does not generate a submodule of self.
```

```
>>> from sage.all import *
>>> V = M.ambient_vector_space()
>>> X = M.submodule_with_basis([ V(B[Integer(0)]+B[Integer(1)])/Integer(2),␣
↪V(B[Integer(1)]-B[Integer(2)])/Integer(2)])
Traceback (most recent call last):
...
ArithmeticError: The given basis does not generate a submodule of self.
```

However, we can still determine the R-span of vectors in the ambient space, or over-ride the submodule check by setting check to False.

```
sage: X = V.span([ V(B[0]+B[1])/2, V(B[1]-B[2])/2 ], ZZ)
sage: X
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1/2    0   1/2]
[   0  1/2  -1/2]
sage: Y = M.submodule([ V(B[0]+B[1])/2, V(B[1]-B[2])/2 ], check=False)
sage: X == Y
True
```

```
>>> from sage.all import *
>>> X = V.span([ V(B[Integer(0)]+B[Integer(1)])/Integer(2), V(B[Integer(1)]-
↪B[Integer(2)])/Integer(2) ], ZZ)
>>> X
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1/2    0   1/2]
[   0  1/2  -1/2]
>>> Y = M.submodule([ V(B[Integer(0)]+B[Integer(1)])/Integer(2),␣
↪V(B[Integer(1)]-B[Integer(2)])/Integer(2) ], check=False)
>>> X == Y
True
```

Next we try to create a submodule of a free module over the principal ideal domain $\mathbf{Q}[x]$, using our general Hermite normal form implementation:

```
sage: R = PolynomialRing(QQ, 'x'); x = R.gen()
sage: M = FreeModule(R, 3)
sage: B = M.basis()
sage: W = M.submodule_with_basis([x*B[0], 2*B[0]- x*B[2]]); W
Free module of degree 3 and rank 2 over Univariate Polynomial Ring in x over␣
↪Rational Field
User basis matrix:
[ x  0  0]
[ 2  0 -x]
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ, 'x'); x = R.gen()
>>> M = FreeModule(R, Integer(3))
>>> B = M.basis()
>>> W = M.submodule_with_basis([x*B[Integer(0)], Integer(2)*B[Integer(0)]-␣
↪x*B[Integer(2)]]); W
Free module of degree 3 and rank 2 over Univariate Polynomial Ring in x over␣
↪Rational Field
User basis matrix:
[ x  0  0]
[ 2  0 -x]
```

vector_space_span(*gens*, *check=True*)

Create the vector subspace of the ambient vector space with given generators.

INPUT:

- `gens` – list of vector in `self`
- `check` – whether or not to verify that each gen is in the ambient vector space

OUTPUT: a vector subspace

EXAMPLES:

We create a 2-dimensional subspace of $\mathbf{Q}^3$.

```
sage: V = VectorSpace(QQ, 3)
sage: B = V.basis()
```

(continues on next page)

(continued from previous page)

```
sage: W = V.vector_space_span([B[0]+B[1], 2*B[1]-B[2]])
sage: W
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[   1    0  1/2]
[   0    1 -1/2]
```

```
>>> from sage.all import *
>>> V = VectorSpace(QQ, Integer(3))
>>> B = V.basis()
>>> W = V.vector_space_span([B[Integer(0)]+B[Integer(1)],␣
↪Integer(2)*B[Integer(1)]-B[Integer(2)]])
>>> W
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[   1    0  1/2]
[   0    1 -1/2]
```

We create a subspace of a vector space over $\mathbf{Q}(i)$.

```
sage: R.<x> = QQ[]
sage: K = NumberField(x^2 + 1, 'a'); a = K.gen()                                #␣
↪needs sage.rings.number_field
sage: V = VectorSpace(K, 3)                                                     #␣
↪needs sage.rings.number_field
sage: V.vector_space_span([2*V.gen(0) + 3*V.gen(2)])                            #␣
↪needs sage.rings.number_field
Vector space of degree 3 and dimension 1
 over Number Field in a with defining polynomial x^2 + 1
Basis matrix:
[  1   0 3/2]
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> K = NumberField(x**Integer(2) + Integer(1), 'a'); a = K.gen()              ␣
↪               # needs sage.rings.number_field
>>> V = VectorSpace(K, Integer(3))                                             ␣
↪     # needs sage.rings.number_field
>>> V.vector_space_span([Integer(2)*V.gen(Integer(0)) + Integer(3)*V.
↪gen(Integer(2))])                          # needs sage.rings.number_field
Vector space of degree 3 and dimension 1
 over Number Field in a with defining polynomial x^2 + 1
Basis matrix:
[  1   0 3/2]
```

We use the `vector_space_span` command to create a vector subspace of the ambient vector space of a submodule of $\mathbf{Z}^3$.

```
sage: M = FreeModule(ZZ, 3)
sage: W = M.submodule([M([1,2,3])])
sage: W.vector_space_span([M([2,3,4])])
Vector space of degree 3 and dimension 1 over Rational Field
```

(continues on next page)

(continued from previous page)

```
Basis matrix:
[  1 3/2   2]
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ, Integer(3))
>>> W = M.submodule([M([Integer(1),Integer(2),Integer(3)])])
>>> W.vector_space_span([M([Integer(2),Integer(3),Integer(4)])])
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[  1 3/2   2]
```

vector_space_span_of_basis(*basis*, *check=True*)

Create the vector subspace of the ambient vector space with given basis.

INPUT:

- `basis` – list of linearly independent vectors
- `check` – whether or not to verify that each gen is in the ambient vector space

OUTPUT: a vector subspace with user-specified basis

EXAMPLES:

```
sage: V = VectorSpace(QQ, 3)
sage: B = V.basis()
sage: W = V.vector_space_span_of_basis([B[0] + B[1], 2*B[1] - B[2]])
sage: W
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[ 1  1  0]
[ 0  2 -1]
```

```
>>> from sage.all import *
>>> V = VectorSpace(QQ, Integer(3))
>>> B = V.basis()
>>> W = V.vector_space_span_of_basis([B[Integer(0)] + B[Integer(1)],␣
↪Integer(2)*B[Integer(1)] - B[Integer(2)]])
>>> W
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[ 1  1  0]
[ 0  2 -1]
```

zero_submodule()

Return the zero submodule of this module.

EXAMPLES:

```
sage: V = FreeModule(ZZ,2)
sage: V.zero_submodule()
Free module of degree 2 and rank 0 over Integer Ring
Echelon basis matrix:
[]
```

```
>>> from sage.all import *
>>> V = FreeModule(ZZ,Integer(2))
>>> V.zero_submodule()
Free module of degree 2 and rank 0 over Integer Ring
Echelon basis matrix:
[]
```

class sage.modules.free_module.**FreeModule_submodule_field**(*ambient*, *gens*, *check=True*, *already_echelonized=False*, *category=None*)

Bases: *FreeModule_submodule_with_basis_field*

An embedded vector subspace with echelonized basis.

EXAMPLES:

Since this is an embedded vector subspace with echelonized basis, the echelon_coordinates() and user coordinates() agree:

```
sage: V = QQ^3
sage: W = V.span([[1,2,3],[4,5,6]])
sage: W
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3)
>>> W = V.span([[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5),
↪Integer(6)]])
>>> W
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
```

```
sage: v = V([1,5,9])
sage: W.echelon_coordinates(v)
[1, 5]
sage: vector(QQ, W.echelon_coordinates(v)) * W.basis_matrix()
(1, 5, 9)
sage: v = V([1,5,9])
sage: W.coordinates(v)
[1, 5]
sage: vector(QQ, W.coordinates(v)) * W.basis_matrix()
(1, 5, 9)
```

```
>>> from sage.all import *
>>> v = V([Integer(1),Integer(5),Integer(9)])
>>> W.echelon_coordinates(v)
[1, 5]
>>> vector(QQ, W.echelon_coordinates(v)) * W.basis_matrix()
```

(continues on next page)

(continued from previous page)

```
(1, 5, 9)
>>> v = V([Integer(1),Integer(5),Integer(9)])
>>> W.coordinates(v)
[1, 5]
>>> vector(QQ, W.coordinates(v)) * W.basis_matrix()
(1, 5, 9)
```

coordinate_vector(*v*, *check=True*)

Write v in terms of the user basis for `self`.

INPUT:

- `v` – vector
- **`check` – boolean (default: `True`); if `True`, also verify that** v is really in `self`

OUTPUT: list

The output is a list c such that if B is the basis for `self`, then

$$\sum c_i B_i = v.$$

If v is not in `self`, raise an `ArithmeticError` exception.

EXAMPLES:

```
sage: V = QQ^3
sage: W = V.span([[1,2,3],[4,5,6]]); W
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
sage: v = V([1,5,9])
sage: W.coordinate_vector(v)
(1, 5)
sage: W.coordinates(v)
[1, 5]
sage: vector(QQ, W.coordinates(v)) * W.basis_matrix()
(1, 5, 9)
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3)
>>> W = V.span([[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5),
↪Integer(6)]]); W
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
>>> v = V([Integer(1),Integer(5),Integer(9)])
>>> W.coordinate_vector(v)
(1, 5)
>>> W.coordinates(v)
[1, 5]
>>> vector(QQ, W.coordinates(v)) * W.basis_matrix()
(1, 5, 9)
```

```
sage: V = VectorSpace(QQ,5, sparse=True)
sage: W = V.subspace([[0,1,2,0,0], [0,-1,0,0,-1/2]])
sage: W.coordinate_vector([0,0,2,0,-1/2])
(0, 2)
```

```
>>> from sage.all import *
>>> V = VectorSpace(QQ,Integer(5), sparse=True)
>>> W = V.subspace([[Integer(0),Integer(1),Integer(2),Integer(0),Integer(0)],␣
↪[Integer(0),-Integer(1),Integer(0),Integer(0),-Integer(1)/Integer(2)]])
>>> W.coordinate_vector([Integer(0),Integer(0),Integer(2),Integer(0),-
↪Integer(1)/Integer(2)])
(0, 2)
```

echelon_coordinates(*v*, *check=True*)

Write v in terms of the echelonized basis of `self`.

INPUT:

- `v` – vector
- `check` – boolean (default: `True`); if `True`, also verify that v is really in `self`

OUTPUT: list

The output is a list c such that if B is the basis for `self`, then

$$\sum c_i B_i = v.$$

If v is not in `self`, raise an `ArithmeticError` exception.

EXAMPLES:

```
sage: V = QQ^3
sage: W = V.span([[1,2,3],[4,5,6]])
sage: W
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3)
>>> W = V.span([[Integer(1),Integer(2),Integer(3)],[Integer(4),Integer(5),
↪Integer(6)]])
>>> W
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
```

```
sage: v = V([1,5,9])
sage: W.echelon_coordinates(v)
[1, 5]
sage: vector(QQ, W.echelon_coordinates(v)) * W.basis_matrix()
(1, 5, 9)
```

```
>>> from sage.all import *
>>> v = V([Integer(1),Integer(5),Integer(9)])
>>> W.echelon_coordinates(v)
[1, 5]
>>> vector(QQ, W.echelon_coordinates(v)) * W.basis_matrix()
(1, 5, 9)
```

has_user_basis()

Return `True` if the basis of this free module is specified by the user, as opposed to being the default echelon form.

EXAMPLES:

```
sage: V = QQ^3
sage: W = V.subspace([[2,'1/2', 1]])
sage: W.has_user_basis()
False
sage: W = V.subspace_with_basis([[2,'1/2',1]])
sage: W.has_user_basis()
True
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3)
>>> W = V.subspace([[Integer(2),'1/2', Integer(1)]])
>>> W.has_user_basis()
False
>>> W = V.subspace_with_basis([[Integer(2),'1/2',Integer(1)]])
>>> W.has_user_basis()
True
```

class sage.modules.free_module.**FreeModule_submodule_pid**(*ambient*, *gens*, *check=True*, *already_echelonized=False*, *category=None*)

Bases: *FreeModule_submodule_with_basis_pid*

An R-submodule of K^n where K is the fraction field of a principal ideal domain R.

EXAMPLES:

```
sage: M = ZZ^3
sage: W = M.span_of_basis([[1,2,3],[4,5,19]]); W
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[ 1  2  3]
[ 4  5 19]
```

```
>>> from sage.all import *
>>> M = ZZ**Integer(3)
>>> W = M.span_of_basis([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5),Integer(19)]]); W
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[ 1  2  3]
[ 4  5 19]
```

Generic tests, including saving and loading submodules and elements:

```
sage: TestSuite(W).run()
sage: v = W.0 + W.1
sage: TestSuite(v).run()
```

```
>>> from sage.all import *
>>> TestSuite(W).run()
>>> v = W.gen(0) + W.gen(1)
>>> TestSuite(v).run()
```

coordinate_vector(*v*, *check=True*)

Write v in terms of the user basis for `self`.

INPUT:

- `v` – vector
- **check – boolean (default: `True`); if `True`, also verify that** v is really in `self`

OUTPUT: list

The output is a list c such that if B is the basis for `self`, then

$$\sum c_i B_i = v.$$

If v is not in self, raise an `ArithmeticError` exception.

EXAMPLES:

```
sage: V = ZZ^3
sage: W = V.span_of_basis([[1,2,3],[4,5,6]])
sage: W.coordinate_vector([1,5,9])
(5, -1)
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(3)
>>> W = V.span_of_basis([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5),Integer(6)]])
>>> W.coordinate_vector([Integer(1),Integer(5),Integer(9)])
(5, -1)
```

has_user_basis()

Return `True` if the basis of this free module is specified by the user, as opposed to being the default echelon form.

EXAMPLES:

```
sage: A = ZZ^3; A
Ambient free module of rank 3 over the principal ideal domain Integer Ring
sage: A.has_user_basis()
False
sage: W = A.span_of_basis([[2,'1/2',1]])
sage: W.has_user_basis()
True
```

(continues on next page)

(continued from previous page)

```
sage: W = A.span([[2,'1/2',1]])
sage: W.has_user_basis()
False
```

```
>>> from sage.all import *
>>> A = ZZ**Integer(3); A
Ambient free module of rank 3 over the principal ideal domain Integer Ring
>>> A.has_user_basis()
False
>>> W = A.span_of_basis([[Integer(2),'1/2',Integer(1)]])
>>> W.has_user_basis()
True
>>> W = A.span([[Integer(2),'1/2',Integer(1)]])
>>> W.has_user_basis()
False
```

class sage.modules.free_module.**FreeModule_submodule_with_basis_field**(*ambient*, *basis*, *check=True*, *echelonize=False*, *echelonized_basis=None*, *already_echelonized=False*, *category=None*)

Bases: *FreeModule_generic_field*, *FreeModule_submodule_with_basis_pid*

An embedded vector subspace with a distinguished user basis.

EXAMPLES:

```
sage: M = QQ^3; W = M.submodule_with_basis([[1,2,3], [4,5,19]]); W
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[ 1  2  3]
[ 4  5 19]
```

```
>>> from sage.all import *
>>> M = QQ**Integer(3); W = M.submodule_with_basis([[Integer(1),Integer(2),
↪Integer(3)], [Integer(4),Integer(5),Integer(19)]]); W
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[ 1  2  3]
[ 4  5 19]
```

Since this is an embedded vector subspace with a distinguished user basis possibly different than the echelonized basis, the echelon_coordinates() and user coordinates() do not agree:

```
sage: V = QQ^3
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3)
```

```
sage: W = V.submodule_with_basis([[1,2,3], [4,5,6]])
sage: W
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[1 2 3]
[4 5 6]
```

```
>>> from sage.all import *
>>> W = V.submodule_with_basis([[Integer(1),Integer(2),Integer(3)], [Integer(4),
↪Integer(5),Integer(6)]])
>>> W
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[1 2 3]
[4 5 6]
```

```
sage: v = V([1,5,9])
sage: W.echelon_coordinates(v)
[1, 5]
sage: vector(QQ, W.echelon_coordinates(v)) * W.echelonized_basis_matrix()
(1, 5, 9)
```

```
>>> from sage.all import *
>>> v = V([Integer(1),Integer(5),Integer(9)])
>>> W.echelon_coordinates(v)
[1, 5]
>>> vector(QQ, W.echelon_coordinates(v)) * W.echelonized_basis_matrix()
(1, 5, 9)
```

```
sage: v = V([1,5,9])
sage: W.coordinates(v)
[5, -1]
sage: vector(QQ, W.coordinates(v)) * W.basis_matrix()
(1, 5, 9)
```

```
>>> from sage.all import *
>>> v = V([Integer(1),Integer(5),Integer(9)])
>>> W.coordinates(v)
[5, -1]
>>> vector(QQ, W.coordinates(v)) * W.basis_matrix()
(1, 5, 9)
```

Generic tests, including saving and loading submodules and elements:

```
sage: TestSuite(W).run()

sage: K.<x> = FractionField(PolynomialRing(QQ,'x'))
sage: M = K^3; W = M.span_of_basis([[1,1,x]])
sage: TestSuite(W).run()
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> TestSuite(W).run()

>>> K = FractionField(PolynomialRing(QQ,'x'), names=('x',)); (x,) = K._first_
↪ngens(1)
>>> M = K**Integer(3); W = M.span_of_basis([[Integer(1),Integer(1),x]])
>>> TestSuite(W).run()
```

is_ambient()

Return `False` since this is not an ambient module.

EXAMPLES:

```
sage: V = QQ^3
sage: V.is_ambient()
True
sage: W = V.span_of_basis([[1,2,3],[4,5,6]])
sage: W.is_ambient()
False
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3)
>>> V.is_ambient()
True
>>> W = V.span_of_basis([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5),Integer(6)]])
>>> W.is_ambient()
False
```

class sage.modules.free_module.**FreeModule_submodule_with_basis_pid**(*ambient*, *basis*, *check=True*, *echelonize=False*, *echelonized_basis=None*, *already_echelonized=False*, *category=None*)

Bases: *FreeModule_generic_pid*

Construct a submodule of a free module over PID with a distinguished basis.

INPUT:

- `ambient` – ambient free module over a principal ideal domain R, i.e. R^n
- `basis` – list of elements of K^n, where K is the fraction field of R; these elements must be linearly independent and will be used as the default basis of the constructed submodule
- `check` – boolean (default: `True`); if `False`, correctness of the input will not be checked and type conversion may be omitted, use with care
- `echelonize` – (default: `False`) if `True`, `basis` will be echelonized and the result will be used as the default basis of the constructed submodule;
- `echelonized_basis` – (default: `None`) if not `None`, must be the echelonized basis spanning the same submodule as `basis`
- `already_echelonized` – boolean (default: `False`); if `True`, `basis` must be already given in the echelonized form

OUTPUT: R-submodule of K^n with the user-specified `basis`

EXAMPLES:

```
sage: M = ZZ^3
sage: W = M.span_of_basis([[1,2,3], [4,5,6]]); W
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1 2 3]
[4 5 6]
```

```
>>> from sage.all import *
>>> M = ZZ**Integer(3)
>>> W = M.span_of_basis([[Integer(1),Integer(2),Integer(3)], [Integer(4),
↪Integer(5),Integer(6)]]); W
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1 2 3]
[4 5 6]
```

Now we create a submodule of the ambient vector space, rather than `M` itself:

```
sage: W = M.span_of_basis([[1,2,3/2], [4,5,6]]); W
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[  1   2 3/2]
[  4   5   6]
```

```
>>> from sage.all import *
>>> W = M.span_of_basis([[Integer(1),Integer(2),Integer(3)/Integer(2)],␣
↪[Integer(4),Integer(5),Integer(6)]]); W
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[  1   2 3/2]
[  4   5   6]
```

ambient()

Return the ambient module or space for `self`.

EXAMPLES:

```
sage: M = ZZ^3
sage: W = M.span_of_basis([[1,2,3],[4,5,6]]); W
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1 2 3]
[4 5 6]
sage: W.ambient()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> M = ZZ**Integer(3)
>>> W = M.span_of_basis([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5),Integer(6)]]); W
```

(continues on next page)

(continued from previous page)

```
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1 2 3]
[4 5 6]
>>> W.ambient()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

Now we create a submodule of the ambient vector space, rather than `M` itself:

```
sage: W = M.span_of_basis([[1,2,3/2],[4,5,6]]); W
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[  1   2 3/2]
[  4   5   6]
sage: W.ambient()
Vector space of dimension 3 over Rational Field
```

```
>>> from sage.all import *
>>> W = M.span_of_basis([[Integer(1),Integer(2),Integer(3)/Integer(2)],
↪[Integer(4),Integer(5),Integer(6)]]); W
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[  1   2 3/2]
[  4   5   6]
>>> W.ambient()
Vector space of dimension 3 over Rational Field
```

A submodule of a submodule:

```
sage: M = ZZ^3
sage: W = M.span_of_basis([[1,2,3],[4,5,6]]); W
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1 2 3]
[4 5 6]
sage: U = W.span_of_basis([[5,7,9]]); U
Free module of degree 3 and rank 1 over Integer Ring
User basis matrix:
[5 7 9]
sage: U.ambient()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> M = ZZ**Integer(3)
>>> W = M.span_of_basis([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5),Integer(6)]]); W
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1 2 3]
[4 5 6]
>>> U = W.span_of_basis([[Integer(5),Integer(7),Integer(9)]]); U
Free module of degree 3 and rank 1 over Integer Ring
```

(continues on next page)

(continued from previous page)

```
User basis matrix:
[5 7 9]
>>> U.ambient()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

ambient_module()

Return the ambient module related to the R-module self, which was used when creating this module, and is of the form R^n. Note that `self` need not be contained in the ambient module, though `self` will be contained in the ambient vector space.

EXAMPLES:

```
sage: A = ZZ^3
sage: M = A.span_of_basis([[1,2,'3/7'],[4,5,6]])
sage: M
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[  1   2 3/7]
[  4   5   6]
sage: M.ambient_module()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
sage: M.is_submodule(M.ambient_module())
False
```

```
>>> from sage.all import *
>>> A = ZZ**Integer(3)
>>> M = A.span_of_basis([[Integer(1),Integer(2),'3/7'],[Integer(4),Integer(5),
↪Integer(6)]])
>>> M
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[  1   2 3/7]
[  4   5   6]
>>> M.ambient_module()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
>>> M.is_submodule(M.ambient_module())
False
```

ambient_vector_space()

Return the ambient vector space in which this free module is embedded.

EXAMPLES:

```
sage: M = ZZ^3;  M.ambient_vector_space()
Vector space of dimension 3 over Rational Field
```

```
>>> from sage.all import *
>>> M = ZZ**Integer(3);  M.ambient_vector_space()
Vector space of dimension 3 over Rational Field
```

```
sage: N = M.span_of_basis([[1,2,'1/5']])
sage: N
```

(continues on next page)

(continued from previous page)

```
Free module of degree 3 and rank 1 over Integer Ring
User basis matrix:
[  1   2 1/5]
sage: M.ambient_vector_space()
Vector space of dimension 3 over Rational Field
sage: M.ambient_vector_space() is N.ambient_vector_space()
True
```

```
>>> from sage.all import *
>>> N = M.span_of_basis([[Integer(1),Integer(2),'1/5']])
>>> N
Free module of degree 3 and rank 1 over Integer Ring
User basis matrix:
[  1   2 1/5]
>>> M.ambient_vector_space()
Vector space of dimension 3 over Rational Field
>>> M.ambient_vector_space() is N.ambient_vector_space()
True
```

If an inner product on the module is specified, then this is preserved on the ambient vector space.

```
sage: M = FreeModule(ZZ,4,inner_product_matrix=1)
sage: V = M.ambient_vector_space()
sage: V
Ambient quadratic space of dimension 4 over Rational Field
Inner product matrix:
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
sage: N = M.submodule([[1,-1,0,0],[0,1,-1,0],[0,0,1,-1]])
sage: N.gram_matrix()
[2 1 1]
[1 2 1]
[1 1 2]
sage: V == N.ambient_vector_space()
True
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ,Integer(4),inner_product_matrix=Integer(1))
>>> V = M.ambient_vector_space()
>>> V
Ambient quadratic space of dimension 4 over Rational Field
Inner product matrix:
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
>>> N = M.submodule([[Integer(1),-Integer(1),Integer(0),Integer(0)],
↪[Integer(0),Integer(1),-Integer(1),Integer(0)],[Integer(0),Integer(0),
↪Integer(1),-Integer(1)]])
>>> N.gram_matrix()
```

(continues on next page)

(continued from previous page)

```
[2 1 1]
[1 2 1]
[1 1 2]
>>> V == N.ambient_vector_space()
True
```

basis()

Return the user basis for this free module.

EXAMPLES:

```
sage: V = ZZ^3
sage: V.basis()
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
sage: M = V.span_of_basis([['1/8',2,1]])
sage: M.basis()
[(1/8, 2, 1)]
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(3)
>>> V.basis()
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
>>> M = V.span_of_basis([['1/8',Integer(2),Integer(1)]])
>>> M.basis()
[(1/8, 2, 1)]
```

change_ring(R)

Return the free module over R obtained by coercing each element of the basis of `self` into a vector over the fraction field of R, then taking the resulting R-module.

INPUT:

- `R` – a principal ideal domain

EXAMPLES:

```
sage: V = QQ^3
sage: W = V.subspace([[2, 1/2, 1]])
sage: W.change_ring(GF(7))
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 2 4]
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3)
>>> W = V.subspace([[Integer(2), Integer(1)/Integer(2), Integer(1)]])
>>> W.change_ring(GF(Integer(7)))
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 2 4]
```

```
sage: M = (ZZ^2) * (1/2)
sage: N = M.change_ring(QQ)
```

(continues on next page)

(continued from previous page)

```
sage: N
Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1 0]
[0 1]
sage: N = M.change_ring(QQ['x'])
sage: N
Free module of degree 2 and rank 2
 over Univariate Polynomial Ring in x over Rational Field
Echelon basis matrix:
[1/2   0]
[  0 1/2]
sage: N.coordinate_ring()
Univariate Polynomial Ring in x over Rational Field
```

```
>>> from sage.all import *
>>> M = (ZZ**Integer(2)) * (Integer(1)/Integer(2))
>>> N = M.change_ring(QQ)
>>> N
Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1 0]
[0 1]
>>> N = M.change_ring(QQ['x'])
>>> N
Free module of degree 2 and rank 2
 over Univariate Polynomial Ring in x over Rational Field
Echelon basis matrix:
[1/2   0]
[  0 1/2]
>>> N.coordinate_ring()
Univariate Polynomial Ring in x over Rational Field
```

The ring must be a principal ideal domain:

```
sage: M.change_ring(ZZ['x'])
Traceback (most recent call last):
...
TypeError: the new ring Univariate Polynomial Ring in x over Integer Ring
should be a principal ideal domain
```

```
>>> from sage.all import *
>>> M.change_ring(ZZ['x'])
Traceback (most recent call last):
...
TypeError: the new ring Univariate Polynomial Ring in x over Integer Ring
should be a principal ideal domain
```

construction()

Return the functorial construction of `self`, namely, the subspace of the ambient module spanned by the given basis.

EXAMPLES:

```
sage: M = ZZ^3
sage: W = M.span_of_basis([[1,2,3],[4,5,6]]); W
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1 2 3]
[4 5 6]
sage: c, V = W.construction()
sage: c(V) == W
True
```

```
>>> from sage.all import *
>>> M = ZZ**Integer(3)
>>> W = M.span_of_basis([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5),Integer(6)]]); W
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1 2 3]
[4 5 6]
>>> c, V = W.construction()
>>> c(V) == W
True
```

coordinate_vector(*v*, *check=True*)

Write v in terms of the user basis for `self`.

INPUT:

- `v` – vector
- `check` – boolean (default: `True`); if `True`, also verify that v is really in `self`

OUTPUT: list

The output is a vector c such that if B is the basis for `self`, then

$$\sum c_i B_i = v.$$

If v is not in self, raise an `ArithmeticError` exception.

EXAMPLES:

```
sage: V = ZZ^3
sage: M = V.span_of_basis([['1/8',2,1]])
sage: M.coordinate_vector([1,16,8])
(8)
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(3)
>>> M = V.span_of_basis([['1/8',Integer(2),Integer(1)]])
>>> M.coordinate_vector([Integer(1),Integer(16),Integer(8)])
(8)
```

echelon_coordinate_vector(*v*, *check=True*)

Write v in terms of the echelonized basis for `self`.

INPUT:

- v – vector
- check – boolean (default: True); if True, also verify that v is really in self

Returns a list c such that if B is the echelonized basis for self, then

$$\sum c_i B_i = v.$$

If v is not in self, raise an ArithmeticError exception.

EXAMPLES:

```
sage: V = ZZ^3
sage: M = V.span_of_basis([['1/2',3,1], [0,'1/6',0]])
sage: B = M.echelonized_basis(); B
[(1/2, 0, 1), (0, 1/6, 0)]
sage: M.echelon_coordinate_vector(['1/2', 3, 1])
(1, 18)
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(3)
>>> M = V.span_of_basis([['1/2',Integer(3),Integer(1)], [Integer(0),'1/6',
↪Integer(0)]])
>>> B = M.echelonized_basis(); B
[(1/2, 0, 1), (0, 1/6, 0)]
>>> M.echelon_coordinate_vector(['1/2', Integer(3), Integer(1)])
(1, 18)
```

echelon_coordinates(*v*, *check=True*)

Write v in terms of the echelonized basis for self.

INPUT:

- v – vector
- check – boolean (default: True); if True, also verify that v is really in self

OUTPUT: list

Returns a list c such that if B is the basis for self, then

$$\sum c_i B_i = v.$$

If v is not in self, raise an ArithmeticError exception.

EXAMPLES:

```
sage: A = ZZ^3
sage: M = A.span_of_basis([[1,2,'3/7'],[4,5,6]])
sage: M.coordinates([8,10,12])
[0, 2]
sage: M.echelon_coordinates([8,10,12])
[8, -2]
sage: B = M.echelonized_basis(); B
[(1, 2, 3/7), (0, 3, -30/7)]
sage: 8*B[0] - 2*B[1]
(8, 10, 12)
```

```
>>> from sage.all import *
>>> A = ZZ**Integer(3)
>>> M = A.span_of_basis([[Integer(1),Integer(2),'3/7'],[Integer(4),Integer(5),
↪Integer(6)]])
>>> M.coordinates([Integer(8),Integer(10),Integer(12)])
[0, 2]
>>> M.echelon_coordinates([Integer(8),Integer(10),Integer(12)])
[8, -2]
>>> B = M.echelonized_basis(); B
[(1, 2, 3/7), (0, 3, -30/7)]
>>> Integer(8)*B[Integer(0)] - Integer(2)*B[Integer(1)]
(8, 10, 12)
```

We do an example with a sparse vector space:

```
sage: V = VectorSpace(QQ,5, sparse=True)
sage: W = V.subspace_with_basis([[0,1,2,0,0], [0,-1,0,0,-1/2]])
sage: W.echelonized_basis()
[(0, 1, 0, 0, 1/2), (0, 0, 1, 0, -1/4)]
sage: W.echelon_coordinates([0,0,2,0,-1/2])
[0, 2]
```

```
>>> from sage.all import *
>>> V = VectorSpace(QQ,Integer(5), sparse=True)
>>> W = V.subspace_with_basis([[Integer(0),Integer(1),Integer(2),Integer(0),
↪Integer(0)], [Integer(0),-Integer(1),Integer(0),Integer(0),-Integer(1)/
↪Integer(2)]])
>>> W.echelonized_basis()
[(0, 1, 0, 0, 1/2), (0, 0, 1, 0, -1/4)]
>>> W.echelon_coordinates([Integer(0),Integer(0),Integer(2),Integer(0),-
↪Integer(1)/Integer(2)])
[0, 2]
```

echelon_to_user_matrix()

Return matrix that transforms the echelon basis to the user basis of `self`. This is a matrix A such that if v is a vector written with respect to the echelon basis for `self` then vA is that vector written with respect to the user basis of `self`.

EXAMPLES:

```
sage: V = QQ^3
sage: W = V.span_of_basis([[1,2,3],[4,5,6]])
sage: W.echelonized_basis()
[(1, 0, -1), (0, 1, 2)]
sage: A = W.echelon_to_user_matrix(); A
[-5/3  2/3]
[ 4/3 -1/3]
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3)
>>> W = V.span_of_basis([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5),Integer(6)]])
>>> W.echelonized_basis()
```

(continues on next page)

(continued from previous page)

```
[(1, 0, -1), (0, 1, 2)]
>>> A = W.echelon_to_user_matrix(); A
[-5/3  2/3]
[ 4/3 -1/3]
```

The vector $(1, 1, 1)$ has coordinates $v = (1, 1)$ with respect to the echelonized basis for `self`. Multiplying vA we find the coordinates of this vector with respect to the user basis.

```
sage: v = vector(QQ, [1,1]); v
(1, 1)
sage: v * A
(-1/3, 1/3)
sage: u0, u1 = W.basis()
sage: (-u0 + u1)/3
(1, 1, 1)
```

```
>>> from sage.all import *
>>> v = vector(QQ, [Integer(1),Integer(1)]); v
(1, 1)
>>> v * A
(-1/3, 1/3)
>>> u0, u1 = W.basis()
>>> (-u0 + u1)/Integer(3)
(1, 1, 1)
```

echelonized_basis()

Return the basis for `self` in echelon form.

EXAMPLES:

```
sage: V = ZZ^3
sage: M = V.span_of_basis([['1/2',3,1], [0,'1/6',0]])
sage: M.basis()
[(1/2, 3, 1), (0, 1/6, 0)]
sage: B = M.echelonized_basis(); B
[(1/2, 0, 1), (0, 1/6, 0)]
sage: V.span(B) == M
True
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(3)
>>> M = V.span_of_basis([['1/2',Integer(3),Integer(1)], [Integer(0),'1/6',
↪Integer(0)]])
>>> M.basis()
[(1/2, 3, 1), (0, 1/6, 0)]
>>> B = M.echelonized_basis(); B
[(1/2, 0, 1), (0, 1/6, 0)]
>>> V.span(B) == M
True
```

echelonized_basis_matrix()

Return basis matrix for `self` in row echelon form.

EXAMPLES:

```
sage: V = FreeModule(ZZ, 3).span_of_basis([[1,2,3],[4,5,6]])
sage: V.basis_matrix()
[1 2 3]
[4 5 6]
sage: V.echelonized_basis_matrix()
[1 2 3]
[0 3 6]
```

```
>>> from sage.all import *
>>> V = FreeModule(ZZ, Integer(3)).span_of_basis([[Integer(1),Integer(2),
↪Integer(3)],[Integer(4),Integer(5),Integer(6)]])
>>> V.basis_matrix()
[1 2 3]
[4 5 6]
>>> V.echelonized_basis_matrix()
[1 2 3]
[0 3 6]
```

has_user_basis()

Return `True` if the basis of this free module is specified by the user, as opposed to being the default echelon form.

EXAMPLES:

```
sage: V = ZZ^3; V.has_user_basis()
False
sage: M = V.span_of_basis([[1,3,1]]); M.has_user_basis()
True
sage: M = V.span([[1,3,1]]); M.has_user_basis()
False
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(3); V.has_user_basis()
False
>>> M = V.span_of_basis([[Integer(1),Integer(3),Integer(1)]]); M.has_user_
↪basis()
True
>>> M = V.span([[Integer(1),Integer(3),Integer(1)]]); M.has_user_basis()
False
```

lift()

The lift (embedding) map from `self` to the ambient module or space.

EXAMPLES:

```
sage: M = ZZ^3
sage: W = M.span_of_basis([[1,2,3],[4,5,6]]); W
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1 2 3]
[4 5 6]
sage: W.lift
```

(continues on next page)

(continued from previous page)

```
Generic morphism:
From: Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1 2 3]
[4 5 6]
To:   Ambient free module of rank 3 over the principal ideal domain Integer␣
↪Ring
sage: w = W([5,7,9])
sage: m = W.lift(w); m
(5, 7, 9)
sage: m.parent()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> M = ZZ**Integer(3)
>>> W = M.span_of_basis([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5),Integer(6)]]); W
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1 2 3]
[4 5 6]
>>> W.lift
Generic morphism:
From: Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1 2 3]
[4 5 6]
To:   Ambient free module of rank 3 over the principal ideal domain Integer␣
↪Ring
>>> w = W([Integer(5),Integer(7),Integer(9)])
>>> m = W.lift(w); m
(5, 7, 9)
>>> m.parent()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

linear_combination_of_basis(*v*)

Return the linear combination of the basis for `self` obtained from the coordinates of v.

INPUT:

- `v` – list

EXAMPLES:

```
sage: V = span([[1,2,3], [4,5,6]], ZZ); V
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[1 2 3]
[0 3 6]
sage: V.linear_combination_of_basis([1,1])
(1, 5, 9)
```

```
>>> from sage.all import *
>>> V = span([[Integer(1),Integer(2),Integer(3)], [Integer(4),Integer(5),
↪Integer(6)]], ZZ); V
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[1 2 3]
[0 3 6]
>>> V.linear_combination_of_basis([Integer(1),Integer(1)])
(1, 5, 9)
```

This should raise an error if the resulting element is not in self:

```
sage: W = (QQ**2).span([[2, 0], [0, 8]], ZZ)
sage: W.linear_combination_of_basis([1, -1/2])
Traceback (most recent call last):
...
TypeError: element [2, -4] is not in free module
```

```
>>> from sage.all import *
>>> W = (QQ**Integer(2)).span([[Integer(2), Integer(0)], [Integer(0),␣
↪Integer(8)]], ZZ)
>>> W.linear_combination_of_basis([Integer(1), -Integer(1)/Integer(2)])
Traceback (most recent call last):
...
TypeError: element [2, -4] is not in free module
```

relations()

Return the submodule defining the relations of `self` as a subquotient (considering the ambient module as a quotient module).

EXAMPLES:

```
sage: V = GF(2)^2
sage: W = V.subspace([[1, 0]])
sage: W.relations() == V.zero_submodule()
True

sage: Q = V / W
sage: Q.relations() == W
True
sage: Q.zero_submodule().relations() == W
True
```

```
>>> from sage.all import *
>>> V = GF(Integer(2))**Integer(2)
>>> W = V.subspace([[Integer(1), Integer(0)]])
>>> W.relations() == V.zero_submodule()
True

>>> Q = V / W
>>> Q.relations() == W
True
```

(continues on next page)

(continued from previous page)

```
>>> Q.zero_submodule().relations() == W
True
```

retract()

The retract map from the ambient space.

This is a partial map, which gives an error for elements not in the subspace.

Calling this map on elements of the ambient space is the same as calling the element constructor of `self`.

EXAMPLES:

```
sage: M = ZZ^3
sage: W = M.span_of_basis([[1,2,3],[4,5,6]]); W
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1 2 3]
[4 5 6]
sage: W.retract
Generic morphism:
From: Ambient free module of rank 3 over the principal ideal domain Integer␣
↪Ring
To:   Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1 2 3]
[4 5 6]
sage: m = M([5, 7, 9])
sage: w = W.retract(m); w
(5, 7, 9)
sage: w.parent()
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1 2 3]
[4 5 6]
```

```
>>> from sage.all import *
>>> M = ZZ**Integer(3)
>>> W = M.span_of_basis([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5),Integer(6)]]); W
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1 2 3]
[4 5 6]
>>> W.retract
Generic morphism:
From: Ambient free module of rank 3 over the principal ideal domain Integer␣
↪Ring
To:   Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1 2 3]
[4 5 6]
>>> m = M([Integer(5), Integer(7), Integer(9)])
>>> w = W.retract(m); w
```

(continues on next page)

(continued from previous page)

```
(5, 7, 9)
>>> w.parent()
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1 2 3]
[4 5 6]
```

user_to_echelon_matrix()

Return matrix that transforms a vector written with respect to the user basis of `self` to one written with respect to the echelon basis. The matrix acts from the right, as is usual in Sage.

EXAMPLES:

```
sage: A = ZZ^3
sage: M = A.span_of_basis([[1,2,3],[4,5,6]])
sage: M.echelonized_basis()
[(1, 2, 3), (0, 3, 6)]
sage: M.user_to_echelon_matrix()
[ 1  0]
[ 4 -1]
```

```
>>> from sage.all import *
>>> A = ZZ**Integer(3)
>>> M = A.span_of_basis([[Integer(1),Integer(2),Integer(3)],[Integer(4),
↪Integer(5),Integer(6)]])
>>> M.echelonized_basis()
[(1, 2, 3), (0, 3, 6)]
>>> M.user_to_echelon_matrix()
[ 1  0]
[ 4 -1]
```

The vector $v = (5, 7, 9)$ in M is $(1, 1)$ with respect to the user basis. Multiplying the above matrix on the right by this vector yields $(5, -1)$, which has components the coordinates of v with respect to the echelon basis.

```
sage: v0,v1 = M.basis(); v = v0+v1
sage: e0,e1 = M.echelonized_basis()
sage: v
(5, 7, 9)
sage: 5*e0 + (-1)*e1
(5, 7, 9)
```

```
>>> from sage.all import *
>>> v0,v1 = M.basis(); v = v0+v1
>>> e0,e1 = M.echelonized_basis()
>>> v
(5, 7, 9)
>>> Integer(5)*e0 + (-Integer(1))*e1
(5, 7, 9)
```

vector_space(*base_field=None*)

Return the vector space associated to this free module via tensor product with the fraction field of the base ring.

EXAMPLES:

```
sage: A = ZZ^3; A
Ambient free module of rank 3 over the principal ideal domain Integer Ring
sage: A.vector_space()
Vector space of dimension 3 over Rational Field
sage: M = A.span_of_basis([['1/3',2,'3/7'],[4,5,6]]); M
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1/3   2 3/7]
[  4   5   6]
sage: M.vector_space()
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[1/3   2 3/7]
[  4   5   6]
```

```
>>> from sage.all import *
>>> A = ZZ**Integer(3); A
Ambient free module of rank 3 over the principal ideal domain Integer Ring
>>> A.vector_space()
Vector space of dimension 3 over Rational Field
>>> M = A.span_of_basis([['1/3',Integer(2),'3/7'],[Integer(4),Integer(5),
↪Integer(6)]]); M
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1/3   2 3/7]
[  4   5   6]
>>> M.vector_space()
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[1/3   2 3/7]
[  4   5   6]
```

class sage.modules.free_module.**Module_free_ambient**(*base_ring*, *degree*, *sparse=False*, *category=None*)

Bases: *Module*

Base class for modules with elements represented by elements of a free module.

Modules whose elements are represented by elements of a free module (such as submodules, quotients, and subquotients of a free module) should be either a subclass of this class or *FreeModule_generic*, which itself is a subclass of this class. If the modules have bases and ranks, then use *FreeModule_generic*. Otherwise, use this class.

INPUT:

- `base_ring` – a commutative ring
- `degree` – nonnegative integer; degree of the ambient free module
- `sparse` – boolean (default: `False`)
- `category` – category (default: `None`)

If `base_ring` is a field, then the default category is the category of finite-dimensional vector spaces over that field; otherwise it is the category of finite-dimensional free modules over that ring. In addition, the category is intersected with the category of finite enumerated sets if the ring is finite or the rank is 0.

EXAMPLES:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: N = M.submodule([vector([x - y, z]), vector([y * z, x * z])])
sage: N.gens()
[(x - y, z), (y*z, x*z)]
sage: N.degree()
2
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_ngens(3)
>>> M = S**Integer(2)
>>> N = M.submodule([vector([x - y, z]), vector([y * z, x * z])])
>>> N.gens()
[(x - y, z), (y*z, x*z)]
>>> N.degree()
2
```

coordinate_ring()

Return the ring over which the entries of the vectors are defined.

EXAMPLES:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: N = M.submodule([vector([x - y, z]), vector([y * z, x * z])])
sage: N.coordinate_ring()
Multivariate Polynomial Ring in x, y, z over Rational Field
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_
↪ngens(3)
>>> M = S**Integer(2)
>>> N = M.submodule([vector([x - y, z]), vector([y * z, x * z])])
>>> N.coordinate_ring()
Multivariate Polynomial Ring in x, y, z over Rational Field
```

degree()

Return the degree of this free module. This is the dimension of the ambient vector space in which it is embedded.

EXAMPLES:

```
sage: M = FreeModule(ZZ, 10)
sage: W = M.submodule([M.gen(0), 2*M.gen(3) - M.gen(0), M.gen(0) + M.gen(3)])
sage: W.degree()
10
sage: W.rank()
2
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ, Integer(10))
```

(continues on next page)

(continued from previous page)

```
>>> W = M.submodule([M.gen(Integer(0)), Integer(2)*M.gen(Integer(3)) - M.
↪gen(Integer(0)), M.gen(Integer(0)) + M.gen(Integer(3))])
>>> W.degree()
10
>>> W.rank()
2
```

free_resolution(**args*, ***kwds*)

Return a free resolution of `self`.

For input options, see `FreeResolution`.

EXAMPLES:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: N = M.submodule([vector([x - y, z]), vector([y * z, x * z])])
sage: res = N.free_resolution(); res                                    #␣
↪needs sage.libs.singular
S^2 <-- S^2 <-- 0
sage: ascii_art(res.chain_complex())                                    #␣
↪needs sage.libs.singular
            [x - y   y*z]
            [    z   x*z]
 0 <-- C_0 <-------------- C_1 <-- 0
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_
↪ngens(3)
>>> M = S**Integer(2)
>>> N = M.submodule([vector([x - y, z]), vector([y * z, x * z])])
>>> res = N.free_resolution(); res                                      #␣
↪needs sage.libs.singular
S^2 <-- S^2 <-- 0
>>> ascii_art(res.chain_complex())                                      #␣
↪needs sage.libs.singular
            [x - y   y*z]
            [    z   x*z]
 0 <-- C_0 <-------------- C_1 <-- 0
```

graded_free_resolution(**args*, ***kwds*)

Return a graded free resolution of `self`.

For input options, see `GradedFiniteFreeResolution`.

EXAMPLES:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: N = M.submodule([vector([x - y, z]), vector([y * z, x * z])])
sage: N.graded_free_resolution(shifts=[1, -1])                          #␣
↪needs sage.libs.singular
S(-1)⊕S(1) <-- S(-2)⊕S(-3) <-- 0
sage: N.graded_free_resolution(shifts=[2, 3])                           #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.libs.singular
S(-2)⊕S(-3) <-- S(-3)⊕S(-4) <-- 0

sage: N = M.submodule([vector([x^3 - y^6, z^2]), vector([y * z, x])])
sage: N.graded_free_resolution(degrees=[2, 1, 3], shifts=[2, 3])            #␣
↪needs sage.libs.singular
S(-2)⊕S(-3) <-- S(-6)⊕S(-8) <-- 0
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_
↪ngens(3)
>>> M = S**Integer(2)
>>> N = M.submodule([vector([x - y, z]), vector([y * z, x * z])])
>>> N.graded_free_resolution(shifts=[Integer(1), -Integer(1)])               ␣
↪              # needs sage.libs.singular
S(-1)⊕S(1) <-- S(-2)⊕S(-3) <-- 0
>>> N.graded_free_resolution(shifts=[Integer(2), Integer(3)])                ␣
↪              # needs sage.libs.singular
S(-2)⊕S(-3) <-- S(-3)⊕S(-4) <-- 0

>>> N = M.submodule([vector([x**Integer(3) - y**Integer(6), z**Integer(2)]),␣
↪vector([y * z, x])])
>>> N.graded_free_resolution(degrees=[Integer(2), Integer(1), Integer(3)],␣
↪shifts=[Integer(2), Integer(3)])            # needs sage.libs.singular
S(-2)⊕S(-3) <-- S(-6)⊕S(-8) <-- 0
```

is_exact()

Test whether elements of this module are represented exactly.

OUTPUT:

Return `True` if elements of this module are represented exactly, i.e., there is no precision loss when doing arithmetic.

EXAMPLES:

```
sage: (ZZ^2).is_exact()
True
sage: (RR^2).is_exact()
False
```

```
>>> from sage.all import *
>>> (ZZ**Integer(2)).is_exact()
True
>>> (RR**Integer(2)).is_exact()
False
```

is_sparse()

Return `True` if the underlying representation of this module uses sparse vectors, and `False` otherwise.

EXAMPLES:

```
sage: FreeModule(ZZ, 2).is_sparse()
False
sage: FreeModule(ZZ, 2, sparse=True).is_sparse()
True
```

```
>>> from sage.all import *
>>> FreeModule(ZZ, Integer(2)).is_sparse()
False
>>> FreeModule(ZZ, Integer(2), sparse=True).is_sparse()
True
```

is_submodule(*other*)

Return `True` if `self` is a submodule of `other`.

EXAMPLES:

Submodule testing over general rings is not guaranteed to work in all cases. However, it will raise an error when it is unable to determine containment.

The zero module can always be tested:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
sage: N.zero_submodule().is_submodule(M)
True
sage: N.zero_submodule().is_submodule(N)
True
sage: M.zero_submodule().is_submodule(N)
True
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_
↪ngens(3)
>>> M = S**Integer(2)
>>> N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
>>> N.zero_submodule().is_submodule(M)
True
>>> N.zero_submodule().is_submodule(N)
True
>>> M.zero_submodule().is_submodule(N)
True
```

It also respects which module it is constructed from:

```
sage: Q = M.quotient_module(N)
sage: Q.zero_submodule().is_submodule(M)
False
sage: Q.zero_submodule().is_submodule(N)
False
sage: M.zero_submodule().is_submodule(Q)
False
sage: N.zero_submodule().is_submodule(Q)
False
```

```
>>> from sage.all import *
>>> Q = M.quotient_module(N)
>>> Q.zero_submodule().is_submodule(M)
False
>>> Q.zero_submodule().is_submodule(N)
False
>>> M.zero_submodule().is_submodule(Q)
False
>>> N.zero_submodule().is_submodule(Q)
False
```

quotient_module(*sub*, *check=True*)

Return the quotient of `self` by the given subspace `sub`.

INPUT:

- `sub` – a submodule of `self` or something that can be turned into one via `self.submodule(sub)`
- `check` – boolean (default: `True`); whether or not to check that `sub` is a submodule

EXAMPLES:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: N = M.submodule([vector([x - y, z]), vector([y * z, x * z])])
sage: M.quotient(N)
Quotient module by Submodule of Ambient free module of rank 2 over
 the integral domain Multivariate Polynomial Ring in x, y, z over Rational␣
↪Field
Generated by the rows of the matrix:
[x - y     z]
[  y*z   x*z]
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_
↪ngens(3)
>>> M = S**Integer(2)
>>> N = M.submodule([vector([x - y, z]), vector([y * z, x * z])])
>>> M.quotient(N)
Quotient module by Submodule of Ambient free module of rank 2 over
 the integral domain Multivariate Polynomial Ring in x, y, z over Rational␣
↪Field
Generated by the rows of the matrix:
[x - y     z]
[  y*z   x*z]
```

relations_matrix()

Return the matrix of relations of `self`.

EXAMPLES:

```
sage: V = GF(2)^2
sage: V.relations_matrix()
[]
sage: W = V.subspace([[1, 0]])
```

(continues on next page)

(continued from previous page)

```
sage: W.relations_matrix()
[]

sage: Q = V / W
sage: Q.relations_matrix()
[1 0]

sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: M.relations_matrix()
[]

sage: N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
sage: Q = M.quotient_module(N)
sage: Q.relations_matrix()
[x - y     z]
[  y*z   x*z]
```

```
>>> from sage.all import *
>>> V = GF(Integer(2))**Integer(2)
>>> V.relations_matrix()
[]
>>> W = V.subspace([[Integer(1), Integer(0)]])
>>> W.relations_matrix()
[]

>>> Q = V / W
>>> Q.relations_matrix()
[1 0]

>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_
↪ngens(3)
>>> M = S**Integer(2)
>>> M.relations_matrix()
[]

>>> N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
>>> Q = M.quotient_module(N)
>>> Q.relations_matrix()
[x - y     z]
[  y*z   x*z]
```

some_elements()

Return some elements of this free module.

See `TestSuite` for a typical use case.

OUTPUT: an iterator

EXAMPLES:

```
sage: F = FreeModule(ZZ, 2)
sage: tuple(F.some_elements())
```

(continues on next page)

(continued from previous page)

```
((1, 0),
 (1, 1),
 (0, 1),
 (-1, 2),
 (-2, 3),
 ...
 (-49, 50))

sage: F = FreeModule(QQ, 3)
sage: tuple(F.some_elements())
((1, 0, 0),
 (1/2, 1/2, 1/2),
 (1/2, -1/2, 2),
 (-2, 0, 1),
 (-1, 42, 2/3),
 (-2/3, 3/2, -3/2),
 (4/5, -4/5, 5/4),
 ...
 (46/103823, -46/103823, 103823/46))

sage: F = FreeModule(SR, 2)                                                    #␣
↪needs sage.symbolic
sage: tuple(F.some_elements())                                                 #␣
↪needs sage.symbolic
((1, 0), (some_variable, some_variable))
```

```
>>> from sage.all import *
>>> F = FreeModule(ZZ, Integer(2))
>>> tuple(F.some_elements())
((1, 0),
 (1, 1),
 (0, 1),
 (-1, 2),
 (-2, 3),
 ...
 (-49, 50))

>>> F = FreeModule(QQ, Integer(3))
>>> tuple(F.some_elements())
((1, 0, 0),
 (1/2, 1/2, 1/2),
 (1/2, -1/2, 2),
 (-2, 0, 1),
 (-1, 42, 2/3),
 (-2/3, 3/2, -3/2),
 (4/5, -4/5, 5/4),
 ...
 (46/103823, -46/103823, 103823/46))

>>> F = FreeModule(SR, Integer(2))                                              ␣
↪     # needs sage.symbolic
>>> tuple(F.some_elements())                                                   #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.symbolic
((1, 0), (some_variable, some_variable))
```

span(*gens*, *base_ring=None*, *check=True*, *already_echelonized=False*)

Return the R-span of `gens`, where R is the `base_ring`.

The default R is the base ring of `self`. Note that this span need not be a submodule of `self`, nor even of the ambient space. It must, however, be contained in the ambient vector space, i.e., the ambient space tensored with the fraction field of R.

INPUT:

- `gens` – list of vectors
- `base_ring` – (optional) a ring
- `check` – boolean (default: `True`); whether or not to coerce entries of gens into base field
- `already_echelonized` – boolean (default: `False`); set this if you know the gens are already in echelon form

EXAMPLES:

```
sage: V = VectorSpace(GF(7), 3)
sage: W = V.subspace([[2, 3, 4]]); W
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 5 2]
sage: W.span([[1, 1, 1]])
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 1 1]
```

```
>>> from sage.all import *
>>> V = VectorSpace(GF(Integer(7)), Integer(3))
>>> W = V.subspace([[Integer(2), Integer(3), Integer(4)]]); W
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 5 2]
>>> W.span([[Integer(1), Integer(1), Integer(1)]])
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 1 1]
```

Over a general ring:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: M.span([vector([x - y, z]), vector([y*z, x*z])])
Submodule of Ambient free module of rank 2 over the integral domain
 Multivariate Polynomial Ring in x, y, z over Rational Field
Generated by the rows of the matrix:
[x - y     z]
[  y*z   x*z]
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_
↪ngens(3)
>>> M = S**Integer(2)
>>> M.span([vector([x - y, z]), vector([y*z, x*z])])
Submodule of Ambient free module of rank 2 over the integral domain
 Multivariate Polynomial Ring in x, y, z over Rational Field
Generated by the rows of the matrix:
[x - y     z]
[  y*z   x*z]
```

Over a PID:

```
sage: V = FreeModule(ZZ, 3)
sage: W = V.submodule([V.gen(0)])
sage: W.span([V.gen(1)])
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[0 1 0]
sage: W.submodule([V.gen(1)])
Traceback (most recent call last):
...
ArithmeticError: argument gens (= [(0, 1, 0)]) does not generate a submodule␣
↪of self
sage: V.span([[1,0,0], [1/5,4,0], [6,3/4,0]])
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[1/5   0   0]
[  0 1/4   0]
```

```
>>> from sage.all import *
>>> V = FreeModule(ZZ, Integer(3))
>>> W = V.submodule([V.gen(Integer(0))])
>>> W.span([V.gen(Integer(1))])
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[0 1 0]
>>> W.submodule([V.gen(Integer(1))])
Traceback (most recent call last):
...
ArithmeticError: argument gens (= [(0, 1, 0)]) does not generate a submodule␣
↪of self
>>> V.span([[Integer(1),Integer(0),Integer(0)], [Integer(1)/Integer(5),
↪Integer(4),Integer(0)], [Integer(6),Integer(3)/Integer(4),Integer(0)]])
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[1/5   0   0]
[  0 1/4   0]
```

It also works with other things than integers:

```
sage: R.<x> = QQ[]
sage: L = R^1
```

(continues on next page)

(continued from previous page)

```
sage: a = L.span([(1/x,)])
sage: a
Free module of degree 1 and rank 1
 over Univariate Polynomial Ring in x over Rational Field
Echelon basis matrix:
[1/x]
sage: b=L.span([(1/x,)])
sage: a(b.gens()[0])
(1/x)
sage: L2 = R^2
sage: L2.span([[(x^2+x)/(x^2-3*x+2), 1/5], [(x^2+2*x)/(x^2-4*x+3), x]])
Free module of degree 2 and rank 2
 over Univariate Polynomial Ring in x over Rational Field
Echelon basis matrix:
[x/(x^3 - 6*x^2 + 11*x - 6)  2/15*x^2 - 17/75*x - 1/75]
[                         0 x^3 - 11/5*x^2 - 3*x + 4/5]
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> L = R**Integer(1)
>>> a = L.span([(Integer(1)/x,)])
>>> a
Free module of degree 1 and rank 1
 over Univariate Polynomial Ring in x over Rational Field
Echelon basis matrix:
[1/x]
>>> b=L.span([(Integer(1)/x,)])
>>> a(b.gens()[Integer(0)])
(1/x)
>>> L2 = R**Integer(2)
>>> L2.span([[(x**Integer(2)+x)/(x**Integer(2)-Integer(3)*x+Integer(2)),␣
↪Integer(1)/Integer(5)], [(x**Integer(2)+Integer(2)*x)/(x**Integer(2)-
↪Integer(4)*x+Integer(3)), x]])
Free module of degree 2 and rank 2
 over Univariate Polynomial Ring in x over Rational Field
Echelon basis matrix:
[x/(x^3 - 6*x^2 + 11*x - 6)  2/15*x^2 - 17/75*x - 1/75]
[                         0 x^3 - 11/5*x^2 - 3*x + 4/5]
```

Note that the `base_ring` can make a huge difference. We repeat the previous example over the fraction field of R and get a simpler vector space.

```
sage: L2.span([[(x^2+x)/(x^2-3*x+2), 1/5], [(x^2+2*x)/(x^2-4*x+3), x]],
....:         base_ring=R.fraction_field())
Vector space of degree 2 and dimension 2 over
 Fraction Field of Univariate Polynomial Ring in x over Rational Field
Basis matrix:
[1 0]
[0 1]
```

```
>>> from sage.all import *
>>> L2.span([[(x**Integer(2)+x)/(x**Integer(2)-Integer(3)*x+Integer(2)),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(1)/Integer(5)], [(x**Integer(2)+Integer(2)*x)/(x**Integer(2)-
↪Integer(4)*x+Integer(3)), x]],
...          base_ring=R.fraction_field())
Vector space of degree 2 and dimension 2 over
 Fraction Field of Univariate Polynomial Ring in x over Rational Field
Basis matrix:
[1 0]
[0 1]
```

submodule(*gens*, *check=True*, *already_echelonized=False*)

Create the R-submodule of the ambient module with given generators, where R is the base ring of `self`.

INPUT:

- `gens` – list of free module elements or a free module
- `check` – boolean (default: `True`); whether or not to verify that the gens are in `self`

OUTPUT:

The submodule spanned by the vectors in the list `gens`. The basis for the subspace is always put in reduced row echelon form (if possible).

EXAMPLES:

We create a submodule of $\mathbf{Z}^3$:

```
sage: M = FreeModule(ZZ, 3)
sage: B = M.basis()
sage: W = M.submodule([B[0] + B[1], 2*B[1] - B[2]])
sage: W
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1  1  0]
[ 0  2 -1]
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ, Integer(3))
>>> B = M.basis()
>>> W = M.submodule([B[Integer(0)] + B[Integer(1)], Integer(2)*B[Integer(1)] -
↪ B[Integer(2)]])
>>> W
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1  1  0]
[ 0  2 -1]
```

We create a submodule of a submodule:

```
sage: W.submodule([3*B[0] + 3*B[1]])
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[3 3 0]
```

```
>>> from sage.all import *
>>> W.submodule([Integer(3)*B[Integer(0)] + Integer(3)*B[Integer(1)]])
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[3 3 0]
```

We try to create a submodule that is not really a submodule, which results in an `ArithmeticError` exception:

```
sage: W.submodule([B[0] - B[1]])
Traceback (most recent call last):
...
ArithmeticError: argument gens (= [(1, -1, 0)]) does not generate a submodule␣
↪of self
```

```
>>> from sage.all import *
>>> W.submodule([B[Integer(0)] - B[Integer(1)]])
Traceback (most recent call last):
...
ArithmeticError: argument gens (= [(1, -1, 0)]) does not generate a submodule␣
↪of self
```

Next we create a submodule of a free module over the principal ideal domain $\mathbf{Q}[x]$, which uses the general Hermite normal form functionality:

```
sage: R = PolynomialRing(QQ, 'x'); x = R.gen()
sage: M = FreeModule(R, 3)
sage: B = M.basis()
sage: W = M.submodule([x*B[0], 2*B[1] - x*B[2]]); W
Free module of degree 3 and rank 2
 over Univariate Polynomial Ring in x over Rational Field
Echelon basis matrix:
[ x  0  0]
[ 0  2 -x]
sage: W.ambient_module()
Ambient free module of rank 3 over the principal ideal domain
 Univariate Polynomial Ring in x over Rational Field
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ, 'x'); x = R.gen()
>>> M = FreeModule(R, Integer(3))
>>> B = M.basis()
>>> W = M.submodule([x*B[Integer(0)], Integer(2)*B[Integer(1)] -␣
↪x*B[Integer(2)]]); W
Free module of degree 3 and rank 2
 over Univariate Polynomial Ring in x over Rational Field
Echelon basis matrix:
[ x  0  0]
[ 0  2 -x]
>>> W.ambient_module()
Ambient free module of rank 3 over the principal ideal domain
 Univariate Polynomial Ring in x over Rational Field
```

Over a generic ring:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: A = S**2
sage: A.submodule([vector([x - y,z]), vector([y*z, x*z])])
Submodule of Ambient free module of rank 2 over the integral domain
 Multivariate Polynomial Ring in x, y, z over Rational Field
Generated by the rows of the matrix:
[x - y     z]
[  y*z   x*z]
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_
↪ngens(3)
>>> A = S**Integer(2)
>>> A.submodule([vector([x - y,z]), vector([y*z, x*z])])
Submodule of Ambient free module of rank 2 over the integral domain
 Multivariate Polynomial Ring in x, y, z over Rational Field
Generated by the rows of the matrix:
[x - y     z]
[  y*z   x*z]
```

zero()

Return the zero vector in this module.

EXAMPLES:

```
sage: M = FreeModule(ZZ, 2)
sage: M.zero()
(0, 0)
sage: M.span([[1,1]]).zero()
(0, 0)
sage: M.zero_submodule().zero()
(0, 0)
sage: M.zero_submodule().zero().is_mutable()
False
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ, Integer(2))
>>> M.zero()
(0, 0)
>>> M.span([[Integer(1),Integer(1)]]).zero()
(0, 0)
>>> M.zero_submodule().zero()
(0, 0)
>>> M.zero_submodule().zero().is_mutable()
False
```

zero_submodule()

Return the zero submodule of this module.

EXAMPLES:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
```

(continues on next page)

(continued from previous page)

```
sage: M.zero_submodule()
Submodule of Ambient free module of rank 2 over the integral domain␣
↪Multivariate Polynomial Ring in x, y, z over Rational Field
Generated by the rows of the matrix:
[]
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_
↪ngens(3)
>>> M = S**Integer(2)
>>> M.zero_submodule()
Submodule of Ambient free module of rank 2 over the integral domain␣
↪Multivariate Polynomial Ring in x, y, z over Rational Field
Generated by the rows of the matrix:
[]
```

zero_vector()

Return the zero vector in this module.

EXAMPLES:

```
sage: M = FreeModule(ZZ, 2)
sage: M.zero_vector()
(0, 0)
sage: M(0)
(0, 0)
sage: M.span([[1,1]]).zero_vector()
(0, 0)
sage: M.zero_submodule().zero_vector()
(0, 0)
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ, Integer(2))
>>> M.zero_vector()
(0, 0)
>>> M(Integer(0))
(0, 0)
>>> M.span([[Integer(1),Integer(1)]]).zero_vector()
(0, 0)
>>> M.zero_submodule().zero_vector()
(0, 0)
```

class sage.modules.free_module.**RealDoubleVectorSpace_class**(*n*)

Bases: *FreeModule_ambient_field*

coordinates(*v*)

sage.modules.free_module.**VectorSpace**(*K*, *dimension_or_basis_keys*, *sparse*, *inner_product_matrix*, *with_basis=None*, *dimension=False*, *basis_keys=None*, ***args*)

EXAMPLES:

The base can be complicated, as long as it is a field.

```
sage: V = VectorSpace(FractionField(PolynomialRing(ZZ,'x')),3)
sage: V
Vector space of dimension 3 over Fraction Field of Univariate Polynomial Ring in␣
↪x over Integer Ring
sage: V.basis()
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
```

```
>>> from sage.all import *
>>> V = VectorSpace(FractionField(PolynomialRing(ZZ,'x')),Integer(3))
>>> V
Vector space of dimension 3 over Fraction Field of Univariate Polynomial Ring in␣
↪x over Integer Ring
>>> V.basis()
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
```

The base must be a field or a `TypeError` is raised.

```
sage: VectorSpace(ZZ,5)
Traceback (most recent call last):
...
TypeError: Argument K (= Integer Ring) must be a field.
```

```
>>> from sage.all import *
>>> VectorSpace(ZZ,Integer(5))
Traceback (most recent call last):
...
TypeError: Argument K (= Integer Ring) must be a field.
```

sage.modules.free_module.**basis_seq**(*V*, *vecs*)

This converts a list vecs of vectors in V to a Sequence of immutable vectors.

Should it? I.e. in most `other` parts of the system the return type of basis or generators is a tuple.

EXAMPLES:

```
sage: V = VectorSpace(QQ,2)
sage: B = V.gens()
sage: B
((1, 0), (0, 1))
sage: v = B[0]
sage: v[0] = 0 # immutable
Traceback (most recent call last):
...
ValueError: vector is immutable; please change a copy instead (use copy())
sage: sage.modules.free_module.basis_seq(V, V.gens())
[(1, 0), (0, 1)]
```

```
>>> from sage.all import *
>>> V = VectorSpace(QQ,Integer(2))
>>> B = V.gens()
>>> B
((1, 0), (0, 1))
>>> v = B[Integer(0)]
```

(continues on next page)

(continued from previous page)

```
>>> v[Integer(0)] = Integer(0) # immutable
Traceback (most recent call last):
...
ValueError: vector is immutable; please change a copy instead (use copy())
>>> sage.modules.free_module.basis_seq(V, V.gens())
[(1, 0), (0, 1)]
```

sage.modules.free_module.**element_class**(*R*, *is_sparse*)

The class of the vectors (elements of a free module) with base ring R and boolean is_sparse.

EXAMPLES:

```
sage: FF = FiniteField(2)
sage: P = PolynomialRing(FF,'x')
sage: sage.modules.free_module.element_class(QQ, is_sparse=True)
<class 'sage.modules.free_module_element.FreeModuleElement_generic_sparse'>
sage: sage.modules.free_module.element_class(QQ, is_sparse=False)
<class 'sage.modules.vector_rational_dense.Vector_rational_dense'>
sage: sage.modules.free_module.element_class(ZZ, is_sparse=True)
<class 'sage.modules.free_module_element.FreeModuleElement_generic_sparse'>
sage: sage.modules.free_module.element_class(ZZ, is_sparse=False)
<class 'sage.modules.vector_integer_dense.Vector_integer_dense'>
sage: sage.modules.free_module.element_class(FF, is_sparse=True)
<class 'sage.modules.free_module_element.FreeModuleElement_generic_sparse'>
sage: sage.modules.free_module.element_class(FF, is_sparse=False)                #␣
↪needs sage.rings.finite_rings
<class 'sage.modules.vector_mod2_dense.Vector_mod2_dense'>
sage: sage.modules.free_module.element_class(GF(7), is_sparse=False)
<class 'sage.modules.vector_modn_dense.Vector_modn_dense'>
sage: sage.modules.free_module.element_class(P, is_sparse=True)
<class 'sage.modules.free_module_element.FreeModuleElement_generic_sparse'>
sage: sage.modules.free_module.element_class(P, is_sparse=False)
<class 'sage.modules.free_module_element.FreeModuleElement_generic_dense'>
```

```
>>> from sage.all import *
>>> FF = FiniteField(Integer(2))
>>> P = PolynomialRing(FF,'x')
>>> sage.modules.free_module.element_class(QQ, is_sparse=True)
<class 'sage.modules.free_module_element.FreeModuleElement_generic_sparse'>
>>> sage.modules.free_module.element_class(QQ, is_sparse=False)
<class 'sage.modules.vector_rational_dense.Vector_rational_dense'>
>>> sage.modules.free_module.element_class(ZZ, is_sparse=True)
<class 'sage.modules.free_module_element.FreeModuleElement_generic_sparse'>
>>> sage.modules.free_module.element_class(ZZ, is_sparse=False)
<class 'sage.modules.vector_integer_dense.Vector_integer_dense'>
>>> sage.modules.free_module.element_class(FF, is_sparse=True)
<class 'sage.modules.free_module_element.FreeModuleElement_generic_sparse'>
>>> sage.modules.free_module.element_class(FF, is_sparse=False)                #␣
↪needs sage.rings.finite_rings
<class 'sage.modules.vector_mod2_dense.Vector_mod2_dense'>
>>> sage.modules.free_module.element_class(GF(Integer(7)), is_sparse=False)
<class 'sage.modules.vector_modn_dense.Vector_modn_dense'>
```

(continues on next page)

(continued from previous page)

```
>>> sage.modules.free_module.element_class(P, is_sparse=True)
<class 'sage.modules.free_module_element.FreeModuleElement_generic_sparse'>
>>> sage.modules.free_module.element_class(P, is_sparse=False)
<class 'sage.modules.free_module_element.FreeModuleElement_generic_dense'>
```

sage.modules.free_module.**is_FreeModule**(*M*)

Return `True` if M inherits from `FreeModule_generic`.

EXAMPLES:

```
sage: from sage.modules.free_module import is_FreeModule
sage: V = ZZ^3
sage: is_FreeModule(V)
doctest:warning...
DeprecationWarning: the function is_FreeModule is deprecated;
use 'isinstance(..., FreeModule_generic)' instead
See https://github.com/sagemath/sage/issues/37924 for details.
True
sage: W = V.span([ V.random_element() for i in range(2) ])
sage: is_FreeModule(W)
True
```

```
>>> from sage.all import *
>>> from sage.modules.free_module import is_FreeModule
>>> V = ZZ**Integer(3)
>>> is_FreeModule(V)
doctest:warning...
DeprecationWarning: the function is_FreeModule is deprecated;
use 'isinstance(..., FreeModule_generic)' instead
See https://github.com/sagemath/sage/issues/37924 for details.
True
>>> W = V.span([ V.random_element() for i in range(Integer(2)) ])
>>> is_FreeModule(W)
True
```

sage.modules.free_module.**span**(*gens*, *base_ring=None*, *check=True*, *already_echelonized=False*)

Return the span of the vectors in `gens` using scalars from `base_ring`.

INPUT:

- `gens` – list of either vectors or lists of ring elements used to generate the span
- `base_ring` – (default: `None`) a principal ideal domain for the ring of scalars
- `check` – (default: `True`) passed to the `span()` method of the ambient module
- `already_echelonized` – (default: `False`) set to `True` if the vectors form the rows of a matrix in echelon form, in order to skip the computation of an echelonized basis for the span.

OUTPUT:

A module (or vector space) that is all the linear combinations of the free module elements (or vectors) with scalars from the ring (or field) given by `base_ring`. See the examples below describing behavior when the base ring is not specified and/or the module elements are given as lists that do not carry explicit base ring information.

EXAMPLES:

The vectors in the list of generators can be given as lists, provided a base ring is specified and the elements of the list are in the ring (or the fraction field of the ring). If the base ring is a field, the span is a vector space.

```
sage: V = span([[1,2,5], [2,2,2]], QQ); V
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -3]
[ 0  1  4]

sage: span([V.gen(0)], QuadraticField(-7,'a'))                                  #␣
↪needs sage.rings.number_field
Vector space of degree 3 and dimension 1 over Number Field in a
 with defining polynomial x^2 + 7 with a = 2.645751311064591?*I
Basis matrix:
[ 1  0 -3]

sage: span([[1,2,3], [2,2,2], [1,2,5]], GF(2))
Vector space of degree 3 and dimension 1 over Finite Field of size 2
Basis matrix:
[1 0 1]
```

```
>>> from sage.all import *
>>> V = span([[Integer(1),Integer(2),Integer(5)], [Integer(2),Integer(2),
↪Integer(2)]], QQ); V
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -3]
[ 0  1  4]

>>> span([V.gen(Integer(0))], QuadraticField(-Integer(7),'a'))                  ␣
↪              # needs sage.rings.number_field
Vector space of degree 3 and dimension 1 over Number Field in a
 with defining polynomial x^2 + 7 with a = 2.645751311064591?*I
Basis matrix:
[ 1  0 -3]

>>> span([[Integer(1),Integer(2),Integer(3)], [Integer(2),Integer(2),Integer(2)],␣
↪[Integer(1),Integer(2),Integer(5)]], GF(Integer(2)))
Vector space of degree 3 and dimension 1 over Finite Field of size 2
Basis matrix:
[1 0 1]
```

If the base ring is not a field, then a module is created. The entries of the vectors can lie outside the ring, if they are in the fraction field of the ring.

```
sage: span([[1,2,5], [2,2,2]], ZZ)
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1  0 -3]
[ 0  2  8]

sage: span([[1,1,1], [1,1/2,1]], ZZ)
Free module of degree 3 and rank 2 over Integer Ring
```

(continues on next page)

(continued from previous page)

```
Echelon basis matrix:
[  1   0   1]
[  0 1/2   0]

sage: R.<x> = QQ[]
sage: M= span( [[x, x^2+1], [1/x, x^3]], R); M
Free module of degree 2 and rank 2 over
Univariate Polynomial Ring in x over Rational Field
Echelon basis matrix:
[          1/x           x^3]
[            0 x^5 - x^2 - 1]
sage: M.basis()[0][0].parent()
Fraction Field of Univariate Polynomial Ring in x over Rational Field
```

```
>>> from sage.all import *
>>> span([[Integer(1),Integer(2),Integer(5)], [Integer(2),Integer(2),Integer(2)]],
↪ ZZ)
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1  0 -3]
[ 0  2  8]

>>> span([[Integer(1),Integer(1),Integer(1)], [Integer(1),Integer(1)/Integer(2),
↪Integer(1)]], ZZ)
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[  1   0   1]
[  0 1/2   0]

>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> M= span( [[x, x**Integer(2)+Integer(1)], [Integer(1)/x, x**Integer(3)]], R); M
Free module of degree 2 and rank 2 over
Univariate Polynomial Ring in x over Rational Field
Echelon basis matrix:
[          1/x           x^3]
[            0 x^5 - x^2 - 1]
>>> M.basis()[Integer(0)][Integer(0)].parent()
Fraction Field of Univariate Polynomial Ring in x over Rational Field
```

A base ring can be inferred if the generators are given as a list of vectors.

```
sage: span([vector(QQ, [1,2,3]), vector(QQ, [4,5,6])])
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
sage: span([vector(QQ, [1,2,3]), vector(ZZ, [4,5,6])])
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
sage: span([vector(ZZ, [1,2,3]), vector(ZZ, [4,5,6])])
```

(continues on next page)

(continued from previous page)

```
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[1 2 3]
[0 3 6]
```

```
>>> from sage.all import *
>>> span([vector(QQ, [Integer(1),Integer(2),Integer(3)]), vector(QQ, [Integer(4),
↪Integer(5),Integer(6)])])
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
>>> span([vector(QQ, [Integer(1),Integer(2),Integer(3)]), vector(ZZ, [Integer(4),
↪Integer(5),Integer(6)])])
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
>>> span([vector(ZZ, [Integer(1),Integer(2),Integer(3)]), vector(ZZ, [Integer(4),
↪Integer(5),Integer(6)])])
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[1 2 3]
[0 3 6]
```

2.3 Elements of free modules

AUTHORS:

- William Stein
- Josh Kantor
- Thomas Feulner (2012-11): Added *FreeModuleElement.hamming_weight()* and *FreeModuleElement_generic_sparse.hamming_weight()*
- Jeroen Demeyer (2015-02-24): Implement fast Cython methods `get_unsafe` and `set_unsafe` similar to other places in Sage (Issue #17562)

EXAMPLES: We create a vector space over **Q** and a subspace of this space.

```
sage: V = QQ^5
sage: W = V.span([V.1, V.2])
```

```
>>> from sage.all import *
>>> V = QQ**Integer(5)
>>> W = V.span([V.gen(1), V.gen(2)])
```

Arithmetic operations always return something in the ambient space, since there is a canonical map from W to V but not from V to W.

```
sage: parent(W.0 + V.1)
Vector space of dimension 5 over Rational Field
```

(continues on next page)

(continued from previous page)

```
sage: parent(V.1 + W.0)
Vector space of dimension 5 over Rational Field
sage: W.0 + V.1
(0, 2, 0, 0, 0)
sage: W.0 - V.0
(-1, 1, 0, 0, 0)
```

```
>>> from sage.all import *
>>> parent(W.gen(0) + V.gen(1))
Vector space of dimension 5 over Rational Field
>>> parent(V.gen(1) + W.gen(0))
Vector space of dimension 5 over Rational Field
>>> W.gen(0) + V.gen(1)
(0, 2, 0, 0, 0)
>>> W.gen(0) - V.gen(0)
(-1, 1, 0, 0, 0)
```

Next we define modules over **Z** and a finite field.

```
sage: K = ZZ^5
sage: M = GF(7)^5
```

```
>>> from sage.all import *
>>> K = ZZ**Integer(5)
>>> M = GF(Integer(7))**Integer(5)
```

Arithmetic between the **Q** and **Z** modules is defined, and the result is always over **Q**, since there is a canonical coercion map to **Q**.

```
sage: K.0 + V.1
(1, 1, 0, 0, 0)
sage: parent(K.0 + V.1)
Vector space of dimension 5 over Rational Field
```

```
>>> from sage.all import *
>>> K.gen(0) + V.gen(1)
(1, 1, 0, 0, 0)
>>> parent(K.gen(0) + V.gen(1))
Vector space of dimension 5 over Rational Field
```

Since there is no canonical coercion map to the finite field from **Q** the following arithmetic is not defined:

```
sage: V.0 + M.0
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +:
 'Vector space of dimension 5 over Rational Field' and
 'Vector space of dimension 5 over Finite Field of size 7'
```

```
>>> from sage.all import *
>>> V.gen(0) + M.gen(0)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +:
 'Vector space of dimension 5 over Rational Field' and
 'Vector space of dimension 5 over Finite Field of size 7'
```

However, there is a map from **Z** to the finite field, so the following is defined, and the result is in the finite field.

```
sage: w = K.0 + M.0; w
(2, 0, 0, 0, 0)
sage: parent(w)
Vector space of dimension 5 over Finite Field of size 7
sage: parent(M.0 + K.0)
Vector space of dimension 5 over Finite Field of size 7
```

```
>>> from sage.all import *
>>> w = K.gen(0) + M.gen(0); w
(2, 0, 0, 0, 0)
>>> parent(w)
Vector space of dimension 5 over Finite Field of size 7
>>> parent(M.gen(0) + K.gen(0))
Vector space of dimension 5 over Finite Field of size 7
```

Matrix vector multiply:

```
sage: MS = MatrixSpace(QQ, 3)
sage: A = MS([0,1,0,1,0,0,0,0,1])
sage: V = QQ^3
sage: v = V([1,2,3])
sage: v * A
(2, 1, 3)
```

```
>>> from sage.all import *
>>> MS = MatrixSpace(QQ, Integer(3))
>>> A = MS([Integer(0),Integer(1),Integer(0),Integer(1),Integer(0),Integer(0),
↪Integer(0),Integer(0),Integer(1)])
>>> V = QQ**Integer(3)
>>> v = V([Integer(1),Integer(2),Integer(3)])
>>> v * A
(2, 1, 3)
```

class sage.modules.free_module_element.**FreeModuleElement**

Bases: Vector

An element of a generic free module.

Mod(*p*)

EXAMPLES:

```
sage: V = vector(ZZ, [5, 9, 13, 15])
sage: V.Mod(7)
(5, 2, 6, 1)
```

(continues on next page)

(continued from previous page)

```
sage: parent(V.Mod(7))
Vector space of dimension 4 over Ring of integers modulo 7
```

```
>>> from sage.all import *
>>> V = vector(ZZ, [Integer(5), Integer(9), Integer(13), Integer(15)])
>>> V.Mod(Integer(7))
(5, 2, 6, 1)
>>> parent(V.Mod(Integer(7)))
Vector space of dimension 4 over Ring of integers modulo 7
```

additive_order()

Return the additive order of `self`.

EXAMPLES:

```
sage: v = vector(Integers(4), [1,2])
sage: v.additive_order()
4
```

```
>>> from sage.all import *
>>> v = vector(Integers(Integer(4)), [Integer(1),Integer(2)])
>>> v.additive_order()
4
```

```
sage: v = vector([1,2,3])
sage: v.additive_order()
+Infinity
```

```
>>> from sage.all import *
>>> v = vector([Integer(1),Integer(2),Integer(3)])
>>> v.additive_order()
+Infinity
```

```
sage: v = vector(Integers(30), [6, 15]); v
(6, 15)
sage: v.additive_order()
10
sage: 10*v
(0, 0)
```

```
>>> from sage.all import *
>>> v = vector(Integers(Integer(30)), [Integer(6), Integer(15)]); v
(6, 15)
>>> v.additive_order()
10
>>> Integer(10)*v
(0, 0)
```

apply_map(*phi*, *R=None*, *sparse=None*)

Apply the given map phi (an arbitrary Python function or callable object) to this free module element. If R is not given, automatically determine the base ring of the resulting element.

INPUT:

- `sparse` – boolean; will control whether the result is sparse. By default, the result is sparse iff self is sparse.
- `phi` – arbitrary Python function or callable object
- `R` – (optional) ring

OUTPUT: a free module element over R

EXAMPLES:

```
sage: m = vector([1,x,sin(x+1)])                                                #␣
↪needs sage.symbolic
sage: m.apply_map(lambda x: x^2)                                                #␣
↪needs sage.symbolic
(1, x^2, sin(x + 1)^2)
sage: m.apply_map(sin)                                                          #␣
↪needs sage.symbolic
(sin(1), sin(x), sin(sin(x + 1)))
```

```
>>> from sage.all import *
>>> m = vector([Integer(1),x,sin(x+Integer(1))])                                ␣
↪              # needs sage.symbolic
>>> m.apply_map(lambda x: x**Integer(2))                                        ␣
↪      # needs sage.symbolic
(1, x^2, sin(x + 1)^2)
>>> m.apply_map(sin)                                                           #␣
↪needs sage.symbolic
(sin(1), sin(x), sin(sin(x + 1)))
```

```
sage: m = vector(ZZ, 9, range(9))
sage: k.<a> = GF(9)                                                             #␣
↪needs sage.rings.finite_rings
sage: m.apply_map(k)                                                            #␣
↪needs sage.rings.finite_rings
(0, 1, 2, 0, 1, 2, 0, 1, 2)
```

```
>>> from sage.all import *
>>> m = vector(ZZ, Integer(9), range(Integer(9)))
>>> k = GF(Integer(9), names=('a',)); (a,) = k._first_ngens(1)# needs sage.
↪rings.finite_rings
>>> m.apply_map(k)                                                             #␣
↪needs sage.rings.finite_rings
(0, 1, 2, 0, 1, 2, 0, 1, 2)
```

In this example, we explicitly specify the codomain.

```
sage: s = GF(3)
sage: f = lambda x: s(x)
sage: n = m.apply_map(f, k); n                                                  #␣
↪needs sage.rings.finite_rings
(0, 1, 2, 0, 1, 2, 0, 1, 2)
sage: n.parent()                                                                #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.rings.finite_rings
Vector space of dimension 9 over Finite Field in a of size 3^2
```

```
>>> from sage.all import *
>>> s = GF(Integer(3))
>>> f = lambda x: s(x)
>>> n = m.apply_map(f, k); n                                                    #␣
↪needs sage.rings.finite_rings
(0, 1, 2, 0, 1, 2, 0, 1, 2)
>>> n.parent()                                                                  #␣
↪needs sage.rings.finite_rings
Vector space of dimension 9 over Finite Field in a of size 3^2
```

If your map sends 0 to a nonzero value, then your resulting vector is not mathematically sparse:

```
sage: v = vector([0] * 6 + [1], sparse=True); v
(0, 0, 0, 0, 0, 0, 1)
sage: v2 = v.apply_map(lambda x: x+1); v2
(1, 1, 1, 1, 1, 1, 2)
```

```
>>> from sage.all import *
>>> v = vector([Integer(0)] * Integer(6) + [Integer(1)], sparse=True); v
(0, 0, 0, 0, 0, 0, 1)
>>> v2 = v.apply_map(lambda x: x+Integer(1)); v2
(1, 1, 1, 1, 1, 1, 2)
```

but it's still represented with a sparse data type:

```
sage: parent(v2)
Ambient sparse free module of rank 7 over the principal ideal domain Integer␣
↪Ring
```

```
>>> from sage.all import *
>>> parent(v2)
Ambient sparse free module of rank 7 over the principal ideal domain Integer␣
↪Ring
```

This data type is inefficient for dense vectors, so you may want to specify sparse=False:

```
sage: v2 = v.apply_map(lambda x: x+1, sparse=False); v2
(1, 1, 1, 1, 1, 1, 2)
sage: parent(v2)
Ambient free module of rank 7 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> v2 = v.apply_map(lambda x: x+Integer(1), sparse=False); v2
(1, 1, 1, 1, 1, 1, 2)
>>> parent(v2)
Ambient free module of rank 7 over the principal ideal domain Integer Ring
```

Or if you have a map that will result in mostly zeroes, you may want to specify sparse=True:

```
sage: v = vector(srange(10))
sage: v2 = v.apply_map(lambda x: 0 if x else 1, sparse=True); v2
(1, 0, 0, 0, 0, 0, 0, 0, 0, 0)
sage: parent(v2)
Ambient sparse free module of rank 10 over the principal ideal domain Integer␣
↪Ring
```

```
>>> from sage.all import *
>>> v = vector(srange(Integer(10)))
>>> v2 = v.apply_map(lambda x: Integer(0) if x else Integer(1), sparse=True);␣
↪v2
(1, 0, 0, 0, 0, 0, 0, 0, 0, 0)
>>> parent(v2)
Ambient sparse free module of rank 10 over the principal ideal domain Integer␣
↪Ring
```

change_ring(*R*)

Change the base ring of this vector.

EXAMPLES:

```
sage: v = vector(QQ['x,y'], [1..5]); v.change_ring(GF(3))
(1, 2, 0, 1, 2)
```

```
>>> from sage.all import *
>>> v = vector(QQ['x,y'], (ellipsis_range(Integer(1),Ellipsis,Integer(5))));␣
↪v.change_ring(GF(Integer(3)))
(1, 2, 0, 1, 2)
```

column()

Return a matrix with a single column and the same entries as the vector `self`.

OUTPUT:

A matrix over the same ring as the vector (or free module element), with a single column. The entries of the column are identical to those of the vector, and in the same order.

EXAMPLES:

```
sage: v = vector(ZZ, [1,2,3])
sage: w = v.column(); w
[1]
[2]
[3]
sage: w.parent()
Full MatrixSpace of 3 by 1 dense matrices over Integer Ring

sage: x = vector(FiniteField(13), [2,4,8,16])
sage: x.column()
[2]
[4]
[8]
[3]
```

```
>>> from sage.all import *
>>> v = vector(ZZ, [Integer(1),Integer(2),Integer(3)])
>>> w = v.column(); w
[1]
[2]
[3]
>>> w.parent()
Full MatrixSpace of 3 by 1 dense matrices over Integer Ring

>>> x = vector(FiniteField(Integer(13)), [Integer(2),Integer(4),Integer(8),
↪Integer(16)])
>>> x.column()
[2]
[4]
[8]
[3]
```

There is more than one way to get one-column matrix from a vector. The `column` method is about equally efficient to making a row and then taking a transpose. Notice that supplying a vector to the matrix constructor demonstrates Sage's preference for rows.

```
sage: x = vector(RDF, [sin(i*pi/20) for i in range(10)])                    #␣
↪needs sage.libs.pari sage.symbolic
sage: x.column() == matrix(x).transpose()
True
sage: x.column() == x.row().transpose()
True
```

```
>>> from sage.all import *
>>> x = vector(RDF, [sin(i*pi/Integer(20)) for i in range(Integer(10))])     ␣
↪              # needs sage.libs.pari sage.symbolic
>>> x.column() == matrix(x).transpose()
True
>>> x.column() == x.row().transpose()
True
```

Sparse or dense implementations are preserved.

```
sage: d = vector(RR, [1.0, 2.0, 3.0])
sage: s = vector(CDF, {2: 5.0+6.0*I})                                       #␣
↪needs sage.symbolic
sage: dm = d.column()
sage: sm = s.column()                                                       #␣
↪needs sage.symbolic
sage: all([d.is_dense(), dm.is_dense(), s.is_sparse(), sm.is_sparse()])     #␣
↪needs sage.symbolic
True
```

```
>>> from sage.all import *
>>> d = vector(RR, [RealNumber('1.0'), RealNumber('2.0'), RealNumber('3.0')])
>>> s = vector(CDF, {Integer(2): RealNumber('5.0')+RealNumber('6.0')*I})     ␣
↪                                  # needs sage.symbolic
>>> dm = d.column()
```

(continues on next page)

(continued from previous page)

```
>>> sm = s.column()                                                    #␣
↪needs sage.symbolic
>>> all([d.is_dense(), dm.is_dense(), s.is_sparse(), sm.is_sparse()])  #␣
↪needs sage.symbolic
True
```

concatenate(*other*, *ring=None*)

Return the result of concatenating this vector with a sequence of elements given by another iterable.

If the optional keyword argument `ring` is passed, this method will return a vector over the specified ring (or fail). If no base ring is given, the base ring is determined automatically by the *vector()* constructor.

EXAMPLES:

```
sage: v = vector([1, 2, 3])
sage: w = vector([4, 5])
sage: v.concatenate(w)
(1, 2, 3, 4, 5)
sage: v.parent()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
sage: w.parent()
Ambient free module of rank 2 over the principal ideal domain Integer Ring
sage: v.concatenate(w).parent()
Ambient free module of rank 5 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> v = vector([Integer(1), Integer(2), Integer(3)])
>>> w = vector([Integer(4), Integer(5)])
>>> v.concatenate(w)
(1, 2, 3, 4, 5)
>>> v.parent()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
>>> w.parent()
Ambient free module of rank 2 over the principal ideal domain Integer Ring
>>> v.concatenate(w).parent()
Ambient free module of rank 5 over the principal ideal domain Integer Ring
```

Forcing a base ring is possible using the `ring` argument:

```
sage: v.concatenate(w, ring=QQ)
(1, 2, 3, 4, 5)
sage: v.concatenate(w, ring=QQ).parent()
Vector space of dimension 5 over Rational Field
```

```
>>> from sage.all import *
>>> v.concatenate(w, ring=QQ)
(1, 2, 3, 4, 5)
>>> v.concatenate(w, ring=QQ).parent()
Vector space of dimension 5 over Rational Field
```

```
sage: v.concatenate(w, ring=Zmod(3))
(1, 2, 0, 1, 2)
```

```
>>> from sage.all import *
>>> v.concatenate(w, ring=Zmod(Integer(3)))
(1, 2, 0, 1, 2)
```

The method accepts arbitrary iterables of elements which can be coerced to a common base ring:

```
sage: v.concatenate(range(4,8))
(1, 2, 3, 4, 5, 6, 7)
sage: v.concatenate(range(4,8)).parent()
Ambient free module of rank 7 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> v.concatenate(range(Integer(4),Integer(8)))
(1, 2, 3, 4, 5, 6, 7)
>>> v.concatenate(range(Integer(4),Integer(8))).parent()
Ambient free module of rank 7 over the principal ideal domain Integer Ring
```

```
sage: w2 = [4, QQbar(-5).sqrt()]
sage: v.concatenate(w2)
(1, 2, 3, 4, 2.236...*I)
sage: v.concatenate(w2).parent()
Vector space of dimension 5 over Algebraic Field
sage: w2 = vector(w2)
sage: v.concatenate(w2)
(1, 2, 3, 4, 2.236...*I)
sage: v.concatenate(w2).parent()
Vector space of dimension 5 over Algebraic Field
```

```
>>> from sage.all import *
>>> w2 = [Integer(4), QQbar(-Integer(5)).sqrt()]
>>> v.concatenate(w2)
(1, 2, 3, 4, 2.236...*I)
>>> v.concatenate(w2).parent()
Vector space of dimension 5 over Algebraic Field
>>> w2 = vector(w2)
>>> v.concatenate(w2)
(1, 2, 3, 4, 2.236...*I)
>>> v.concatenate(w2).parent()
Vector space of dimension 5 over Algebraic Field
```

```
sage: w2 = polygen(QQ)^4 + 5
sage: v.concatenate(w2)
(1, 2, 3, 5, 0, 0, 0, 1)
sage: v.concatenate(w2).parent()
Vector space of dimension 8 over Rational Field
sage: v.concatenate(w2, ring=ZZ)
(1, 2, 3, 5, 0, 0, 0, 1)
sage: v.concatenate(w2, ring=ZZ).parent()
Ambient free module of rank 8 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> w2 = polygen(QQ)**Integer(4) + Integer(5)
```

(continues on next page)

(continued from previous page)

```
>>> v.concatenate(w2)
(1, 2, 3, 5, 0, 0, 0, 1)
>>> v.concatenate(w2).parent()
Vector space of dimension 8 over Rational Field
>>> v.concatenate(w2, ring=ZZ)
(1, 2, 3, 5, 0, 0, 0, 1)
>>> v.concatenate(w2, ring=ZZ).parent()
Ambient free module of rank 8 over the principal ideal domain Integer Ring
```

```
sage: v.concatenate(GF(9).gens())
(1, 2, 0, z2)
sage: v.concatenate(GF(9).gens()).parent()
Vector space of dimension 4 over Finite Field in z2 of size 3^2
```

```
>>> from sage.all import *
>>> v.concatenate(GF(Integer(9)).gens())
(1, 2, 0, z2)
>>> v.concatenate(GF(Integer(9)).gens()).parent()
Vector space of dimension 4 over Finite Field in z2 of size 3^2
```

conjugate()

Return a vector where every entry has been replaced by its complex conjugate.

OUTPUT:

A vector of the same length, over the same ring, but with each entry replaced by the complex conjugate, as implemented by the `conjugate()` method for elements of the base ring, which is presently always complex conjugation.

EXAMPLES:

```
sage: v = vector(CDF, [2.3 - 5.4*I, -1.7 + 3.6*I])                    #␣
↪needs sage.symbolic
sage: w = v.conjugate(); w                                             #␣
↪needs sage.symbolic
(2.3 + 5.4*I, -1.7 - 3.6*I)
sage: w.parent()                                                       #␣
↪needs sage.symbolic
Vector space of dimension 2 over Complex Double Field
```

```
>>> from sage.all import *
>>> v = vector(CDF, [RealNumber('2.3') - RealNumber('5.4')*I, -RealNumber('1.7
↪') + RealNumber('3.6')*I])                          # needs sage.symbolic
>>> w = v.conjugate(); w                                               #␣
↪needs sage.symbolic
(2.3 + 5.4*I, -1.7 - 3.6*I)
>>> w.parent()                                                         #␣
↪needs sage.symbolic
Vector space of dimension 2 over Complex Double Field
```

Even if conjugation seems nonsensical over a certain ring, this method for vectors cooperates silently.

```
sage: u = vector(ZZ, range(6))
sage: u.conjugate()
(0, 1, 2, 3, 4, 5)
```

```
>>> from sage.all import *
>>> u = vector(ZZ, range(Integer(6)))
>>> u.conjugate()
(0, 1, 2, 3, 4, 5)
```

Sage implements a few specialized subfields of the complex numbers, such as the cyclotomic fields. This example uses such a field containing a primitive 7-th root of unity named `a`.

```
sage: # needs sage.rings.number_field
sage: F.<a> = CyclotomicField(7)
sage: v = vector(F, [a^i for i in range(7)])
sage: v
(1, a, a^2, a^3, a^4, a^5, -a^5 - a^4 - a^3 - a^2 - a - 1)
sage: v.conjugate()
(1, -a^5 - a^4 - a^3 - a^2 - a - 1, a^5, a^4, a^3, a^2, a)
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> F = CyclotomicField(Integer(7), names=('a',)); (a,) = F._first_ngens(1)
>>> v = vector(F, [a**i for i in range(Integer(7))])
>>> v
(1, a, a^2, a^3, a^4, a^5, -a^5 - a^4 - a^3 - a^2 - a - 1)
>>> v.conjugate()
(1, -a^5 - a^4 - a^3 - a^2 - a - 1, a^5, a^4, a^3, a^2, a)
```

Sparse vectors are returned as such.

```
sage: # needs sage.symbolic
sage: v = vector(CC, {1: 5 - 6*I, 3: -7*I}); v
(0.000000000000000, 5.00000000000000 - 6.00000000000000*I, 0.000000000000000,␣
↪-7.00000000000000*I)
sage: v.is_sparse()
True
sage: vc = v.conjugate(); vc
(0.000000000000000, 5.00000000000000 + 6.00000000000000*I, 0.000000000000000,␣
↪7.00000000000000*I)
sage: vc.conjugate()
(0.000000000000000, 5.00000000000000 - 6.00000000000000*I, 0.000000000000000,␣
↪-7.00000000000000*I)
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> v = vector(CC, {Integer(1): Integer(5) - Integer(6)*I, Integer(3): -
↪Integer(7)*I}); v
(0.000000000000000, 5.00000000000000 - 6.00000000000000*I, 0.000000000000000,␣
↪-7.00000000000000*I)
>>> v.is_sparse()
True
```

(continues on next page)

(continued from previous page)

```
>>> vc = v.conjugate(); vc
(0.000000000000000, 5.00000000000000 + 6.00000000000000*I, 0.000000000000000,␣
↪7.00000000000000*I)
>>> vc.conjugate()
(0.000000000000000, 5.00000000000000 - 6.00000000000000*I, 0.000000000000000,␣
↪-7.00000000000000*I)
```

coordinate_ring()

Return the ring from which the coefficients of this vector come.

This is different from `base_ring()`, which returns the ring of scalars.

EXAMPLES:

```
sage: M = (ZZ^2) * (1/2)
sage: v = M([0,1/2])
sage: v.base_ring()
Integer Ring
sage: v.coordinate_ring()
Rational Field
```

```
>>> from sage.all import *
>>> M = (ZZ**Integer(2)) * (Integer(1)/Integer(2))
>>> v = M([Integer(0),Integer(1)/Integer(2)])
>>> v.base_ring()
Integer Ring
>>> v.coordinate_ring()
Rational Field
```

cross_product(*right*)

Return the cross product of `self` and `right`, which is only defined for vectors of length 3 or 7.

INPUT:

- `right` – a vector of the same size as `self`, either degree three or degree seven

OUTPUT:

The cross product (vector product) of `self` and `right`, a vector of the same size of `self` and `right`.

This product is performed under the assumption that the basis vectors are orthonormal. See the method `cross_product()` of vector fields for more general cases.

EXAMPLES:

```
sage: v = vector([1,2,3]); w = vector([0,5,-9])
sage: v.cross_product(v)
(0, 0, 0)
sage: u = v.cross_product(w); u
(-33, 9, 5)
sage: u.dot_product(v)
0
sage: u.dot_product(w)
0
```

```
>>> from sage.all import *
>>> v = vector([Integer(1),Integer(2),Integer(3)]); w = vector([Integer(0),
↪Integer(5),-Integer(9)])
>>> v.cross_product(v)
(0, 0, 0)
>>> u = v.cross_product(w); u
(-33, 9, 5)
>>> u.dot_product(v)
0
>>> u.dot_product(w)
0
```

The cross product is defined for degree seven vectors as well: see Wikipedia article Cross_product. The 3-D cross product is achieved using the quaternions, whereas the 7-D cross product is achieved using the octonions.

```
sage: u = vector(QQ, [1, -1/3, 57, -9, 56/4, -4,1])
sage: v = vector(QQ, [37, 55, -99/57, 9, -12, 11/3, 4/98])
sage: u.cross_product(v)
(1394815/2793, -2808401/2793, 39492/49, -48737/399, -9151880/2793, 62513/2793,
↪ -326603/171)
```

```
>>> from sage.all import *
>>> u = vector(QQ, [Integer(1), -Integer(1)/Integer(3), Integer(57), -
↪Integer(9), Integer(56)/Integer(4), -Integer(4),Integer(1)])
>>> v = vector(QQ, [Integer(37), Integer(55), -Integer(99)/Integer(57),␣
↪Integer(9), -Integer(12), Integer(11)/Integer(3), Integer(4)/Integer(98)])
>>> u.cross_product(v)
(1394815/2793, -2808401/2793, 39492/49, -48737/399, -9151880/2793, 62513/2793,
↪ -326603/171)
```

The degree seven cross product is anticommutative.

```
sage: u.cross_product(v) + v.cross_product(u)
(0, 0, 0, 0, 0, 0, 0)
```

```
>>> from sage.all import *
>>> u.cross_product(v) + v.cross_product(u)
(0, 0, 0, 0, 0, 0, 0)
```

The degree seven cross product is distributive across addition.

```
sage: v = vector([-12, -8/9, 42, 89, -37, 60/99, 73])
sage: u = vector([31, -42/7, 97, 80, 30/55, -32, 64])
sage: w = vector([-25/4, 40, -89, -91, -72/7, 79, 58])
sage: v.cross_product(u + w) - (v.cross_product(u) + v.cross_product(w))
(0, 0, 0, 0, 0, 0, 0)
```

```
>>> from sage.all import *
>>> v = vector([-Integer(12), -Integer(8)/Integer(9), Integer(42),␣
↪Integer(89), -Integer(37), Integer(60)/Integer(99), Integer(73)])
>>> u = vector([Integer(31), -Integer(42)/Integer(7), Integer(97),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(80), Integer(30)/Integer(55), -Integer(32), Integer(64)])
>>> w = vector([-Integer(25)/Integer(4), Integer(40), -Integer(89), -
↪Integer(91), -Integer(72)/Integer(7), Integer(79), Integer(58)])
>>> v.cross_product(u + w) - (v.cross_product(u) + v.cross_product(w))
(0, 0, 0, 0, 0, 0, 0)
```

The degree seven cross product respects scalar multiplication.

```
sage: v = vector([2, 17, -11/5, 21, -6, 2/17, 16])
sage: u = vector([-8, 9, -21, -6, -5/3, 12, 99])
sage: (5*v).cross_product(u) - 5*(v.cross_product(u))
(0, 0, 0, 0, 0, 0, 0)
sage: v.cross_product(5*u) - 5*(v.cross_product(u))
(0, 0, 0, 0, 0, 0, 0)
sage: (5*v).cross_product(u) - (v.cross_product(5*u))
(0, 0, 0, 0, 0, 0, 0)
```

```
>>> from sage.all import *
>>> v = vector([Integer(2), Integer(17), -Integer(11)/Integer(5), Integer(21),
↪ -Integer(6), Integer(2)/Integer(17), Integer(16)])
>>> u = vector([-Integer(8), Integer(9), -Integer(21), -Integer(6), -
↪Integer(5)/Integer(3), Integer(12), Integer(99)])
>>> (Integer(5)*v).cross_product(u) - Integer(5)*(v.cross_product(u))
(0, 0, 0, 0, 0, 0, 0)
>>> v.cross_product(Integer(5)*u) - Integer(5)*(v.cross_product(u))
(0, 0, 0, 0, 0, 0, 0)
>>> (Integer(5)*v).cross_product(u) - (v.cross_product(Integer(5)*u))
(0, 0, 0, 0, 0, 0, 0)
```

The degree seven cross product respects the scalar triple product.

```
sage: v = vector([2,6,-7/4,-9/12,-7,12,9])
sage: u = vector([22,-7,-9/11,12,15,15/7,11])
sage: w = vector([-11,17,19,-12/5,44,21/56,-8])
sage: v.dot_product(u.cross_product(w)) - w.dot_product(v.cross_product(u))
0
```

```
>>> from sage.all import *
>>> v = vector([Integer(2),Integer(6),-Integer(7)/Integer(4),-Integer(9)/
↪Integer(12),-Integer(7),Integer(12),Integer(9)])
>>> u = vector([Integer(22),-Integer(7),-Integer(9)/Integer(11),Integer(12),
↪Integer(15),Integer(15)/Integer(7),Integer(11)])
>>> w = vector([-Integer(11),Integer(17),Integer(19),-Integer(12)/Integer(5),
↪Integer(44),Integer(21)/Integer(56),-Integer(8)])
>>> v.dot_product(u.cross_product(w)) - w.dot_product(v.cross_product(u))
0
```

AUTHOR:

Billy Wonderly (2010-05-11), Added 7-D Cross Product

cross_product_matrix()

Return the matrix which describes a cross product between `self` and some other vector.

This operation is sometimes written using the hat operator: see Wikipedia article Hat_operator#Cross_product. It is only defined for vectors of length 3 or 7. For a vector v the cross product matrix $\hat{v}$ is a matrix which satisfies $\hat{v} \cdot w = v \times w$ and also $w \cdot \hat{v} = w \times v$ for all vectors w. The basis vectors are assumed to be orthonormal.

OUTPUT: the cross product matrix of this vector

EXAMPLES:

```
sage: v = vector([1, 2, 3])
sage: vh = v.cross_product_matrix()
sage: vh
[ 0 -3  2]
[ 3  0 -1]
[-2  1  0]
sage: w = random_vector(3, x=1, y=100)
sage: vh*w == v.cross_product(w)
True
sage: w*vh == w.cross_product(v)
True
sage: vh.is_alternating()
True
```

```
>>> from sage.all import *
>>> v = vector([Integer(1), Integer(2), Integer(3)])
>>> vh = v.cross_product_matrix()
>>> vh
[ 0 -3  2]
[ 3  0 -1]
[-2  1  0]
>>> w = random_vector(Integer(3), x=Integer(1), y=Integer(100))
>>> vh*w == v.cross_product(w)
True
>>> w*vh == w.cross_product(v)
True
>>> vh.is_alternating()
True
```

curl(*variables=None*)

Return the curl of this two-dimensional or three-dimensional vector function.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: vector([-y, x, 0]).curl()
(0, 0, 2)
sage: vector([y, -x, x*y*z]).curl()
(x*z, -y*z, -2)
sage: vector([y^2, 0, 0]).curl()
(0, 0, -2*y)
sage: (R^3).random_element().curl().div()
0
```

```
>>> from sage.all import *
>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> vector([-y, x, Integer(0)]).curl()
(0, 0, 2)
>>> vector([y, -x, x*y*z]).curl()
(x*z, -y*z, -2)
>>> vector([y**Integer(2), Integer(0), Integer(0)]).curl()
(0, 0, -2*y)
>>> (R**Integer(3)).random_element().curl().div()
0
```

For rings where the variable order is not well defined, it must be defined explicitly:

```
sage: v = vector(SR, [-y, x, 0])                                                #␣
↪needs sage.symbolic
sage: v.curl()                                                                  #␣
↪needs sage.symbolic
Traceback (most recent call last):
...
ValueError: Unable to determine ordered variable names for Symbolic Ring
sage: v.curl([x, y, z])                                                         #␣
↪needs sage.symbolic
(0, 0, 2)
```

```
>>> from sage.all import *
>>> v = vector(SR, [-y, x, Integer(0)])                                          ␣
↪     # needs sage.symbolic
>>> v.curl()                                                                   #␣
↪needs sage.symbolic
Traceback (most recent call last):
...
ValueError: Unable to determine ordered variable names for Symbolic Ring
>>> v.curl([x, y, z])                                                          #␣
↪needs sage.symbolic
(0, 0, 2)
```

Note that callable vectors have well defined variable orderings:

```
sage: v(x, y, z) = (-y, x, 0)                                                   #␣
↪needs sage.symbolic
sage: v.curl()                                                                  #␣
↪needs sage.symbolic
(x, y, z) |--> (0, 0, 2)
```

```
>>> from sage.all import *
>>> __tmp__=var("x,y,z"); v = symbolic_expression((-y, x, Integer(0))           ␣
↪                                        ).function(x,y,z)# needs sage.symbolic
>>> v.curl()                                                                   #␣
↪needs sage.symbolic
(x, y, z) |--> (0, 0, 2)
```

In two dimensions, this returns a scalar value:

```
sage: R.<x,y> = QQ[]
sage: vector([-y, x]).curl()
2
```

```
>>> from sage.all import *
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> vector([-y, x]).curl()
2
```

See also

`curl()` of vector fields on Euclidean spaces (and more generally pseudo-Riemannian manifolds), in particular for computing the curl in curvilinear coordinates.

degree()

Return the degree of this vector, which is simply the number of entries.

EXAMPLES:

```
sage: sage.modules.free_module_element.FreeModuleElement(QQ^389).degree()
389
sage: vector([1,2/3,8]).degree()
3
```

```
>>> from sage.all import *
>>> sage.modules.free_module_element.FreeModuleElement(QQ**Integer(389)).
↪degree()
389
>>> vector([Integer(1),Integer(2)/Integer(3),Integer(8)]).degree()
3
```

denominator()

Return the least common multiple of the denominators of the entries of `self`.

EXAMPLES:

```
sage: v = vector([1/2,2/5,3/14])
sage: v.denominator()
70
sage: 2*5*7
70
```

```
>>> from sage.all import *
>>> v = vector([Integer(1)/Integer(2),Integer(2)/Integer(5),Integer(3)/
↪Integer(14)])
>>> v.denominator()
70
>>> Integer(2)*Integer(5)*Integer(7)
70
```

```
sage: M = (ZZ^2)*(1/2)
sage: M.basis()[0].denominator()
2
```

```
>>> from sage.all import *
>>> M = (ZZ**Integer(2))*(Integer(1)/Integer(2))
>>> M.basis()[Integer(0)].denominator()
2
```

dense_vector()

Return dense version of `self`. If `self` is dense, just return `self`; otherwise, create and return correspond dense vector.

EXAMPLES:

```
sage: vector([-1,0,3,0,0,0]).dense_vector().is_dense()
True
sage: vector([-1,0,3,0,0,0],sparse=True).dense_vector().is_dense()
True
sage: vector([-1,0,3,0,0,0],sparse=True).dense_vector()
(-1, 0, 3, 0, 0, 0)
```

```
>>> from sage.all import *
>>> vector([-Integer(1),Integer(0),Integer(3),Integer(0),Integer(0),
↪Integer(0)]).dense_vector().is_dense()
True
>>> vector([-Integer(1),Integer(0),Integer(3),Integer(0),Integer(0),
↪Integer(0)],sparse=True).dense_vector().is_dense()
True
>>> vector([-Integer(1),Integer(0),Integer(3),Integer(0),Integer(0),
↪Integer(0)],sparse=True).dense_vector()
(-1, 0, 3, 0, 0, 0)
```

derivative(**args*)

Derivative with respect to variables supplied in args.

Multiple variables and iteration counts may be supplied; see documentation for the global derivative() function for more details.

diff() is an alias of this function.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: v = vector([1,x,x^2])
sage: v.derivative(x)
(0, 1, 2*x)
sage: type(v.derivative(x)) == type(v)
True
sage: v = vector([1,x,x^2], sparse=True)
sage: v.derivative(x)
(0, 1, 2*x)
sage: type(v.derivative(x)) == type(v)
True
```

(continues on next page)

(continued from previous page)

```
sage: v.derivative(x,x)
(0, 0, 2)
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> v = vector([Integer(1),x,x**Integer(2)])
>>> v.derivative(x)
(0, 1, 2*x)
>>> type(v.derivative(x)) == type(v)
True
>>> v = vector([Integer(1),x,x**Integer(2)], sparse=True)
>>> v.derivative(x)
(0, 1, 2*x)
>>> type(v.derivative(x)) == type(v)
True
>>> v.derivative(x,x)
(0, 0, 2)
```

dict(*copy=True*)

Return dictionary of nonzero entries of `self`.

More precisely, this returns 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`.

OUTPUT: Python dictionary

EXAMPLES:

```
sage: v = vector([0,0,0,0,1/2,0,3/14])
sage: v.dict()
{4: 1/2, 6: 3/14}
sage: sorted(v.support())
[4, 6]
```

```
>>> from sage.all import *
>>> v = vector([Integer(0),Integer(0),Integer(0),Integer(0),Integer(1)/
↪Integer(2),Integer(0),Integer(3)/Integer(14)])
>>> v.dict()
{4: 1/2, 6: 3/14}
>>> sorted(v.support())
[4, 6]
```

In some cases, when `copy=False`, we get back a dangerous reference:

```
sage: v = vector({0:5, 2:3/7}, sparse=True)
sage: v.dict(copy=False)
{0: 5, 2: 3/7}
sage: v.dict(copy=False)[0] = 18
```

(continues on next page)

(continued from previous page)

```
sage: v
(18, 0, 3/7)
```

```
>>> from sage.all import *
>>> v = vector({Integer(0):Integer(5), Integer(2):Integer(3)/Integer(7)},␣
↪sparse=True)
>>> v.dict(copy=False)
{0: 5, 2: 3/7}
>>> v.dict(copy=False)[Integer(0)] = Integer(18)
>>> v
(18, 0, 3/7)
```

diff(**args*)

Derivative with respect to variables supplied in args.

Multiple variables and iteration counts may be supplied; see documentation for the global derivative() function for more details.

diff() is an alias of this function.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: v = vector([1,x,x^2])
sage: v.derivative(x)
(0, 1, 2*x)
sage: type(v.derivative(x)) == type(v)
True
sage: v = vector([1,x,x^2], sparse=True)
sage: v.derivative(x)
(0, 1, 2*x)
sage: type(v.derivative(x)) == type(v)
True
sage: v.derivative(x,x)
(0, 0, 2)
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> v = vector([Integer(1),x,x**Integer(2)])
>>> v.derivative(x)
(0, 1, 2*x)
>>> type(v.derivative(x)) == type(v)
True
>>> v = vector([Integer(1),x,x**Integer(2)], sparse=True)
>>> v.derivative(x)
(0, 1, 2*x)
>>> type(v.derivative(x)) == type(v)
True
>>> v.derivative(x,x)
(0, 0, 2)
```

div(*variables=None*)

Return the divergence of this vector function.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: vector([x, y, z]).div()
3
sage: vector([x*y, y*z, z*x]).div()
x + y + z

sage: R.<x,y,z,w> = QQ[]
sage: vector([x*y, y*z, z*x]).div([x, y, z])
x + y + z
sage: vector([x*y, y*z, z*x]).div([z, x, y])
0
sage: vector([x*y, y*z, z*x]).div([x, y, w])
y + z

sage: vector(SR, [x*y, y*z, z*x]).div()                                    #␣
↪needs sage.symbolic
Traceback (most recent call last):
...
ValueError: Unable to determine ordered variable names for Symbolic Ring
sage: vector(SR, [x*y, y*z, z*x]).div([x, y, z])                           #␣
↪needs sage.symbolic
x + y + z
```

```
>>> from sage.all import *
>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> vector([x, y, z]).div()
3
>>> vector([x*y, y*z, z*x]).div()
x + y + z

>>> R = QQ['x, y, z, w']; (x, y, z, w,) = R._first_ngens(4)
>>> vector([x*y, y*z, z*x]).div([x, y, z])
x + y + z
>>> vector([x*y, y*z, z*x]).div([z, x, y])
0
>>> vector([x*y, y*z, z*x]).div([x, y, w])
y + z

>>> vector(SR, [x*y, y*z, z*x]).div()                                      #␣
↪needs sage.symbolic
Traceback (most recent call last):
...
ValueError: Unable to determine ordered variable names for Symbolic Ring
>>> vector(SR, [x*y, y*z, z*x]).div([x, y, z])                             #␣
↪needs sage.symbolic
x + y + z
```

See also

`divergence()` of vector fields on Euclidean spaces (and more generally pseudo-Riemannian manifolds),

in particular for computing the divergence in curvilinear coordinates.

dot_product(*right*)

Return the dot product of `self` and `right`, which is the sum of the product of the corresponding entries.

INPUT:

- `right` – a vector of the same degree as `self`. It does not need to belong to the same parent as `self`, so long as the necessary products and sums are defined.

OUTPUT:

If `self` and `right` are the vectors $\vec{x}$ and $\vec{y}$, of degree n, then this method returns

$$\sum_{i=1}^{n} x_i y_i$$

Note

The *inner_product()* is a more general version of this method, and the *hermitian_inner_product()* method may be more appropriate if your vectors have complex entries.

EXAMPLES:

```
sage: V = FreeModule(ZZ, 3)
sage: v = V([1,2,3])
sage: w = V([4,5,6])
sage: v.dot_product(w)
32
```

```
>>> from sage.all import *
>>> V = FreeModule(ZZ, Integer(3))
>>> v = V([Integer(1),Integer(2),Integer(3)])
>>> w = V([Integer(4),Integer(5),Integer(6)])
>>> v.dot_product(w)
32
```

```
sage: R.<x> = QQ[]
sage: v = vector([x,x^2,3*x]); w = vector([2*x,x,3+x])
sage: v*w
x^3 + 5*x^2 + 9*x
sage: (x*2*x) + (x^2*x) + (3*x*(3+x))
x^3 + 5*x^2 + 9*x
sage: w*v
x^3 + 5*x^2 + 9*x
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> v = vector([x,x**Integer(2),Integer(3)*x]); w = vector([Integer(2)*x,x,
↪Integer(3)+x])
>>> v*w
```

(continues on next page)

(continued from previous page)

```
x^3 + 5*x^2 + 9*x
>>> (x*Integer(2)*x) + (x**Integer(2)*x) + (Integer(3)*x*(Integer(3)+x))
x^3 + 5*x^2 + 9*x
>>> w*v
x^3 + 5*x^2 + 9*x
```

The vectors may be from different vector spaces, provided the necessary operations make sense. Notice that coercion will generate a result of the same type, even if the order of the arguments is reversed.:

```
sage: v = vector(ZZ, [1,2,3])
sage: w = vector(FiniteField(3), [0,1,2])
sage: ip = w.dot_product(v); ip
2
sage: ip.parent()
Finite Field of size 3

sage: ip = v.dot_product(w); ip
2
sage: ip.parent()
Finite Field of size 3
```

```
>>> from sage.all import *
>>> v = vector(ZZ, [Integer(1),Integer(2),Integer(3)])
>>> w = vector(FiniteField(Integer(3)), [Integer(0),Integer(1),Integer(2)])
>>> ip = w.dot_product(v); ip
2
>>> ip.parent()
Finite Field of size 3

>>> ip = v.dot_product(w); ip
2
>>> ip.parent()
Finite Field of size 3
```

The dot product of a vector with itself is the 2-norm, squared.

```
sage: v = vector(QQ, [3, 4, 7])
sage: v.dot_product(v) - v.norm()^2                                             #␣
↪needs sage.symbolic
0
```

```
>>> from sage.all import *
>>> v = vector(QQ, [Integer(3), Integer(4), Integer(7)])
>>> v.dot_product(v) - v.norm()**Integer(2)                                     ␣
↪      # needs sage.symbolic
0
```

element()

Simply return `self`. This is useful, since for many objects, `self.element()` returns a vector corresponding to `self`.

EXAMPLES:

```
sage: v = vector([1/2,2/5,0]); v
(1/2, 2/5, 0)
sage: v.element()
(1/2, 2/5, 0)
```

```
>>> from sage.all import *
>>> v = vector([Integer(1)/Integer(2),Integer(2)/Integer(5),Integer(0)]); v
(1/2, 2/5, 0)
>>> v.element()
(1/2, 2/5, 0)
```

get(*i*)

Like `__getitem__` but without bounds checking: i must satisfy `0 <= i < self.degree`.

EXAMPLES:

```
sage: vector(SR, [1/2,2/5,0]).get(0)                                    #␣
↪needs sage.symbolic
1/2
```

```
>>> from sage.all import *
>>> vector(SR, [Integer(1)/Integer(2),Integer(2)/Integer(5),Integer(0)]).
↪get(Integer(0))                                    # needs sage.symbolic
1/2
```

hamming_weight()

Return the number of positions `i` such that `self[i] != 0`.

EXAMPLES:

```
sage: vector([-1,0,3,0,0,0,0.01]).hamming_weight()
3
```

```
>>> from sage.all import *
>>> vector([-Integer(1),Integer(0),Integer(3),Integer(0),Integer(0),
↪Integer(0),RealNumber('0.01')]).hamming_weight()
3
```

hermitian_inner_product(*right*)

Return the dot product, but with the entries of the first vector conjugated beforehand.

INPUT:

- `right` – a vector of the same degree as `self`

OUTPUT:

If `self` and `right` are the vectors $\vec{x}$ and $\vec{y}$ of degree n then this routine computes

$$\sum_{i=1}^{n} \overline{x}_i y_i$$

where the bar indicates complex conjugation.

Note

If your vectors do not contain complex entries, then *dot_product()* will return the same result without the overhead of conjugating elements of `self`.

If you are not computing a weighted inner product, and your vectors do not have complex entries, then the *dot_product()* will return the same result.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: v = vector(CDF, [2+3*I, 5-4*I])
sage: w = vector(CDF, [6-4*I, 2+3*I])
sage: v.hermitian_inner_product(w)
-2.0 - 3.0*I
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> v = vector(CDF, [Integer(2)+Integer(3)*I, Integer(5)-Integer(4)*I])
>>> w = vector(CDF, [Integer(6)-Integer(4)*I, Integer(2)+Integer(3)*I])
>>> v.hermitian_inner_product(w)
-2.0 - 3.0*I
```

Sage implements a few specialized fields over the complex numbers, such as cyclotomic fields and quadratic number fields. So long as the base rings have a conjugate method, then the Hermitian inner product will be available.

```
sage: # needs sage.rings.number_field
sage: Q.<a> = QuadraticField(-7)
sage: a^2
-7
sage: v = vector(Q, [3+a, 5-2*a])
sage: w = vector(Q, [6, 4+3*a])
sage: v.hermitian_inner_product(w)
17*a - 4
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> Q = QuadraticField(-Integer(7), names=('a',)); (a,) = Q._first_ngens(1)
>>> a**Integer(2)
-7
>>> v = vector(Q, [Integer(3)+a, Integer(5)-Integer(2)*a])
>>> w = vector(Q, [Integer(6), Integer(4)+Integer(3)*a])
>>> v.hermitian_inner_product(w)
17*a - 4
```

The Hermitian inner product should be additive in each argument (we only need to test one), linear in each argument (with conjugation on the first scalar), and anti-commutative.

```
sage: # needs sage.rings.complex_double sage.symbolic
sage: alpha = CDF(5.0 + 3.0*I)
sage: u = vector(CDF, [2+4*I, -3+5*I, 2-7*I])
sage: v = vector(CDF, [-1+3*I, 5+4*I, 9-2*I])
```

(continues on next page)

(continued from previous page)

```
sage: w = vector(CDF, [8+3*I, -4+7*I, 3-6*I])
sage: (u+v).hermitian_inner_product(w) == u.hermitian_inner_product(w) + v.
↪hermitian_inner_product(w)
True
sage: (alpha*u).hermitian_inner_product(w) == alpha.conjugate()*u.hermitian_
↪inner_product(w)
True
sage: u.hermitian_inner_product(alpha*w) == alpha*u.hermitian_inner_product(w)
True
sage: u.hermitian_inner_product(v) == v.hermitian_inner_product(u).conjugate()
True
```

```
>>> from sage.all import *
>>> # needs sage.rings.complex_double sage.symbolic
>>> alpha = CDF(RealNumber('5.0') + RealNumber('3.0')*I)
>>> u = vector(CDF, [Integer(2)+Integer(4)*I, -Integer(3)+Integer(5)*I,␣
↪Integer(2)-Integer(7)*I])
>>> v = vector(CDF, [-Integer(1)+Integer(3)*I, Integer(5)+Integer(4)*I,␣
↪Integer(9)-Integer(2)*I])
>>> w = vector(CDF, [Integer(8)+Integer(3)*I, -Integer(4)+Integer(7)*I,␣
↪Integer(3)-Integer(6)*I])
>>> (u+v).hermitian_inner_product(w) == u.hermitian_inner_product(w) + v.
↪hermitian_inner_product(w)
True
>>> (alpha*u).hermitian_inner_product(w) == alpha.conjugate()*u.hermitian_
↪inner_product(w)
True
>>> u.hermitian_inner_product(alpha*w) == alpha*u.hermitian_inner_product(w)
True
>>> u.hermitian_inner_product(v) == v.hermitian_inner_product(u).conjugate()
True
```

For vectors with complex entries, the Hermitian inner product has a more natural relationship with the 2-norm (which is the default for the *norm()* method). The norm squared equals the Hermitian inner product of the vector with itself.

```
sage: # needs sage.rings.complex_double sage.symbolic
sage: v = vector(CDF, [-0.66+0.47*I, -0.60+0.91*I, -0.62-0.87*I, 0.53+0.32*I])
sage: abs(v.norm()^2 - v.hermitian_inner_product(v)) < 1.0e-10
True
```

```
>>> from sage.all import *
>>> # needs sage.rings.complex_double sage.symbolic
>>> v = vector(CDF, [-RealNumber('0.66')+RealNumber('0.47')*I, -RealNumber('0.
↪60')+RealNumber('0.91')*I, -RealNumber('0.62')-RealNumber('0.87')*I,␣
↪RealNumber('0.53')+RealNumber('0.32')*I])
>>> abs(v.norm()**Integer(2) - v.hermitian_inner_product(v)) < RealNumber('1.
↪0e-10')
True
```

inner_product(*right*)

Return the inner product of `self` and `right`, possibly using an inner product matrix from the parent of

`self`.

INPUT:

- `right` – a vector of the same degree as `self`

OUTPUT:

If the parent vector space does not have an inner product matrix defined, then this is the usual dot product (*dot_product()*). If `self` and `right` are considered as single column matrices, $\vec{x}$ and $\vec{y}$, and A is the inner product matrix, then this method computes

$$(\vec{x})^t A\vec{y}$$

where t indicates the transpose.

Note

If your vectors have complex entries, the *hermitian_inner_product()* may be more appropriate for your purposes.

EXAMPLES:

```
sage: v = vector(QQ, [1,2,3])
sage: w = vector(QQ, [-1,2,-3])
sage: v.inner_product(w)
-6
sage: v.inner_product(w) == v.dot_product(w)
True
```

```
>>> from sage.all import *
>>> v = vector(QQ, [Integer(1),Integer(2),Integer(3)])
>>> w = vector(QQ, [-Integer(1),Integer(2),-Integer(3)])
>>> v.inner_product(w)
-6
>>> v.inner_product(w) == v.dot_product(w)
True
```

The vector space or free module that is the parent to `self` can have an inner product matrix defined, which will be used by this method. This matrix will be passed through to subspaces.

```
sage: ipm = matrix(ZZ,[[2,0,-1], [0,2,0], [-1,0,6]])
sage: M = FreeModule(ZZ, 3, inner_product_matrix=ipm)
sage: v = M([1,0,0])
sage: v.inner_product(v)
2
sage: K = M.span_of_basis([[0/2,-1/2,-1/2], [0,1/2,-1/2], [2,0,0]])
sage: (K.0).inner_product(K.0)
2
sage: w = M([1,3,-1])
sage: v = M([2,-4,5])
sage: w.row()*ipm*v.column() == w.inner_product(v)
True
```

```
>>> from sage.all import *
>>> ipm = matrix(ZZ,[[Integer(2),Integer(0),-Integer(1)], [Integer(0),
↪Integer(2),Integer(0)], [-Integer(1),Integer(0),Integer(6)]])
>>> M = FreeModule(ZZ, Integer(3), inner_product_matrix=ipm)
>>> v = M([Integer(1),Integer(0),Integer(0)])
>>> v.inner_product(v)
2
>>> K = M.span_of_basis([[Integer(0)/Integer(2),-Integer(1)/Integer(2),-
↪Integer(1)/Integer(2)], [Integer(0),Integer(1)/Integer(2),-Integer(1)/
↪Integer(2)], [Integer(2),Integer(0),Integer(0)]])
>>> (K.gen(0)).inner_product(K.gen(0))
2
>>> w = M([Integer(1),Integer(3),-Integer(1)])
>>> v = M([Integer(2),-Integer(4),Integer(5)])
>>> w.row()*ipm*v.column() == w.inner_product(v)
True
```

Note that the inner product matrix comes from the parent of `self`. So if a vector is not an element of the correct parent, the result could be a source of confusion.

```
sage: V = VectorSpace(QQ, 2, inner_product_matrix=[[1,2],[2,1]])
sage: v = V([12, -10])
sage: w = vector(QQ, [10,12])
sage: v.inner_product(w)
88
sage: w.inner_product(v)
0
sage: w = V(w)
sage: w.inner_product(v)
88
```

```
>>> from sage.all import *
>>> V = VectorSpace(QQ, Integer(2), inner_product_matrix=[[Integer(1),
↪Integer(2)],[Integer(2),Integer(1)]])
>>> v = V([Integer(12), -Integer(10)])
>>> w = vector(QQ, [Integer(10),Integer(12)])
>>> v.inner_product(w)
88
>>> w.inner_product(v)
0
>>> w = V(w)
>>> w.inner_product(v)
88
```

Note

The use of an inner product matrix makes no restrictions on the nature of the matrix. In particular, in this context it need not be Hermitian and positive-definite (as it is in the example above).

integral(**args*, ***kwds*)

Return a symbolic integral of the vector, component-wise.

integrate() is an alias of the function.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: t = var('t')
sage: r = vector([t,t^2,sin(t)])
sage: r.integral(t)
(1/2*t^2, 1/3*t^3, -cos(t))
sage: integrate(r, t)
(1/2*t^2, 1/3*t^3, -cos(t))
sage: r.integrate(t, 0, 1)
(1/2, 1/3, -cos(1) + 1)
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> t = var('t')
>>> r = vector([t,t**Integer(2),sin(t)])
>>> r.integral(t)
(1/2*t^2, 1/3*t^3, -cos(t))
>>> integrate(r, t)
(1/2*t^2, 1/3*t^3, -cos(t))
>>> r.integrate(t, Integer(0), Integer(1))
(1/2, 1/3, -cos(1) + 1)
```

integrate(**args*, ***kwds*)

Return a symbolic integral of the vector, component-wise.

integrate() is an alias of the function.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: t = var('t')
sage: r = vector([t,t^2,sin(t)])
sage: r.integral(t)
(1/2*t^2, 1/3*t^3, -cos(t))
sage: integrate(r, t)
(1/2*t^2, 1/3*t^3, -cos(t))
sage: r.integrate(t, 0, 1)
(1/2, 1/3, -cos(1) + 1)
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> t = var('t')
>>> r = vector([t,t**Integer(2),sin(t)])
>>> r.integral(t)
(1/2*t^2, 1/3*t^3, -cos(t))
>>> integrate(r, t)
(1/2*t^2, 1/3*t^3, -cos(t))
>>> r.integrate(t, Integer(0), Integer(1))
(1/2, 1/3, -cos(1) + 1)
```

is_dense()

Return `True` if this is a dense vector, which is just a statement about the data structure, not the number of

nonzero entries.

EXAMPLES:

```
sage: vector([1/2, 2/5, 0]).is_dense()
True
sage: vector([1/2, 2/5, 0], sparse=True).is_dense()
False
```

```
>>> from sage.all import *
>>> vector([Integer(1)/Integer(2), Integer(2)/Integer(5), Integer(0)]).is_
↪dense()
True
>>> vector([Integer(1)/Integer(2), Integer(2)/Integer(5), Integer(0)],␣
↪sparse=True).is_dense()
False
```

is_sparse()

Return `True` if this is a sparse vector, which is just a statement about the data structure, not the number of nonzero entries.

EXAMPLES:

```
sage: vector([1/2, 2/5, 0]).is_sparse()
False
sage: vector([1/2, 2/5, 0], sparse=True).is_sparse()
True
```

```
>>> from sage.all import *
>>> vector([Integer(1)/Integer(2), Integer(2)/Integer(5), Integer(0)]).is_
↪sparse()
False
>>> vector([Integer(1)/Integer(2), Integer(2)/Integer(5), Integer(0)],␣
↪sparse=True).is_sparse()
True
```

is_vector()

Return `True`, since this is a vector.

EXAMPLES:

```
sage: vector([1/2, 2/5, 0]).is_vector()
True
```

```
>>> from sage.all import *
>>> vector([Integer(1)/Integer(2), Integer(2)/Integer(5), Integer(0)]).is_
↪vector()
True
```

items()

Return an iterator over `self`.

EXAMPLES:

```
sage: v = vector([1,2/3,pi])                                                #␣
↪needs sage.symbolic
sage: v.items()                                                             #␣
↪needs sage.symbolic
<...generator object at ...>
sage: list(v.items())                                                       #␣
↪needs sage.symbolic
[(0, 1), (1, 2/3), (2, pi)]
```

```
>>> from sage.all import *
>>> v = vector([Integer(1),Integer(2)/Integer(3),pi])                       ␣
↪                        # needs sage.symbolic
>>> v.items()                                                              #␣
↪needs sage.symbolic
<...generator object at ...>
>>> list(v.items())                                                        #␣
↪needs sage.symbolic
[(0, 1), (1, 2/3), (2, pi)]
```

iteritems()

Return an iterator over `self`.

EXAMPLES:

```
sage: v = vector([1,2/3,pi])                                                #␣
↪needs sage.symbolic
sage: v.items()                                                             #␣
↪needs sage.symbolic
<...generator object at ...>
sage: list(v.items())                                                       #␣
↪needs sage.symbolic
[(0, 1), (1, 2/3), (2, pi)]
```

```
>>> from sage.all import *
>>> v = vector([Integer(1),Integer(2)/Integer(3),pi])                       ␣
↪                        # needs sage.symbolic
>>> v.items()                                                              #␣
↪needs sage.symbolic
<...generator object at ...>
>>> list(v.items())                                                        #␣
↪needs sage.symbolic
[(0, 1), (1, 2/3), (2, pi)]
```

lift()

Lift `self` to the cover ring.

OUTPUT:

Return a lift of `self` to the covering ring of the base ring R, which is by definition the ring returned by calling `cover_ring()` on R, or just R itself if the `cover_ring()` method is not defined.

EXAMPLES:

```
sage: V = vector(Integers(7), [5, 9, 13, 15]) ; V
(5, 2, 6, 1)
sage: V.lift()
(5, 2, 6, 1)
sage: parent(V.lift())
Ambient free module of rank 4 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> V = vector(Integers(Integer(7)), [Integer(5), Integer(9), Integer(13),␣
↪Integer(15)]) ; V
(5, 2, 6, 1)
>>> V.lift()
(5, 2, 6, 1)
>>> parent(V.lift())
Ambient free module of rank 4 over the principal ideal domain Integer Ring
```

If the base ring does not have a cover method, return a copy of the vector:

```
sage: W = vector(QQ, [1, 2, 3])
sage: W1 = W.lift()
sage: W is W1
False
sage: parent(W1)
Vector space of dimension 3 over Rational Field
```

```
>>> from sage.all import *
>>> W = vector(QQ, [Integer(1), Integer(2), Integer(3)])
>>> W1 = W.lift()
>>> W is W1
False
>>> parent(W1)
Vector space of dimension 3 over Rational Field
```

lift_centered()

Lift to a congruent, centered vector.

INPUT:

- `self` A vector with coefficients in $Integers(n)$.

OUTPUT:

- The unique integer vector v such that foreach i, $Mod(v[i], n) = Mod(self[i], n)$ and $-n/2 < v[i] \leq n/2$.

EXAMPLES:

```
sage: V = vector(Integers(7), [5, 9, 13, 15]) ; V
(5, 2, 6, 1)
sage: V.lift_centered()
(-2, 2, -1, 1)
sage: parent(V.lift_centered())
Ambient free module of rank 4 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> V = vector(Integers(Integer(7)), [Integer(5), Integer(9), Integer(13),␣
↪Integer(15)]) ; V
(5, 2, 6, 1)
>>> V.lift_centered()
(-2, 2, -1, 1)
>>> parent(V.lift_centered())
Ambient free module of rank 4 over the principal ideal domain Integer Ring
```

list(*copy=True*)

Return list of elements of `self`.

INPUT:

- `copy` – boolean (default: `True`); whether returned list is a copy that is safe to change (ignored)

EXAMPLES:

```
sage: P.<x,y,z> = QQ[]
sage: v = vector([x,y,z], sparse=True)
sage: type(v)
<class 'sage.modules.free_module_element.FreeModuleElement_generic_sparse'>
sage: a = v.list(); a
[x, y, z]
sage: a[0] = x*y; v
(x, y, z)
```

```
>>> from sage.all import *
>>> P = QQ['x, y, z']; (x, y, z,) = P._first_ngens(3)
>>> v = vector([x,y,z], sparse=True)
>>> type(v)
<class 'sage.modules.free_module_element.FreeModuleElement_generic_sparse'>
>>> a = v.list(); a
[x, y, z]
>>> a[Integer(0)] = x*y; v
(x, y, z)
```

The optional argument `copy` is ignored:

```
sage: a = v.list(copy=False); a
[x, y, z]
sage: a[0] = x*y; v
(x, y, z)
```

```
>>> from sage.all import *
>>> a = v.list(copy=False); a
[x, y, z]
>>> a[Integer(0)] = x*y; v
(x, y, z)
```

list_from_positions(*positions*)

Return list of elements chosen from this vector using the given positions of this vector.

INPUT:

- positions – iterable of integers

EXAMPLES:

```
sage: v = vector([1, 2/3, pi])                                                  #␣
↪needs sage.symbolic
sage: v.list_from_positions([0,0,0,2,1])                                        #␣
↪needs sage.symbolic
[1, 1, 1, pi, 2/3]
```

```
>>> from sage.all import *
>>> v = vector([Integer(1), Integer(2)/Integer(3), pi])                         ␣
↪                        # needs sage.symbolic
>>> v.list_from_positions([Integer(0),Integer(0),Integer(0),Integer(2),
↪Integer(1)])                                        # needs sage.symbolic
[1, 1, 1, pi, 2/3]
```

monic()

Return this vector divided through by the first nonzero entry of this vector.

EXAMPLES:

```
sage: v = vector(QQ, [0, 4/3, 5, 1, 2])
sage: v.monic()
(0, 1, 15/4, 3/4, 3/2)
sage: v = vector(QQ, [])
sage: v.monic()
()
```

```
>>> from sage.all import *
>>> v = vector(QQ, [Integer(0), Integer(4)/Integer(3), Integer(5), Integer(1),
↪ Integer(2)])
>>> v.monic()
(0, 1, 15/4, 3/4, 3/2)
>>> v = vector(QQ, [])
>>> v.monic()
()
```

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: V = ZZ^3
sage: v = V([1, 0, 5])
sage: v.monomial_coefficients()
{0: 1, 2: 5}
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(3)
>>> v = V([Integer(1), Integer(0), Integer(5)])
>>> v.monomial_coefficients()
{0: 1, 2: 5}
```

Check that it works for submodules (Issue #34455):

```
sage: V = ZZ^3
sage: U = V.submodule([[1, 2, 3], [1, 1, 1]])
sage: U
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1  0 -1]
[ 0  1  2]
sage: u = U([2, 3, 4])
sage: u.monomial_coefficients()
{0: 2, 1: 3}
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(3)
>>> U = V.submodule([[Integer(1), Integer(2), Integer(3)], [Integer(1),␣
↪Integer(1), Integer(1)]])
>>> U
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1  0 -1]
[ 0  1  2]
>>> u = U([Integer(2), Integer(3), Integer(4)])
>>> u.monomial_coefficients()
{0: 2, 1: 3}
```

nintegral(**args*, ***kwds*)

Return a numeric integral of the vector, component-wise, and the result of the nintegral command on each component of the input.

nintegrate() is an alias of the function.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: t = var('t')
sage: r = vector([t,t^2,sin(t)])
sage: vec, answers = r.nintegral(t,0,1)
sage: vec  # abs tol 1e-15
(0.5, 0.3333333333333334, 0.4596976941318602)
sage: type(vec)
<class 'sage.modules.vector_real_double_dense.Vector_real_double_dense'>
sage: answers
[(0.5, 5.55111512312578...e-15, 21, 0),
 (0.3333333333333..., 3.70074341541719...e-15, 21, 0),
 (0.45969769413186..., 5.10366964392284...e-15, 21, 0)]

sage: # needs sage.symbolic
```

(continues on next page)

(continued from previous page)

```
sage: r = vector([t,0,1], sparse=True)
sage: r.nintegral(t, 0, 1)
((0.5, 0.0, 1.0),
 {0: (0.5, 5.55111512312578...e-15, 21, 0),
  2: (1.0, 1.11022302462515...e-14, 21, 0)})
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> t = var('t')
>>> r = vector([t,t**Integer(2),sin(t)])
>>> vec, answers = r.nintegral(t,Integer(0),Integer(1))
>>> vec  # abs tol 1e-15
(0.5, 0.3333333333333334, 0.4596976941318602)
>>> type(vec)
<class 'sage.modules.vector_real_double_dense.Vector_real_double_dense'>
>>> answers
[(0.5, 5.55111512312578...e-15, 21, 0),
 (0.3333333333333..., 3.70074341541719...e-15, 21, 0),
 (0.45969769413186..., 5.10366964392284...e-15, 21, 0)]

>>> # needs sage.symbolic
>>> r = vector([t,Integer(0),Integer(1)], sparse=True)
>>> r.nintegral(t, Integer(0), Integer(1))
((0.5, 0.0, 1.0),
 {0: (0.5, 5.55111512312578...e-15, 21, 0),
  2: (1.0, 1.11022302462515...e-14, 21, 0)})
```

nintegrate(**args*, ***kwds*)

Return a numeric integral of the vector, component-wise, and the result of the nintegral command on each component of the input.

nintegrate() is an alias of the function.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: t = var('t')
sage: r = vector([t,t^2,sin(t)])
sage: vec, answers = r.nintegral(t,0,1)
sage: vec  # abs tol 1e-15
(0.5, 0.3333333333333334, 0.4596976941318602)
sage: type(vec)
<class 'sage.modules.vector_real_double_dense.Vector_real_double_dense'>
sage: answers
[(0.5, 5.55111512312578...e-15, 21, 0),
 (0.3333333333333..., 3.70074341541719...e-15, 21, 0),
 (0.45969769413186..., 5.10366964392284...e-15, 21, 0)]

sage: # needs sage.symbolic
sage: r = vector([t,0,1], sparse=True)
sage: r.nintegral(t, 0, 1)
((0.5, 0.0, 1.0),
 {0: (0.5, 5.55111512312578...e-15, 21, 0),
```

(continues on next page)

(continued from previous page)

```
  2: (1.0, 1.11022302462515...e-14, 21, 0)})
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> t = var('t')
>>> r = vector([t,t**Integer(2),sin(t)])
>>> vec, answers = r.nintegral(t,Integer(0),Integer(1))
>>> vec  # abs tol 1e-15
(0.5, 0.3333333333333334, 0.4596976941318602)
>>> type(vec)
<class 'sage.modules.vector_real_double_dense.Vector_real_double_dense'>
>>> answers
[(0.5, 5.55111512312578...e-15, 21, 0),
 (0.3333333333333..., 3.70074341541719...e-15, 21, 0),
 (0.45969769413186..., 5.10366964392284...e-15, 21, 0)]

>>> # needs sage.symbolic
>>> r = vector([t,Integer(0),Integer(1)], sparse=True)
>>> r.nintegral(t, Integer(0), Integer(1))
((0.5, 0.0, 1.0),
 {0: (0.5, 5.55111512312578...e-15, 21, 0),
  2: (1.0, 1.11022302462515...e-14, 21, 0)})
```

nonzero_positions()

Return the sorted list of integers `i` such that `self[i] != 0`.

EXAMPLES:

```
sage: vector([-1,0,3,0,0,0,0.01]).nonzero_positions()
[0, 2, 6]
```

```
>>> from sage.all import *
>>> vector([-Integer(1),Integer(0),Integer(3),Integer(0),Integer(0),
↪Integer(0),RealNumber('0.01')]).nonzero_positions()
[0, 2, 6]
```

norm(*p='__two__'*)

Return the p-norm of `self`.

INPUT:

- `p` – (default: 2) `p` can be a real number greater than 1, infinity (`oo` or `Infinity`), or a symbolic expression:
  - $p = 1$: the taxicab (Manhattan) norm
  - $p = 2$: the usual Euclidean norm (the default)
  - $p = \infty$: the maximum entry (in absolute value)

Note

See also `sage.misc.functional.norm()`

EXAMPLES:

```
sage: v = vector([1,2,-3])
sage: v.norm(5)                                                             #␣
↪needs sage.symbolic
276^(1/5)
```

```
>>> from sage.all import *
>>> v = vector([Integer(1),Integer(2),-Integer(3)])
>>> v.norm(Integer(5))                                                      ␣
↪     # needs sage.symbolic
276^(1/5)
```

The default is the usual Euclidean norm.

```
sage: v.norm()                                                              #␣
↪needs sage.symbolic
sqrt(14)
sage: v.norm(2)                                                             #␣
↪needs sage.symbolic
sqrt(14)
```

```
>>> from sage.all import *
>>> v.norm()                                                               #␣
↪needs sage.symbolic
sqrt(14)
>>> v.norm(Integer(2))                                                      ␣
↪     # needs sage.symbolic
sqrt(14)
```

The infinity norm is the maximum size (in absolute value) of the entries.

```
sage: v.norm(Infinity)
3
sage: v.norm(oo)
3
```

```
>>> from sage.all import *
>>> v.norm(Infinity)
3
>>> v.norm(oo)
3
```

Real or symbolic values may be used for `p`.

```
sage: v=vector(RDF,[1,2,3])
sage: v.norm(5)
3.077384885394063

sage: # needs sage.symbolic
sage: v.norm(pi/2)    # abs tol 1e-15
4.216595864704748
sage: _ = var('a b c d p'); v = vector([a, b, c, d])
```

(continues on next page)

(continued from previous page)

```
sage: v.norm(p)
(abs(a)^p + abs(b)^p + abs(c)^p + abs(d)^p)^(1/p)
```

```
>>> from sage.all import *
>>> v=vector(RDF,[Integer(1),Integer(2),Integer(3)])
>>> v.norm(Integer(5))
3.077384885394063

>>> # needs sage.symbolic
>>> v.norm(pi/Integer(2))     # abs tol 1e-15
4.216595864704748
>>> _ = var('a b c d p'); v = vector([a, b, c, d])
>>> v.norm(p)
(abs(a)^p + abs(b)^p + abs(c)^p + abs(d)^p)^(1/p)
```

Notice that the result may be a symbolic expression, owing to the necessity of taking a square root (in the default case). These results can be converted to numerical values if needed.

```
sage: v = vector(ZZ, [3,4])
sage: nrm = v.norm(); nrm
5
sage: nrm.parent()
Rational Field

sage: # needs sage.symbolic
sage: v = vector(QQ, [3, 5])
sage: nrm = v.norm(); nrm
sqrt(34)
sage: nrm.parent()
Symbolic Ring
sage: numeric = N(nrm); numeric
5.83095189484...
sage: numeric.parent()
Real Field with 53 bits of precision
```

```
>>> from sage.all import *
>>> v = vector(ZZ, [Integer(3),Integer(4)])
>>> nrm = v.norm(); nrm
5
>>> nrm.parent()
Rational Field

>>> # needs sage.symbolic
>>> v = vector(QQ, [Integer(3), Integer(5)])
>>> nrm = v.norm(); nrm
sqrt(34)
>>> nrm.parent()
Symbolic Ring
>>> numeric = N(nrm); numeric
5.83095189484...
>>> numeric.parent()
Real Field with 53 bits of precision
```

normalized(*p='__two__'*)

Return the input vector divided by the p-norm.

INPUT:

- `p` – (default: 2) p value for the norm

EXAMPLES:

```
sage: v = vector(QQ, [4, 1, 3, 2])
sage: v.normalized()                                                    #␣
↪needs sage.symbolic
(2/15*sqrt(30), 1/30*sqrt(30), 1/10*sqrt(30), 1/15*sqrt(30))
sage: sum(v.normalized(1))
1
```

```
>>> from sage.all import *
>>> v = vector(QQ, [Integer(4), Integer(1), Integer(3), Integer(2)])
>>> v.normalized()                                                      #␣
↪needs sage.symbolic
(2/15*sqrt(30), 1/30*sqrt(30), 1/10*sqrt(30), 1/15*sqrt(30))
>>> sum(v.normalized(Integer(1)))
1
```

Note that normalizing the vector may change the base ring:

```
sage: v.base_ring() == v.normalized().base_ring()                       #␣
↪needs sage.symbolic
False
sage: u = vector(RDF, [-3, 4, 6, 9])
sage: u.base_ring() == u.normalized().base_ring()
True
```

```
>>> from sage.all import *
>>> v.base_ring() == v.normalized().base_ring()                         #␣
↪needs sage.symbolic
False
>>> u = vector(RDF, [-Integer(3), Integer(4), Integer(6), Integer(9)])
>>> u.base_ring() == u.normalized().base_ring()
True
```

numerical_approx(*prec=None*, *digits=None*, *algorithm=None*)

Return a numerical approximation of `self` with `prec` bits (or decimal `digits`) of precision, by approximating all entries.

INPUT:

- `prec` – precision in bits
- `digits` – precision in decimal digits (only used if `prec` is not given)
- `algorithm` – which algorithm to use to compute the approximation of the entries (the accepted algorithms depend on the object)

If neither `prec` nor `digits` is given, the default precision is 53 bits (roughly 16 digits).

EXAMPLES:

```
sage: v = vector(RealField(212), [1,2,3])
sage: v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)
sage: _.parent()
Vector space of dimension 3 over Real Field with 53 bits of precision
sage: numerical_approx(v)
(1.00000000000000, 2.00000000000000, 3.00000000000000)
sage: _.parent()
Vector space of dimension 3 over Real Field with 53 bits of precision
sage: v.n(prec=75)
(1.0000000000000000000, 2.0000000000000000000, 3.0000000000000000000)
sage: _.parent()
Vector space of dimension 3 over Real Field with 75 bits of precision
sage: numerical_approx(v, digits=3)
(1.00, 2.00, 3.00)
sage: _.parent()
Vector space of dimension 3 over Real Field with 14 bits of precision
```

```
>>> from sage.all import *
>>> v = vector(RealField(Integer(212)), [Integer(1),Integer(2),Integer(3)])
>>> v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)
>>> _.parent()
Vector space of dimension 3 over Real Field with 53 bits of precision
>>> numerical_approx(v)
(1.00000000000000, 2.00000000000000, 3.00000000000000)
>>> _.parent()
Vector space of dimension 3 over Real Field with 53 bits of precision
>>> v.n(prec=Integer(75))
(1.0000000000000000000, 2.0000000000000000000, 3.0000000000000000000)
>>> _.parent()
Vector space of dimension 3 over Real Field with 75 bits of precision
>>> numerical_approx(v, digits=Integer(3))
(1.00, 2.00, 3.00)
>>> _.parent()
Vector space of dimension 3 over Real Field with 14 bits of precision
```

Both functional and object-oriented usage is possible.

```
sage: u = vector(QQ, [1/2, 1/3, 1/4])
sage: u.n()
(0.500000000000000, 0.333333333333333, 0.250000000000000)
sage: u.numerical_approx()
(0.500000000000000, 0.333333333333333, 0.250000000000000)
sage: n(u)
(0.500000000000000, 0.333333333333333, 0.250000000000000)
sage: N(u)
(0.500000000000000, 0.333333333333333, 0.250000000000000)
sage: numerical_approx(u)
(0.500000000000000, 0.333333333333333, 0.250000000000000)
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> u = vector(QQ, [Integer(1)/Integer(2), Integer(1)/Integer(3), Integer(1)/
↪Integer(4)])
>>> u.n()
(0.500000000000000, 0.333333333333333, 0.250000000000000)
>>> u.numerical_approx()
(0.500000000000000, 0.333333333333333, 0.250000000000000)
>>> n(u)
(0.500000000000000, 0.333333333333333, 0.250000000000000)
>>> N(u)
(0.500000000000000, 0.333333333333333, 0.250000000000000)
>>> numerical_approx(u)
(0.500000000000000, 0.333333333333333, 0.250000000000000)
```

Precision (bits) and digits (decimal) may be specified. When both are given, `prec` wins.

```
sage: u = vector(QQ, [1/2, 1/3, 1/4])
sage: n(u, prec=15)
(0.5000, 0.3333, 0.2500)
sage: n(u, digits=5)
(0.50000, 0.33333, 0.25000)
sage: n(u, prec=30, digits=100)
(0.50000000, 0.33333333, 0.25000000)
```

```
>>> from sage.all import *
>>> u = vector(QQ, [Integer(1)/Integer(2), Integer(1)/Integer(3), Integer(1)/
↪Integer(4)])
>>> n(u, prec=Integer(15))
(0.5000, 0.3333, 0.2500)
>>> n(u, digits=Integer(5))
(0.50000, 0.33333, 0.25000)
>>> n(u, prec=Integer(30), digits=Integer(100))
(0.50000000, 0.33333333, 0.25000000)
```

These are some legacy doctests that were part of various specialized versions of the numerical approximation routine that were removed as part of Issue #12195.

```
sage: v = vector(ZZ, [1,2,3])
sage: v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)
sage: _.parent()
Vector space of dimension 3 over Real Field with 53 bits of precision
sage: v.n(prec=75)
(1.0000000000000000000, 2.0000000000000000000, 3.0000000000000000000)
sage: _.parent()
Vector space of dimension 3 over Real Field with 75 bits of precision

sage: v = vector(RDF, [1,2,3])
sage: v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)
sage: _.parent()
Vector space of dimension 3 over Real Field with 53 bits of precision
sage: v = vector(CDF, [1,2,3])
```

(continues on next page)

(continued from previous page)

```
sage: v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)
sage: _.parent()
Vector space of dimension 3 over Complex Field with 53 bits of precision

sage: v = vector(Integers(8), [1,2,3])
sage: v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)
sage: _.parent()
Vector space of dimension 3 over Real Field with 53 bits of precision
sage: v.n(prec=75)
(1.0000000000000000000, 2.0000000000000000000, 3.0000000000000000000)
sage: _.parent()
Vector space of dimension 3 over Real Field with 75 bits of precision

sage: v = vector(QQ, [1,2,3])
sage: v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)
sage: _.parent()
Vector space of dimension 3 over Real Field with 53 bits of precision
sage: v.n(prec=75)
(1.0000000000000000000, 2.0000000000000000000, 3.0000000000000000000)
sage: _.parent()
Vector space of dimension 3 over Real Field with 75 bits of precision
```

```
>>> from sage.all import *
>>> v = vector(ZZ, [Integer(1),Integer(2),Integer(3)])
>>> v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)
>>> _.parent()
Vector space of dimension 3 over Real Field with 53 bits of precision
>>> v.n(prec=Integer(75))
(1.0000000000000000000, 2.0000000000000000000, 3.0000000000000000000)
>>> _.parent()
Vector space of dimension 3 over Real Field with 75 bits of precision

>>> v = vector(RDF, [Integer(1),Integer(2),Integer(3)])
>>> v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)
>>> _.parent()
Vector space of dimension 3 over Real Field with 53 bits of precision
>>> v = vector(CDF, [Integer(1),Integer(2),Integer(3)])
>>> v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)
>>> _.parent()
Vector space of dimension 3 over Complex Field with 53 bits of precision

>>> v = vector(Integers(Integer(8)), [Integer(1),Integer(2),Integer(3)])
>>> v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)
>>> _.parent()
Vector space of dimension 3 over Real Field with 53 bits of precision
```

(continues on next page)

(continued from previous page)

```
>>> v.n(prec=Integer(75))
(1.0000000000000000000, 2.0000000000000000000, 3.0000000000000000000)
>>> _.parent()
Vector space of dimension 3 over Real Field with 75 bits of precision

>>> v = vector(QQ, [Integer(1),Integer(2),Integer(3)])
>>> v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)
>>> _.parent()
Vector space of dimension 3 over Real Field with 53 bits of precision
>>> v.n(prec=Integer(75))
(1.0000000000000000000, 2.0000000000000000000, 3.0000000000000000000)
>>> _.parent()
Vector space of dimension 3 over Real Field with 75 bits of precision
```

```
sage: v = vector(GF(2), [1,2,3])
sage: v.n()
(1.00000000000000, 0.000000000000000, 1.00000000000000)
sage: _.parent()
Vector space of dimension 3 over Real Field with 53 bits of precision
sage: v.n(prec=75)
(1.0000000000000000000, 0.0000000000000000000, 1.0000000000000000000)
sage: _.parent()
Vector space of dimension 3 over Real Field with 75 bits of precision
```

```
>>> from sage.all import *
>>> v = vector(GF(Integer(2)), [Integer(1),Integer(2),Integer(3)])
>>> v.n()
(1.00000000000000, 0.000000000000000, 1.00000000000000)
>>> _.parent()
Vector space of dimension 3 over Real Field with 53 bits of precision
>>> v.n(prec=Integer(75))
(1.0000000000000000000, 0.0000000000000000000, 1.0000000000000000000)
>>> _.parent()
Vector space of dimension 3 over Real Field with 75 bits of precision
```

numpy(*dtype='object'*)

Convert `self` to a numpy array.

INPUT:

- `dtype` – the numpy dtype of the returned array

EXAMPLES:

```
sage: # needs numpy
sage: v = vector([1,2,3])
sage: v.numpy()
array([1, 2, 3], dtype=object)
sage: v.numpy() * v.numpy()
array([1, 4, 9], dtype=object)

sage: vector(QQ, [1, 2, 5/6]).numpy()                                    #␣
```

(continues on next page)

(continued from previous page)

```
↪needs numpy
array([1, 2, 5/6], dtype=object)
```

```
>>> from sage.all import *
>>> # needs numpy
>>> v = vector([Integer(1),Integer(2),Integer(3)])
>>> v.numpy()
array([1, 2, 3], dtype=object)
>>> v.numpy() * v.numpy()
array([1, 4, 9], dtype=object)

>>> vector(QQ, [Integer(1), Integer(2), Integer(5)/Integer(6)]).numpy()
↪                                   # needs numpy
array([1, 2, 5/6], dtype=object)
```

By default, the `object` dtype is used. Alternatively, the desired dtype can be passed in as a parameter:

```
sage: # needs numpy
sage: v = vector(QQ, [1, 2, 5/6])
sage: v.numpy()
array([1, 2, 5/6], dtype=object)
sage: v.numpy(dtype=float)
array([1.        , 2.        , 0.83333333])
sage: v.numpy(dtype=int)
array([1, 2, 0])
sage: import numpy
sage: v.numpy(dtype=numpy.uint8)
array([1, 2, 0], dtype=uint8)
```

```
>>> from sage.all import *
>>> # needs numpy
>>> v = vector(QQ, [Integer(1), Integer(2), Integer(5)/Integer(6)])
>>> v.numpy()
array([1, 2, 5/6], dtype=object)
>>> v.numpy(dtype=float)
array([1.        , 2.        , 0.83333333])
>>> v.numpy(dtype=int)
array([1, 2, 0])
>>> import numpy
>>> v.numpy(dtype=numpy.uint8)
array([1, 2, 0], dtype=uint8)
```

Passing a dtype of None will let numpy choose a native type, which can be more efficient but may have unintended consequences:

```
sage: # needs numpy
sage: v.numpy(dtype=None)
array([1.        , 2.        , 0.83333333])

sage: w = vector(ZZ, [0, 1, 2^63 -1]); w
(0, 1, 9223372036854775807)
sage: wn = w.numpy(dtype=None); wn                                              #
```

(continues on next page)

(continued from previous page)

```
↪needs numpy
array([                  0,                   1, 9223372036854775807]...)
sage: wn.dtype                                                          #␣
↪needs numpy
dtype('int64')
sage: w.dot_product(w)
850705917302346158473969077842325012 50
sage: wn.dot(wn)        # overflow                                      #␣
↪needs numpy
2
```

```
>>> from sage.all import *
>>> # needs numpy
>>> v.numpy(dtype=None)
array([1.        , 2.        , 0.83333333])

>>> w = vector(ZZ, [Integer(0), Integer(1), Integer(2)**Integer(63) -
↪Integer(1)]); w
(0, 1, 9223372036854775807)
>>> wn = w.numpy(dtype=None); wn                                        #␣
↪needs numpy
array([                  0,                   1, 9223372036854775807]...)
>>> wn.dtype                                                            #␣
↪needs numpy
dtype('int64')
>>> w.dot_product(w)
850705917302346158473969077842325012 50
>>> wn.dot(wn)        # overflow                                        #␣
↪needs numpy
2
```

Numpy can give rather obscure errors; we wrap these to give a bit of context:

```
sage: vector([1, 1/2, QQ['x'].0]).numpy(dtype=float)                    #␣
↪needs numpy
Traceback (most recent call last):
...
ValueError: Could not convert vector over Univariate Polynomial Ring in x
over Rational Field to numpy array of type <... 'float'>:
setting an array element with a sequence.
```

```
>>> from sage.all import *
>>> vector([Integer(1), Integer(1)/Integer(2), QQ['x'].gen(0)]).
↪numpy(dtype=float)                        # needs numpy
Traceback (most recent call last):
...
ValueError: Could not convert vector over Univariate Polynomial Ring in x
over Rational Field to numpy array of type <... 'float'>:
setting an array element with a sequence.
```

outer_product(*right*)

Return a matrix, the outer product of two vectors `self` and `right`.

INPUT:

- `right` – a vector (or free module element) of any size, whose elements are compatible (with regard to multiplication) with the elements of `self`.

OUTPUT:

The outer product of two vectors x and y (respectively `self` and `right`) can be described several ways. If we interpret x as a $m \times 1$ matrix and interpret y as a $1 \times n$ matrix, then the outer product is the $m \times n$ matrix from the usual matrix product xy. Notice how this is the "opposite" in some ways from an inner product (which would require $m = n$).

If we just consider vectors, use each entry of x to create a scalar multiples of the vector y and use these vectors as the rows of a matrix. Or use each entry of y to create a scalar multiples of x and use these vectors as the columns of a matrix.

EXAMPLES:

```
sage: u = vector(QQ, [1/2, 1/3, 1/4, 1/5])
sage: v = vector(ZZ, [60, 180, 600])
sage: u.outer_product(v)
[ 30  90 300]
[ 20  60 200]
[ 15  45 150]
[ 12  36 120]
sage: M = v.outer_product(u); M
[ 30  20  15  12]
[ 90  60  45  36]
[300 200 150 120]
sage: M.parent()
Full MatrixSpace of 3 by 4 dense matrices over Rational Field
```

```
>>> from sage.all import *
>>> u = vector(QQ, [Integer(1)/Integer(2), Integer(1)/Integer(3), Integer(1)/
↪Integer(4), Integer(1)/Integer(5)])
>>> v = vector(ZZ, [Integer(60), Integer(180), Integer(600)])
>>> u.outer_product(v)
[ 30  90 300]
[ 20  60 200]
[ 15  45 150]
[ 12  36 120]
>>> M = v.outer_product(u); M
[ 30  20  15  12]
[ 90  60  45  36]
[300 200 150 120]
>>> M.parent()
Full MatrixSpace of 3 by 4 dense matrices over Rational Field
```

The more general `sage.matrix.matrix2.tensor_product()` is an operation on a pair of matrices. If we construct a pair of vectors as a column vector and a row vector, then an outer product and a tensor product are identical. Thus `tensor_product` is a synonym for this method.

```
sage: u = vector(QQ, [1/2, 1/3, 1/4, 1/5])
sage: v = vector(ZZ, [60, 180, 600])
sage: u.tensor_product(v) == (u.column()).tensor_product(v.row())
True
```

```
>>> from sage.all import *
>>> u = vector(QQ, [Integer(1)/Integer(2), Integer(1)/Integer(3), Integer(1)/
↪Integer(4), Integer(1)/Integer(5)])
>>> v = vector(ZZ, [Integer(60), Integer(180), Integer(600)])
>>> u.tensor_product(v) == (u.column()).tensor_product(v.row())
True
```

The result is always a dense matrix, no matter if the two vectors are, or are not, dense.

```
sage: d = vector(ZZ,[4,5], sparse=False)
sage: s = vector(ZZ, [1,2,3], sparse=True)
sage: dd = d.outer_product(d)
sage: ds = d.outer_product(s)
sage: sd = s.outer_product(d)
sage: ss = s.outer_product(s)
sage: all([dd.is_dense(), ds.is_dense(), sd.is_dense(), dd.is_dense()])
True
```

```
>>> from sage.all import *
>>> d = vector(ZZ,[Integer(4),Integer(5)], sparse=False)
>>> s = vector(ZZ, [Integer(1),Integer(2),Integer(3)], sparse=True)
>>> dd = d.outer_product(d)
>>> ds = d.outer_product(s)
>>> sd = s.outer_product(d)
>>> ss = s.outer_product(s)
>>> all([dd.is_dense(), ds.is_dense(), sd.is_dense(), dd.is_dense()])
True
```

Vectors with no entries do the right thing.

```
sage: v = vector(ZZ, [])
sage: z = v.outer_product(v)
sage: z.parent()
Full MatrixSpace of 0 by 0 dense matrices over Integer Ring
```

```
>>> from sage.all import *
>>> v = vector(ZZ, [])
>>> z = v.outer_product(v)
>>> z.parent()
Full MatrixSpace of 0 by 0 dense matrices over Integer Ring
```

There is a fair amount of latitude in the value of the `right` vector, and the matrix that results can have entries from a new ring large enough to contain the result. If you know better, you can sometimes bring the result down to a less general ring.

```
sage: R.<t> = ZZ[]
sage: v = vector(R, [12, 24*t])
sage: w = vector(QQ, [1/2, 1/3, 1/4])
sage: op = v.outer_product(w); op
[   6    4    3]
[12*t  8*t  6*t]
sage: op.base_ring()
Univariate Polynomial Ring in t over Rational Field
```

(continues on next page)

(continued from previous page)

```
sage: m = op.change_ring(R); m
[   6    4    3]
[12*t  8*t  6*t]
sage: m.base_ring()
Univariate Polynomial Ring in t over Integer Ring
```

```
>>> from sage.all import *
>>> R = ZZ['t']; (t,) = R._first_ngens(1)
>>> v = vector(R, [Integer(12), Integer(24)*t])
>>> w = vector(QQ, [Integer(1)/Integer(2), Integer(1)/Integer(3), Integer(1)/
↪Integer(4)])
>>> op = v.outer_product(w); op
[   6    4    3]
[12*t  8*t  6*t]
>>> op.base_ring()
Univariate Polynomial Ring in t over Rational Field
>>> m = op.change_ring(R); m
[   6    4    3]
[12*t  8*t  6*t]
>>> m.base_ring()
Univariate Polynomial Ring in t over Integer Ring
```

But some inputs are not compatible, even if vectors.

```
sage: w = vector(GF(5), [1,2])
sage: v = vector(GF(7), [1,2,3,4])
sage: z = w.outer_product(v)
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *:
'Full MatrixSpace of 2 by 1 dense matrices over Finite Field of size 5' and
'Full MatrixSpace of 1 by 4 dense matrices over Finite Field of size 7'
```

```
>>> from sage.all import *
>>> w = vector(GF(Integer(5)), [Integer(1),Integer(2)])
>>> v = vector(GF(Integer(7)), [Integer(1),Integer(2),Integer(3),Integer(4)])
>>> z = w.outer_product(v)
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *:
'Full MatrixSpace of 2 by 1 dense matrices over Finite Field of size 5' and
'Full MatrixSpace of 1 by 4 dense matrices over Finite Field of size 7'
```

And some inputs don't make any sense at all.

```
sage: w = vector(QQ, [5,10])
sage: z = w.outer_product(6)
Traceback (most recent call last):
...
TypeError: right operand in an outer product must be a vector,
not an element of Integer Ring
```

```
>>> from sage.all import *
>>> w = vector(QQ, [Integer(5),Integer(10)])
>>> z = w.outer_product(Integer(6))
Traceback (most recent call last):
...
TypeError: right operand in an outer product must be a vector,
not an element of Integer Ring
```

pairwise_product(*right*)

Return the pairwise product of `self` and `right`, which is a vector of the products of the corresponding entries.

INPUT:

- `right` – vector of the same degree as `self`. It need not be in the same vector space as `self`, as long as the coefficients can be multiplied.

EXAMPLES:

```
sage: V = FreeModule(ZZ, 3)
sage: v = V([1,2,3])
sage: w = V([4,5,6])
sage: v.pairwise_product(w)
(4, 10, 18)
sage: sum(v.pairwise_product(w)) == v.dot_product(w)
True
```

```
>>> from sage.all import *
>>> V = FreeModule(ZZ, Integer(3))
>>> v = V([Integer(1),Integer(2),Integer(3)])
>>> w = V([Integer(4),Integer(5),Integer(6)])
>>> v.pairwise_product(w)
(4, 10, 18)
>>> sum(v.pairwise_product(w)) == v.dot_product(w)
True
```

```
sage: W = VectorSpace(GF(3), 3)
sage: w = W([0,1,2])
sage: w.pairwise_product(v)
(0, 2, 0)
sage: w.pairwise_product(v).parent()
Vector space of dimension 3 over Finite Field of size 3
```

```
>>> from sage.all import *
>>> W = VectorSpace(GF(Integer(3)), Integer(3))
>>> w = W([Integer(0),Integer(1),Integer(2)])
>>> w.pairwise_product(v)
(0, 2, 0)
>>> w.pairwise_product(v).parent()
Vector space of dimension 3 over Finite Field of size 3
```

Implicit coercion is well defined (regardless of order), so we get 2 even if we do the dot product in the other order.

```
sage: v.pairwise_product(w).parent()
Vector space of dimension 3 over Finite Field of size 3
```

```
>>> from sage.all import *
>>> v.pairwise_product(w).parent()
Vector space of dimension 3 over Finite Field of size 3
```

plot(*plot_type=None*, *start=None*, ***kwds*)

INPUT:

- **plot_type** – **(default: 'arrow' if v has 3 or fewer components,** otherwise 'step') type of plot. Options are:
  - 'arrow' to draw an arrow
  - 'point' to draw a point at the coordinates specified by the vector
  - 'step' to draw a step function representing the coordinates of the vector.

  Both 'arrow' and 'point' raise exceptions if the vector has more than 3 dimensions.
- `start` – (default: origin in correct dimension) may be a tuple, list, or vector

EXAMPLES:

The following both plot the given vector:

```
sage: v = vector(RDF, (1,2))
sage: A = plot(v)                                                           #␣
↪needs sage.plot
sage: B = v.plot()                                                          #␣
↪needs sage.plot
sage: A + B  # should just show one vector                                  #␣
↪needs sage.plot
Graphics object consisting of 2 graphics primitives
```

```
>>> from sage.all import *
>>> v = vector(RDF, (Integer(1),Integer(2)))
>>> A = plot(v)                                                           #␣
↪needs sage.plot
>>> B = v.plot()                                                          #␣
↪needs sage.plot
>>> A + B  # should just show one vector                                  #␣
↪needs sage.plot
Graphics object consisting of 2 graphics primitives
```

Examples of the plot types:

```
sage: # needs sage.plot
sage: A = plot(v, plot_type='arrow')
sage: B = plot(v, plot_type='point', color='green', size=20)
sage: C = plot(v, plot_type='step') # calls v.plot_step()
sage: A+B+C
Graphics object consisting of 3 graphics primitives
```

```
>>> from sage.all import *
>>> # needs sage.plot
>>> A = plot(v, plot_type='arrow')
>>> B = plot(v, plot_type='point', color='green', size=Integer(20))
>>> C = plot(v, plot_type='step') # calls v.plot_step()
>>> A+B+C
Graphics object consisting of 3 graphics primitives
```

You can use the optional arguments for *plot_step()*:

```
sage: eps = 0.1
sage: plot(v, plot_type='step', eps=eps, xmax=5, hue=0)                    #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> eps = RealNumber('0.1')
>>> plot(v, plot_type='step', eps=eps, xmax=Integer(5), hue=Integer(0))      ␣
↪              # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

Three-dimensional examples:

```
sage: v = vector(RDF, (1,2,1))
sage: plot(v)  # defaults to an arrow plot                                 #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> v = vector(RDF, (Integer(1),Integer(2),Integer(1)))
>>> plot(v)  # defaults to an arrow plot                                   #␣
↪needs sage.plot
Graphics3d Object
```

```
sage: plot(v, plot_type='arrow')                                           #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> plot(v, plot_type='arrow')                                             #␣
↪needs sage.plot
Graphics3d Object
```

```
sage: from sage.plot.plot3d.shapes2 import frame3d                         #␣
↪needs sage.plot
sage: plot(v, plot_type='point')+frame3d((0,0,0), v.list())                #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> from sage.plot.plot3d.shapes2 import frame3d                           #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.plot
>>> plot(v, plot_type='point')+frame3d((Integer(0),Integer(0),Integer(0)), v.
↪list())                  # needs sage.plot
Graphics3d Object
```

```
sage: plot(v, plot_type='step')  # calls v.plot_step()                    #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> plot(v, plot_type='step')  # calls v.plot_step()                     #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
sage: plot(v, plot_type='step', eps=eps, xmax=5, hue=0)                  #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> plot(v, plot_type='step', eps=eps, xmax=Integer(5), hue=Integer(0))      ␣
↪              # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

With greater than three coordinates, it defaults to a step plot:

```
sage: v = vector(RDF, (1,2,3,4))
sage: plot(v)                                                            #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> v = vector(RDF, (Integer(1),Integer(2),Integer(3),Integer(4)))
>>> plot(v)                                                              #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

One dimensional vectors are plotted along the horizontal axis of the coordinate plane:

```
sage: plot(vector([1]))                                                  #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> plot(vector([Integer(1)]))                                               ␣
↪     # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

An optional start argument may also be specified by a tuple, list, or vector:

```
sage: u = vector([1,2]); v = vector([2,5])
sage: plot(u, start=v)                                                   #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> u = vector([Integer(1),Integer(2)]); v = vector([Integer(2),Integer(5)])
>>> plot(u, start=v)                                                    #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

plot_step(*xmin=0*, *xmax=1*, *eps=None*, *res=None*, *connect=True*, ***kwds*)

INPUT:

- `xmin` – (default: 0) start x position to start plotting
- `xmax` – (default: 1) stop x position to stop plotting
- `eps` – (default: determined by xmax) we view this vector as defining a function at the points xmin, xmin + eps, xmin + 2*eps, …,
- `res` – (default: all points) total number of points to include in the graph
- `connect` – boolean (default: `True`); if `True` draws a line, otherwise draw a list of points

EXAMPLES:

```
sage: eps = 0.1
sage: v = vector(RDF, [sin(n*eps) for n in range(100)])
sage: v.plot_step(eps=eps, xmax=5, hue=0)                               #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> eps = RealNumber('0.1')
>>> v = vector(RDF, [sin(n*eps) for n in range(Integer(100))])
>>> v.plot_step(eps=eps, xmax=Integer(5), hue=Integer(0))                ␣
↪              # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

row()

Return a matrix with a single row and the same entries as the vector `self`.

OUTPUT:

A matrix over the same ring as the vector (or free module element), with a single row. The entries of the row are identical to those of the vector, and in the same order.

EXAMPLES:

```
sage: v = vector(ZZ, [1,2,3])
sage: w = v.row(); w
[1 2 3]
sage: w.parent()
Full MatrixSpace of 1 by 3 dense matrices over Integer Ring

sage: x = vector(FiniteField(13), [2,4,8,16])
```

(continues on next page)

(continued from previous page)

```
sage: x.row()
[2 4 8 3]
```

```
>>> from sage.all import *
>>> v = vector(ZZ, [Integer(1),Integer(2),Integer(3)])
>>> w = v.row(); w
[1 2 3]
>>> w.parent()
Full MatrixSpace of 1 by 3 dense matrices over Integer Ring

>>> x = vector(FiniteField(Integer(13)), [Integer(2),Integer(4),Integer(8),
↪Integer(16)])
>>> x.row()
[2 4 8 3]
```

There is more than one way to get one-row matrix from a vector, but the `row` method is more efficient than making a column and then taking a transpose. Notice that supplying a vector to the matrix constructor demonstrates Sage's preference for rows.

```
sage: x = vector(RDF, [sin(i*pi/20) for i in range(10)])                    #␣
↪needs sage.symbolic
sage: x.row() == matrix(x)
True
sage: x.row() == x.column().transpose()
True
```

```
>>> from sage.all import *
>>> x = vector(RDF, [sin(i*pi/Integer(20)) for i in range(Integer(10))])     ␣
↪              # needs sage.symbolic
>>> x.row() == matrix(x)
True
>>> x.row() == x.column().transpose()
True
```

Sparse or dense implementations are preserved.

```
sage: d = vector(RR, [1.0, 2.0, 3.0])
sage: s = vector(CDF, {2: 5.0+6.0*I})                                       #␣
↪needs sage.symbolic
sage: dm = d.row()
sage: sm = s.row()                                                          #␣
↪needs sage.symbolic
sage: all([d.is_dense(), dm.is_dense(), s.is_sparse(), sm.is_sparse()])     #␣
↪needs sage.symbolic
True
```

```
>>> from sage.all import *
>>> d = vector(RR, [RealNumber('1.0'), RealNumber('2.0'), RealNumber('3.0')])
>>> s = vector(CDF, {Integer(2): RealNumber('5.0')+RealNumber('6.0')*I})     ␣
↪                                  # needs sage.symbolic
>>> dm = d.row()
>>> sm = s.row()                                                            #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.symbolic
>>> all([d.is_dense(), dm.is_dense(), s.is_sparse(), sm.is_sparse()])      #␣
↪needs sage.symbolic
True
```

set(*i*, *value*)

Like `__setitem__` but without type or bounds checking: i must satisfy `0 <= i < self.degree` and `value` must be an element of the coordinate ring.

EXAMPLES:

```
sage: v = vector(SR, [1/2,2/5,0]); v                                        #␣
↪needs sage.symbolic
(1/2, 2/5, 0)
sage: v.set(2, pi); v                                                       #␣
↪needs sage.symbolic
(1/2, 2/5, pi)
```

```
>>> from sage.all import *
>>> v = vector(SR, [Integer(1)/Integer(2),Integer(2)/Integer(5),Integer(0)]);␣
↪v                                          # needs sage.symbolic
(1/2, 2/5, 0)
>>> v.set(Integer(2), pi); v                                                 ␣
↪     # needs sage.symbolic
(1/2, 2/5, pi)
```

sparse_vector()

Return sparse version of `self`. If `self` is sparse, just return self; otherwise, create and return correspond sparse vector.

EXAMPLES:

```
sage: vector([-1,0,3,0,0,0]).sparse_vector().is_sparse()
True
sage: vector([-1,0,3,0,0,0]).sparse_vector().is_sparse()
True
sage: vector([-1,0,3,0,0,0]).sparse_vector()
(-1, 0, 3, 0, 0, 0)
```

```
>>> from sage.all import *
>>> vector([-Integer(1),Integer(0),Integer(3),Integer(0),Integer(0),
↪Integer(0)]).sparse_vector().is_sparse()
True
>>> vector([-Integer(1),Integer(0),Integer(3),Integer(0),Integer(0),
↪Integer(0)]).sparse_vector().is_sparse()
True
>>> vector([-Integer(1),Integer(0),Integer(3),Integer(0),Integer(0),
↪Integer(0)]).sparse_vector()
(-1, 0, 3, 0, 0, 0)
```

subs(*in_dict=None*, ***kwds*)

EXAMPLES:

```
sage: # needs sage.symbolic
sage: var('a,b,d,e')
(a, b, d, e)
sage: v = vector([a, b, d, e])
sage: v.substitute(a=1)
(1, b, d, e)
sage: v.subs(a=b, b=d)
(b, d, d, e)
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> var('a,b,d,e')
(a, b, d, e)
>>> v = vector([a, b, d, e])
>>> v.substitute(a=Integer(1))
(1, b, d, e)
>>> v.subs(a=b, b=d)
(b, d, d, e)
```

support()

Return the integers `i` such that `self[i] != 0`. This is the same as the `nonzero_positions` function.

EXAMPLES:

```
sage: vector([-1,0,3,0,0,0,0.01]).support()
[0, 2, 6]
```

```
>>> from sage.all import *
>>> vector([-Integer(1),Integer(0),Integer(3),Integer(0),Integer(0),
↪Integer(0),RealNumber('0.01')]).support()
[0, 2, 6]
```

tensor_product(*right*)

Return a matrix, the outer product of two vectors `self` and `right`.

INPUT:

- `right` – a vector (or free module element) of any size, whose elements are compatible (with regard to multiplication) with the elements of `self`.

OUTPUT:

The outer product of two vectors x and y (respectively `self` and `right`) can be described several ways. If we interpret x as a $m \times 1$ matrix and interpret y as a $1 \times n$ matrix, then the outer product is the $m \times n$ matrix from the usual matrix product xy. Notice how this is the "opposite" in some ways from an inner product (which would require $m = n$).

If we just consider vectors, use each entry of x to create a scalar multiples of the vector y and use these vectors as the rows of a matrix. Or use each entry of y to create a scalar multiples of x and use these vectors as the columns of a matrix.

EXAMPLES:

```
sage: u = vector(QQ, [1/2, 1/3, 1/4, 1/5])
sage: v = vector(ZZ, [60, 180, 600])
```

(continues on next page)

(continued from previous page)

```
sage: u.outer_product(v)
[ 30  90 300]
[ 20  60 200]
[ 15  45 150]
[ 12  36 120]
sage: M = v.outer_product(u); M
[ 30  20  15  12]
[ 90  60  45  36]
[300 200 150 120]
sage: M.parent()
Full MatrixSpace of 3 by 4 dense matrices over Rational Field
```

```
>>> from sage.all import *
>>> u = vector(QQ, [Integer(1)/Integer(2), Integer(1)/Integer(3), Integer(1)/
↪Integer(4), Integer(1)/Integer(5)])
>>> v = vector(ZZ, [Integer(60), Integer(180), Integer(600)])
>>> u.outer_product(v)
[ 30  90 300]
[ 20  60 200]
[ 15  45 150]
[ 12  36 120]
>>> M = v.outer_product(u); M
[ 30  20  15  12]
[ 90  60  45  36]
[300 200 150 120]
>>> M.parent()
Full MatrixSpace of 3 by 4 dense matrices over Rational Field
```

The more general `sage.matrix.matrix2.tensor_product()` is an operation on a pair of matrices. If we construct a pair of vectors as a column vector and a row vector, then an outer product and a tensor product are identical. Thus `tensor_product` is a synonym for this method.

```
sage: u = vector(QQ, [1/2, 1/3, 1/4, 1/5])
sage: v = vector(ZZ, [60, 180, 600])
sage: u.tensor_product(v) == (u.column()).tensor_product(v.row())
True
```

```
>>> from sage.all import *
>>> u = vector(QQ, [Integer(1)/Integer(2), Integer(1)/Integer(3), Integer(1)/
↪Integer(4), Integer(1)/Integer(5)])
>>> v = vector(ZZ, [Integer(60), Integer(180), Integer(600)])
>>> u.tensor_product(v) == (u.column()).tensor_product(v.row())
True
```

The result is always a dense matrix, no matter if the two vectors are, or are not, dense.

```
sage: d = vector(ZZ,[4,5], sparse=False)
sage: s = vector(ZZ, [1,2,3], sparse=True)
sage: dd = d.outer_product(d)
sage: ds = d.outer_product(s)
sage: sd = s.outer_product(d)
sage: ss = s.outer_product(s)
```

(continues on next page)

(continued from previous page)

```
sage: all([dd.is_dense(), ds.is_dense(), sd.is_dense(), dd.is_dense()])
True
```

```
>>> from sage.all import *
>>> d = vector(ZZ,[Integer(4),Integer(5)], sparse=False)
>>> s = vector(ZZ, [Integer(1),Integer(2),Integer(3)], sparse=True)
>>> dd = d.outer_product(d)
>>> ds = d.outer_product(s)
>>> sd = s.outer_product(d)
>>> ss = s.outer_product(s)
>>> all([dd.is_dense(), ds.is_dense(), sd.is_dense(), dd.is_dense()])
True
```

Vectors with no entries do the right thing.

```
sage: v = vector(ZZ, [])
sage: z = v.outer_product(v)
sage: z.parent()
Full MatrixSpace of 0 by 0 dense matrices over Integer Ring
```

```
>>> from sage.all import *
>>> v = vector(ZZ, [])
>>> z = v.outer_product(v)
>>> z.parent()
Full MatrixSpace of 0 by 0 dense matrices over Integer Ring
```

There is a fair amount of latitude in the value of the `right` vector, and the matrix that results can have entries from a new ring large enough to contain the result. If you know better, you can sometimes bring the result down to a less general ring.

```
sage: R.<t> = ZZ[]
sage: v = vector(R, [12, 24*t])
sage: w = vector(QQ, [1/2, 1/3, 1/4])
sage: op = v.outer_product(w); op
[   6    4    3]
[12*t  8*t  6*t]
sage: op.base_ring()
Univariate Polynomial Ring in t over Rational Field
sage: m = op.change_ring(R); m
[   6    4    3]
[12*t  8*t  6*t]
sage: m.base_ring()
Univariate Polynomial Ring in t over Integer Ring
```

```
>>> from sage.all import *
>>> R = ZZ['t']; (t,) = R._first_ngens(1)
>>> v = vector(R, [Integer(12), Integer(24)*t])
>>> w = vector(QQ, [Integer(1)/Integer(2), Integer(1)/Integer(3), Integer(1)/
↪Integer(4)])
>>> op = v.outer_product(w); op
[   6    4    3]
[12*t  8*t  6*t]
```

(continues on next page)

(continued from previous page)

```
>>> op.base_ring()
Univariate Polynomial Ring in t over Rational Field
>>> m = op.change_ring(R); m
[   6    4    3]
[12*t  8*t  6*t]
>>> m.base_ring()
Univariate Polynomial Ring in t over Integer Ring
```

But some inputs are not compatible, even if vectors.

```
sage: w = vector(GF(5), [1,2])
sage: v = vector(GF(7), [1,2,3,4])
sage: z = w.outer_product(v)
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *:
'Full MatrixSpace of 2 by 1 dense matrices over Finite Field of size 5' and
'Full MatrixSpace of 1 by 4 dense matrices over Finite Field of size 7'
```

```
>>> from sage.all import *
>>> w = vector(GF(Integer(5)), [Integer(1),Integer(2)])
>>> v = vector(GF(Integer(7)), [Integer(1),Integer(2),Integer(3),Integer(4)])
>>> z = w.outer_product(v)
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *:
'Full MatrixSpace of 2 by 1 dense matrices over Finite Field of size 5' and
'Full MatrixSpace of 1 by 4 dense matrices over Finite Field of size 7'
```

And some inputs don't make any sense at all.

```
sage: w = vector(QQ, [5,10])
sage: z = w.outer_product(6)
Traceback (most recent call last):
...
TypeError: right operand in an outer product must be a vector,
not an element of Integer Ring
```

```
>>> from sage.all import *
>>> w = vector(QQ, [Integer(5),Integer(10)])
>>> z = w.outer_product(Integer(6))
Traceback (most recent call last):
...
TypeError: right operand in an outer product must be a vector,
not an element of Integer Ring
```

class sage.modules.free_module_element.**FreeModuleElement_generic_dense**

Bases: *FreeModuleElement*

A generic dense element of a free module.

function(**args*)

Return a vector over a callable symbolic expression ring.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: x, y = var('x,y')
sage: v = vector([x, y, x*sin(y)])
sage: w = v.function([x,y]); w
(x, y) |--> (x, y, x*sin(y))
sage: w.coordinate_ring()
Callable function ring with arguments (x, y)
sage: w(1,2)
(1, 2, sin(2))
sage: w(2,1)
(2, 1, 2*sin(1))
sage: w(y=1,x=2)
(2, 1, 2*sin(1))
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> x, y = var('x,y')
>>> v = vector([x, y, x*sin(y)])
>>> w = v.function([x,y]); w
(x, y) |--> (x, y, x*sin(y))
>>> w.coordinate_ring()
Callable function ring with arguments (x, y)
>>> w(Integer(1),Integer(2))
(1, 2, sin(2))
>>> w(Integer(2),Integer(1))
(2, 1, 2*sin(1))
>>> w(y=Integer(1),x=Integer(2))
(2, 1, 2*sin(1))
```

```
sage: # needs sage.symbolic
sage: x,y = var('x,y')
sage: v = vector([x, y, x*sin(y)])
sage: w = v.function([x]); w
x |--> (x, y, x*sin(y))
sage: w.coordinate_ring()
Callable function ring with argument x
sage: w(4)
(4, y, 4*sin(y))
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> x,y = var('x,y')
>>> v = vector([x, y, x*sin(y)])
>>> w = v.function([x]); w
x |--> (x, y, x*sin(y))
>>> w.coordinate_ring()
Callable function ring with argument x
>>> w(Integer(4))
(4, y, 4*sin(y))
```

list(*copy=True*)

Return list of elements of `self`.

INPUT:

- `copy` – boolean; return list of underlying entries

EXAMPLES:

```
sage: P.<x,y,z> = QQ[]
sage: v = vector([x,y,z])
sage: type(v)
<class 'sage.modules.free_module_element.FreeModuleElement_generic_dense'>
sage: a = v.list(); a
[x, y, z]
sage: a[0] = x*y; v
(x, y, z)
sage: a = v.list(copy=False); a
[x, y, z]
sage: a[0] = x*y; v
(x*y, y, z)
```

```
>>> from sage.all import *
>>> P = QQ['x, y, z']; (x, y, z,) = P._first_ngens(3)
>>> v = vector([x,y,z])
>>> type(v)
<class 'sage.modules.free_module_element.FreeModuleElement_generic_dense'>
>>> a = v.list(); a
[x, y, z]
>>> a[Integer(0)] = x*y; v
(x, y, z)
>>> a = v.list(copy=False); a
[x, y, z]
>>> a[Integer(0)] = x*y; v
(x*y, y, z)
```

class sage.modules.free_module_element.**FreeModuleElement_generic_sparse**

Bases: *FreeModuleElement*

A generic sparse free module element is a dictionary with keys ints i and entries in the base ring.

denominator()

Return the least common multiple of the denominators of the entries of `self`.

EXAMPLES:

```
sage: v = vector([1/2,2/5,3/14], sparse=True)
sage: v.denominator()
70
```

```
>>> from sage.all import *
>>> v = vector([Integer(1)/Integer(2),Integer(2)/Integer(5),Integer(3)/
↪Integer(14)], sparse=True)
>>> v.denominator()
70
```

dict(*copy=True*)

Return dictionary of nonzero entries of `self`.

More precisely, this returns 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`.

OUTPUT: Python dictionary

EXAMPLES:

```
sage: v = vector([0,0,0,0,1/2,0,3/14], sparse=True)
sage: v.dict()
{4: 1/2, 6: 3/14}
sage: sorted(v.support())
[4, 6]
```

```
>>> from sage.all import *
>>> v = vector([Integer(0),Integer(0),Integer(0),Integer(0),Integer(1)/
↪Integer(2),Integer(0),Integer(3)/Integer(14)], sparse=True)
>>> v.dict()
{4: 1/2, 6: 3/14}
>>> sorted(v.support())
[4, 6]
```

hamming_weight()

Return the number of positions `i` such that `self[i] != 0`.

EXAMPLES:

```
sage: v = vector({1: 1, 3: -2})
sage: w = vector({1: 4, 3: 2})
sage: v+w
(0, 5, 0, 0)
sage: (v+w).hamming_weight()
1
```

```
>>> from sage.all import *
>>> v = vector({Integer(1): Integer(1), Integer(3): -Integer(2)})
>>> w = vector({Integer(1): Integer(4), Integer(3): Integer(2)})
>>> v+w
(0, 5, 0, 0)
>>> (v+w).hamming_weight()
1
```

items()

Return an iterator over the entries of `self`.

EXAMPLES:

```
sage: v = vector([1,2/3,pi], sparse=True)                                    #␣
↪needs sage.symbolic
```

(continues on next page)

(continued from previous page)

```
sage: next(v.items())                                                       #␣
↪needs sage.symbolic
(0, 1)
sage: list(v.items())                                                       #␣
↪needs sage.symbolic
[(0, 1), (1, 2/3), (2, pi)]
```

```
>>> from sage.all import *
>>> v = vector([Integer(1),Integer(2)/Integer(3),pi], sparse=True)          ␣
↪                        # needs sage.symbolic
>>> next(v.items())                                                        #␣
↪needs sage.symbolic
(0, 1)
>>> list(v.items())                                                        #␣
↪needs sage.symbolic
[(0, 1), (1, 2/3), (2, pi)]
```

iteritems()

Return an iterator over the entries of `self`.

EXAMPLES:

```
sage: v = vector([1,2/3,pi], sparse=True)                                   #␣
↪needs sage.symbolic
sage: next(v.items())                                                       #␣
↪needs sage.symbolic
(0, 1)
sage: list(v.items())                                                       #␣
↪needs sage.symbolic
[(0, 1), (1, 2/3), (2, pi)]
```

```
>>> from sage.all import *
>>> v = vector([Integer(1),Integer(2)/Integer(3),pi], sparse=True)          ␣
↪                        # needs sage.symbolic
>>> next(v.items())                                                        #␣
↪needs sage.symbolic
(0, 1)
>>> list(v.items())                                                        #␣
↪needs sage.symbolic
[(0, 1), (1, 2/3), (2, pi)]
```

list(*copy=True*)

Return list of elements of `self`.

INPUT:

- `copy` – ignored for sparse vectors

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: M = FreeModule(R, 3, sparse=True) * (1/x)
sage: v = M([-x^2, 3/x, 0])
```

(continues on next page)

(continued from previous page)

```
sage: type(v)
<class 'sage.modules.free_module_element.FreeModuleElement_generic_sparse'>
sage: a = v.list()
sage: a
[-x^2, 3/x, 0]
sage: [parent(c) for c in a]
[Fraction Field of Univariate Polynomial Ring in x over Rational Field,
 Fraction Field of Univariate Polynomial Ring in x over Rational Field,
 Fraction Field of Univariate Polynomial Ring in x over Rational Field]
```

```
>>> from sage.all import *
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> M = FreeModule(R, Integer(3), sparse=True) * (Integer(1)/x)
>>> v = M([-x**Integer(2), Integer(3)/x, Integer(0)])
>>> type(v)
<class 'sage.modules.free_module_element.FreeModuleElement_generic_sparse'>
>>> a = v.list()
>>> a
[-x^2, 3/x, 0]
>>> [parent(c) for c in a]
[Fraction Field of Univariate Polynomial Ring in x over Rational Field,
 Fraction Field of Univariate Polynomial Ring in x over Rational Field,
 Fraction Field of Univariate Polynomial Ring in x over Rational Field]
```

nonzero_positions()

Return the list of numbers `i` such that `self[i] != 0`.

EXAMPLES:

```
sage: v = vector({1: 1, 3: -2})
sage: w = vector({1: 4, 3: 2})
sage: v+w
(0, 5, 0, 0)
sage: (v+w).nonzero_positions()
[1]
```

```
>>> from sage.all import *
>>> v = vector({Integer(1): Integer(1), Integer(3): -Integer(2)})
>>> w = vector({Integer(1): Integer(4), Integer(3): Integer(2)})
>>> v+w
(0, 5, 0, 0)
>>> (v+w).nonzero_positions()
[1]
```

numerical_approx(*prec=None*, *digits=None*, *algorithm=None*)

Return a numerical approximation of `self` with `prec` bits (or decimal `digits`) of precision, by approximating all entries.

INPUT:

- `prec` – precision in bits
- `digits` – precision in decimal digits (only used if `prec` is not given)

- `algorithm` – which algorithm to use to compute the approximation of the entries (the accepted algorithms depend on the object)

If neither `prec` nor `digits` is given, the default precision is 53 bits (roughly 16 digits).

EXAMPLES:

```
sage: v = vector(RealField(200), [1,2,3], sparse=True)
sage: v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)
sage: _.parent()
Sparse vector space of dimension 3 over Real Field with 53 bits of precision
sage: v.n(prec=75)
(1.0000000000000000000, 2.0000000000000000000, 3.0000000000000000000)
sage: _.parent()
Sparse vector space of dimension 3 over Real Field with 75 bits of precision
```

```
>>> from sage.all import *
>>> v = vector(RealField(Integer(200)), [Integer(1),Integer(2),Integer(3)],␣
↪sparse=True)
>>> v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)
>>> _.parent()
Sparse vector space of dimension 3 over Real Field with 53 bits of precision
>>> v.n(prec=Integer(75))
(1.0000000000000000000, 2.0000000000000000000, 3.0000000000000000000)
>>> _.parent()
Sparse vector space of dimension 3 over Real Field with 75 bits of precision
```

sage.modules.free_module_element.**free_module_element**(*arg0*, *arg1=None*, *arg2=None*, *sparse=None*, *immutable=False*)

Return a vector or free module element with specified entries.

CALL FORMATS:

This constructor can be called in several different ways. In each case, `sparse=True` or `sparse=False` as well as `immutable=True` or `immutable=False` can be supplied as an option. `free_module_element()` is an alias for `vector()`.

1. vector(object)
2. vector(ring, object)
3. vector(object, ring)
4. vector(ring, degree, object)
5. vector(ring, degree)

INPUT:

- `object` – list, dictionary, or other iterable containing the entries of the vector, including any object that is palatable to the `Sequence` constructor
- `ring` – a base ring (or field) for the vector space or free module, which contains all of the elements
- `degree` – integer specifying the number of entries in the vector or free module element
- `sparse` – boolean, whether the result should be a sparse vector
- `immutable` – boolean (default: `False`); whether the result should be an immutable vector

In call format 4, an error is raised if the `degree` does not match the length of `object` so this call can provide some safeguards. Note however that using this format when `object` is a dictionary is unlikely to work properly.

OUTPUT:

An element of the ambient vector space or free module with the given base ring and implied or specified dimension or rank, containing the specified entries and with correct degree.

In call format 5, no entries are specified, so the element is populated with all zeros.

If the `sparse` option is not supplied, the output will generally have a dense representation. The exception is if `object` is a dictionary, then the representation will be sparse.

EXAMPLES:

```
sage: v = vector([1,2,3]); v
(1, 2, 3)
sage: v.parent()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
sage: v = vector([1,2,3/5]); v
(1, 2, 3/5)
sage: v.parent()
Vector space of dimension 3 over Rational Field
```

```
>>> from sage.all import *
>>> v = vector([Integer(1),Integer(2),Integer(3)]); v
(1, 2, 3)
>>> v.parent()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
>>> v = vector([Integer(1),Integer(2),Integer(3)/Integer(5)]); v
(1, 2, 3/5)
>>> v.parent()
Vector space of dimension 3 over Rational Field
```

All entries must *canonically* coerce to some common ring:

```
sage: v = vector([17, GF(11)(5), 19/3]); v
Traceback (most recent call last):
...
TypeError: unable to find a common ring for all elements
```

```
>>> from sage.all import *
>>> v = vector([Integer(17), GF(Integer(11))(Integer(5)), Integer(19)/
↪Integer(3)]); v
Traceback (most recent call last):
...
TypeError: unable to find a common ring for all elements
```

```
sage: v = vector([17, GF(11)(5), 19]); v
(6, 5, 8)
sage: v.parent()
Vector space of dimension 3 over Finite Field of size 11
sage: v = vector([17, GF(11)(5), 19], QQ); v
(17, 5, 19)
sage: v.parent()
```

(continues on next page)

(continued from previous page)

```
Vector space of dimension 3 over Rational Field
sage: v = vector((1,2,3), QQ); v
(1, 2, 3)
sage: v.parent()
Vector space of dimension 3 over Rational Field
sage: v = vector(QQ, (1,2,3)); v
(1, 2, 3)
sage: v.parent()
Vector space of dimension 3 over Rational Field
sage: v = vector(vector([1,2,3])); v
(1, 2, 3)
sage: v.parent()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> v = vector([Integer(17), GF(Integer(11))(Integer(5)), Integer(19)]); v
(6, 5, 8)
>>> v.parent()
Vector space of dimension 3 over Finite Field of size 11
>>> v = vector([Integer(17), GF(Integer(11))(Integer(5)), Integer(19)], QQ); v
(17, 5, 19)
>>> v.parent()
Vector space of dimension 3 over Rational Field
>>> v = vector((Integer(1),Integer(2),Integer(3)), QQ); v
(1, 2, 3)
>>> v.parent()
Vector space of dimension 3 over Rational Field
>>> v = vector(QQ, (Integer(1),Integer(2),Integer(3))); v
(1, 2, 3)
>>> v.parent()
Vector space of dimension 3 over Rational Field
>>> v = vector(vector([Integer(1),Integer(2),Integer(3)])); v
(1, 2, 3)
>>> v.parent()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

You can also use `free_module_element`, which is the same as `vector`.

```
sage: free_module_element([1/3, -4/5])
(1/3, -4/5)
```

```
>>> from sage.all import *
>>> free_module_element([Integer(1)/Integer(3), -Integer(4)/Integer(5)])
(1/3, -4/5)
```

We make a vector mod 3 out of a vector over $\mathbf{Z}$.

```
sage: vector(vector([1,2,3]), GF(3))
(1, 2, 0)
```

```
>>> from sage.all import *
>>> vector(vector([Integer(1),Integer(2),Integer(3)]), GF(Integer(3)))
```

(continues on next page)

(continued from previous page)

```
(1, 2, 0)
```

The degree of a vector may be specified:

```
sage: vector(QQ, 4, [1,1/2,1/3,1/4])
(1, 1/2, 1/3, 1/4)
```

```
>>> from sage.all import *
>>> vector(QQ, Integer(4), [Integer(1),Integer(1)/Integer(2),Integer(1)/
↪Integer(3),Integer(1)/Integer(4)])
(1, 1/2, 1/3, 1/4)
```

But it is an error if the degree and size of the list of entries are mismatched:

```
sage: vector(QQ, 5, [1,1/2,1/3,1/4])
Traceback (most recent call last):
...
ValueError: incompatible degrees in vector constructor
```

```
>>> from sage.all import *
>>> vector(QQ, Integer(5), [Integer(1),Integer(1)/Integer(2),Integer(1)/
↪Integer(3),Integer(1)/Integer(4)])
Traceback (most recent call last):
...
ValueError: incompatible degrees in vector constructor
```

Providing no entries populates the vector with zeros, but of course, you must specify the degree since it is not implied. Here we use a finite field as the base ring.

```
sage: w = vector(FiniteField(7), 4); w
(0, 0, 0, 0)
sage: w.parent()
Vector space of dimension 4 over Finite Field of size 7
```

```
>>> from sage.all import *
>>> w = vector(FiniteField(Integer(7)), Integer(4)); w
(0, 0, 0, 0)
>>> w.parent()
Vector space of dimension 4 over Finite Field of size 7
```

The fastest method to construct a zero vector is to call the `zero_vector()` method directly on a free module or vector space, since vector(...) must do a small amount of type checking. Almost as fast as the `zero_vector()` method is the *zero_vector()* constructor, which defaults to the integers.

```
sage: vector(ZZ, 5)           # works fine
(0, 0, 0, 0, 0)
sage: (ZZ^5).zero_vector()    # very tiny bit faster
(0, 0, 0, 0, 0)
sage: zero_vector(ZZ, 5)      # similar speed to vector(...)
(0, 0, 0, 0, 0)
sage: z = zero_vector(5); z
(0, 0, 0, 0, 0)
```

(continues on next page)

(continued from previous page)

```
sage: z.parent()
Ambient free module of rank 5 over
the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> vector(ZZ, Integer(5))          # works fine
(0, 0, 0, 0, 0)
>>> (ZZ**Integer(5)).zero_vector()    # very tiny bit faster
(0, 0, 0, 0, 0)
>>> zero_vector(ZZ, Integer(5))     # similar speed to vector(...)
(0, 0, 0, 0, 0)
>>> z = zero_vector(Integer(5)); z
(0, 0, 0, 0, 0)
>>> z.parent()
Ambient free module of rank 5 over
the principal ideal domain Integer Ring
```

Here we illustrate the creation of sparse vectors by using a dictionary:

```
sage: vector({1:1.1, 3:3.14})
(0.000000000000000, 1.10000000000000, 0.000000000000000, 3.14000000000000)
```

```
>>> from sage.all import *
>>> vector({Integer(1):RealNumber('1.1'), Integer(3):RealNumber('3.14')})
(0.000000000000000, 1.10000000000000, 0.000000000000000, 3.14000000000000)
```

With no degree given, a dictionary of entries implicitly declares a degree by the largest index (key) present. So you can provide a terminal element (perhaps a zero?) to set the degree. But it is probably safer to just include a degree in your construction.

```
sage: v = vector(QQ, {0:1/2, 4:-6, 7:0}); v
(1/2, 0, 0, 0, -6, 0, 0, 0)
sage: v.degree()
8
sage: v.is_sparse()
True
sage: w = vector(QQ, 8, {0:1/2, 4:-6})
sage: w == v
True
```

```
>>> from sage.all import *
>>> v = vector(QQ, {Integer(0):Integer(1)/Integer(2), Integer(4):-Integer(6),␣
↪Integer(7):Integer(0)}); v
(1/2, 0, 0, 0, -6, 0, 0, 0)
>>> v.degree()
8
>>> v.is_sparse()
True
>>> w = vector(QQ, Integer(8), {Integer(0):Integer(1)/Integer(2), Integer(4):-
↪Integer(6)})
>>> w == v
True
```

It is an error to specify a negative degree.

```
sage: vector(RR, -4, [1.0, 2.0, 3.0, 4.0])
Traceback (most recent call last):
...
ValueError: cannot specify the degree of a vector as a negative integer (-4)
```

```
>>> from sage.all import *
>>> vector(RR, -Integer(4), [RealNumber('1.0'), RealNumber('2.0'), RealNumber('3.0
↪'), RealNumber('4.0')])
Traceback (most recent call last):
...
ValueError: cannot specify the degree of a vector as a negative integer (-4)
```

It is an error to create a zero vector but not provide a ring as the first argument.

```
sage: vector('junk', 20)
Traceback (most recent call last):
...
TypeError: first argument must be base ring of zero vector, not junk
```

```
>>> from sage.all import *
>>> vector('junk', Integer(20))
Traceback (most recent call last):
...
TypeError: first argument must be base ring of zero vector, not junk
```

And it is an error to specify an index in a dictionary that is greater than or equal to a requested degree.

```
sage: vector(ZZ, 10, {3:4, 7:-2, 10:637})
Traceback (most recent call last):
...
ValueError: dictionary of entries has a key (index) exceeding the requested degree
```

```
>>> from sage.all import *
>>> vector(ZZ, Integer(10), {Integer(3):Integer(4), Integer(7):-Integer(2),␣
↪Integer(10):Integer(637)})
Traceback (most recent call last):
...
ValueError: dictionary of entries has a key (index) exceeding the requested degree
```

A 1-dimensional numpy array of type float or complex may be passed to vector. Unless an explicit ring is given, the result will be a vector in the appropriate dimensional vector space over the real double field or the complex double field. The data in the array must be contiguous, so column-wise slices of numpy matrices will raise an exception.

```
sage: # needs numpy
sage: import numpy
sage: x = numpy.random.randn(10)
sage: y = vector(x)
sage: parent(y)
Vector space of dimension 10 over Real Double Field
sage: parent(vector(RDF, x))
Vector space of dimension 10 over Real Double Field
```

(continues on next page)

(continued from previous page)

```
sage: parent(vector(CDF, x))
Vector space of dimension 10 over Complex Double Field
sage: parent(vector(RR, x))
Vector space of dimension 10 over Real Field with 53 bits of precision
sage: v = numpy.random.randn(10) * complex(0,1)
sage: w = vector(v)
sage: parent(w)
Vector space of dimension 10 over Complex Double Field
```

```
>>> from sage.all import *
>>> # needs numpy
>>> import numpy
>>> x = numpy.random.randn(Integer(10))
>>> y = vector(x)
>>> parent(y)
Vector space of dimension 10 over Real Double Field
>>> parent(vector(RDF, x))
Vector space of dimension 10 over Real Double Field
>>> parent(vector(CDF, x))
Vector space of dimension 10 over Complex Double Field
>>> parent(vector(RR, x))
Vector space of dimension 10 over Real Field with 53 bits of precision
>>> v = numpy.random.randn(Integer(10)) * complex(Integer(0),Integer(1))
>>> w = vector(v)
>>> parent(w)
Vector space of dimension 10 over Complex Double Field
```

Multi-dimensional arrays are not supported:

```
sage: # needs numpy
sage: import numpy as np
sage: a = np.array([[1, 2, 3], [4, 5, 6]], np.float64)
sage: vector(a)
Traceback (most recent call last):
...
TypeError: cannot convert 2-dimensional array to a vector
```

```
>>> from sage.all import *
>>> # needs numpy
>>> import numpy as np
>>> a = np.array([[Integer(1), Integer(2), Integer(3)], [Integer(4), Integer(5),␣
↪Integer(6)]], np.float64)
>>> vector(a)
Traceback (most recent call last):
...
TypeError: cannot convert 2-dimensional array to a vector
```

If any of the arguments to vector have Python type int, real, or complex, they will first be coerced to the appropriate Sage objects. This fixes Issue #3847.

```
sage: v = vector([int(0)]); v
(0)
```

(continues on next page)

(continued from previous page)

```
sage: v[0].parent()
Integer Ring
sage: v = vector(range(10)); v
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
sage: v[3].parent()
Integer Ring
sage: v = vector([float(23.4), int(2), complex(2+7*I), 1]); v                    #␣
↪needs sage.symbolic
(23.4, 2.0, 2.0 + 7.0*I, 1.0)
sage: v[1].parent()                                                              #␣
↪needs sage.symbolic
Complex Double Field
```

```
>>> from sage.all import *
>>> v = vector([int(Integer(0))]); v
(0)
>>> v[Integer(0)].parent()
Integer Ring
>>> v = vector(range(Integer(10))); v
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> v[Integer(3)].parent()
Integer Ring
>>> v = vector([float(RealNumber('23.4')), int(Integer(2)),␣
↪complex(Integer(2)+Integer(7)*I), Integer(1)]); v                    # needs␣
↪sage.symbolic
(23.4, 2.0, 2.0 + 7.0*I, 1.0)
>>> v[Integer(1)].parent()                                                      ␣
↪     # needs sage.symbolic
Complex Double Field
```

If the argument is a vector, it doesn't change the base ring. This fixes Issue #6643:

```
sage: # needs sage.rings.number_field
sage: K.<sqrt3> = QuadraticField(3)
sage: u = vector(K, (1/2, sqrt3/2))
sage: vector(u).base_ring()
Number Field in sqrt3 with defining polynomial x^2 - 3 with sqrt3 = 1.
↪732050807568878?
sage: v = vector(K, (0, 1))
sage: vector(v).base_ring()
Number Field in sqrt3 with defining polynomial x^2 - 3 with sqrt3 = 1.
↪732050807568878?
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> K = QuadraticField(Integer(3), names=('sqrt3',)); (sqrt3,) = K._first_ngens(1)
>>> u = vector(K, (Integer(1)/Integer(2), sqrt3/Integer(2)))
>>> vector(u).base_ring()
Number Field in sqrt3 with defining polynomial x^2 - 3 with sqrt3 = 1.
↪732050807568878?
>>> v = vector(K, (Integer(0), Integer(1)))
>>> vector(v).base_ring()
```

(continues on next page)

(continued from previous page)

```
Number Field in sqrt3 with defining polynomial x^2 - 3 with sqrt3 = 1.
↪732050807568878?
```

Constructing a vector from a numpy array behaves as expected:

```
sage: # needs numpy
sage: import numpy
sage: a = numpy.array([1,2,3])
sage: v = vector(a); v
(1, 2, 3)
sage: parent(v)
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> # needs numpy
>>> import numpy
>>> a = numpy.array([Integer(1),Integer(2),Integer(3)])
>>> v = vector(a); v
(1, 2, 3)
>>> parent(v)
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

Complex numbers can be converted naturally to a sequence of length 2. And then to a vector.

```
sage: c = CDF(2 + 3*I)                                                    #␣
↪needs sage.rings.complex_double sage.symbolic
sage: v = vector(c); v                                                    #␣
↪needs sage.rings.complex_double sage.symbolic
(2.0, 3.0)
```

```
>>> from sage.all import *
>>> c = CDF(Integer(2) + Integer(3)*I)                                     ␣
↪              # needs sage.rings.complex_double sage.symbolic
>>> v = vector(c); v                                                     #␣
↪needs sage.rings.complex_double sage.symbolic
(2.0, 3.0)
```

A generator, or other iterable, may also be supplied as input. Anything that can be converted to a `Sequence` is a possible input.

```
sage: type(i^2 for i in range(3))
<... 'generator'>
sage: v = vector(i^2 for i in range(3)); v
(0, 1, 4)
```

```
>>> from sage.all import *
>>> type(i**Integer(2) for i in range(Integer(3)))
<... 'generator'>
>>> v = vector(i**Integer(2) for i in range(Integer(3))); v
(0, 1, 4)
```

An empty list, without a ring given, will default to the integers.

```
sage: x = vector([]); x
()
sage: x.parent()
Ambient free module of rank 0 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> x = vector([]); x
()
>>> x.parent()
Ambient free module of rank 0 over the principal ideal domain Integer Ring
```

The `immutable` switch allows to create an immutable vector.

```
sage: v = vector(QQ, {0:1/2, 4:-6, 7:0}, immutable=True); v
(1/2, 0, 0, 0, -6, 0, 0, 0)
sage: v.is_immutable()
True
```

```
>>> from sage.all import *
>>> v = vector(QQ, {Integer(0):Integer(1)/Integer(2), Integer(4):-Integer(6),␣
↪Integer(7):Integer(0)}, immutable=True); v
(1/2, 0, 0, 0, -6, 0, 0, 0)
>>> v.is_immutable()
True
```

The `immutable` switch works regardless of the type of valid input to the constructor.

```
sage: v = vector(ZZ, 4, immutable=True)
sage: v.is_immutable()
True
sage: w = vector(ZZ, [1,2,3])
sage: v = vector(w, ZZ, immutable=True)
sage: v.is_immutable()
True
sage: v = vector(QQ, w, immutable=True)
sage: v.is_immutable()
True

sage: # needs numpy sage.symbolic
sage: import numpy as np
sage: w = np.array([1, 2, pi], float)
sage: v = vector(w, immutable=True)
sage: v.is_immutable()
True
sage: w = np.array([i, 2, 3], complex)
sage: v = vector(w, immutable=True)
sage: v.is_immutable()
True
```

```
>>> from sage.all import *
>>> v = vector(ZZ, Integer(4), immutable=True)
>>> v.is_immutable()
```

(continues on next page)

(continued from previous page)

```
True
>>> w = vector(ZZ, [Integer(1),Integer(2),Integer(3)])
>>> v = vector(w, ZZ, immutable=True)
>>> v.is_immutable()
True
>>> v = vector(QQ, w, immutable=True)
>>> v.is_immutable()
True

>>> # needs numpy sage.symbolic
>>> import numpy as np
>>> w = np.array([Integer(1), Integer(2), pi], float)
>>> v = vector(w, immutable=True)
>>> v.is_immutable()
True
>>> w = np.array([i, Integer(2), Integer(3)], complex)
>>> v = vector(w, immutable=True)
>>> v.is_immutable()
True
```

sage.modules.free_module_element.**is_FreeModuleElement**(*x*)

EXAMPLES:

```
sage: sage.modules.free_module_element.is_FreeModuleElement(0)
doctest:warning...
DeprecationWarning: The function is_FreeModuleElement is deprecated;
use 'isinstance(..., FreeModuleElement)' instead.
See https://github.com/sagemath/sage/issues/38184 for details.
False
sage: sage.modules.free_module_element.is_FreeModuleElement(vector([1,2,3]))
True
```

```
>>> from sage.all import *
>>> sage.modules.free_module_element.is_FreeModuleElement(Integer(0))
doctest:warning...
DeprecationWarning: The function is_FreeModuleElement is deprecated;
use 'isinstance(..., FreeModuleElement)' instead.
See https://github.com/sagemath/sage/issues/38184 for details.
False
>>> sage.modules.free_module_element.is_FreeModuleElement(vector([Integer(1),
↪Integer(2),Integer(3)]))
True
```

sage.modules.free_module_element.**make_FreeModuleElement_generic_dense**(*parent*, *entries*, *degree*)

EXAMPLES:

```
sage: sage.modules.free_module_element.make_FreeModuleElement_generic_dense(QQ^3,␣
↪[1,2,-3/7], 3)
(1, 2, -3/7)
```

```
>>> from sage.all import *
>>> sage.modules.free_module_element.make_FreeModuleElement_generic_
```

(continues on next page)

(continued from previous page)

```
↪dense(QQ**Integer(3), [Integer(1),Integer(2),-Integer(3)/Integer(7)],␣
↪Integer(3))
(1, 2, -3/7)
```

sage.modules.free_module_element.**make_FreeModuleElement_generic_dense_v1**(*parent*, *entries*, *degree*, *is_mutable*)

EXAMPLES:

```
sage: v = sage.modules.free_module_element.make_FreeModuleElement_generic_dense_
↪v1(QQ^3, [1,2,-3/7], 3, True); v
(1, 2, -3/7)
sage: v[0] = 10; v
(10, 2, -3/7)
sage: v = sage.modules.free_module_element.make_FreeModuleElement_generic_dense_
↪v1(QQ^3, [1,2,-3/7], 3, False); v
(1, 2, -3/7)
sage: v[0] = 10
Traceback (most recent call last):
...
ValueError: vector is immutable; please change a copy instead (use copy())
```

```
>>> from sage.all import *
>>> v = sage.modules.free_module_element.make_FreeModuleElement_generic_dense_
↪v1(QQ**Integer(3), [Integer(1),Integer(2),-Integer(3)/Integer(7)], Integer(3),␣
↪True); v
(1, 2, -3/7)
>>> v[Integer(0)] = Integer(10); v
(10, 2, -3/7)
>>> v = sage.modules.free_module_element.make_FreeModuleElement_generic_dense_
↪v1(QQ**Integer(3), [Integer(1),Integer(2),-Integer(3)/Integer(7)], Integer(3),␣
↪False); v
(1, 2, -3/7)
>>> v[Integer(0)] = Integer(10)
Traceback (most recent call last):
...
ValueError: vector is immutable; please change a copy instead (use copy())
```

sage.modules.free_module_element.**make_FreeModuleElement_generic_sparse**(*parent*, *entries*, *degree*)

EXAMPLES:

```
sage: v = sage.modules.free_module_element.make_FreeModuleElement_generic_
↪sparse(QQ^3, {2:5/2}, 3); v
(0, 0, 5/2)
```

```
>>> from sage.all import *
>>> v = sage.modules.free_module_element.make_FreeModuleElement_generic_
↪sparse(QQ**Integer(3), {Integer(2):Integer(5)/Integer(2)}, Integer(3)); v
(0, 0, 5/2)
```

sage.modules.free_module_element.**make_FreeModuleElement_generic_sparse_v1**(*parent*, *entries*, *degree*, *is_mutable*)

EXAMPLES:

```
sage: v = sage.modules.free_module_element.make_FreeModuleElement_generic_sparse_
↪v1(QQ^3, {2:5/2}, 3, False); v
(0, 0, 5/2)
sage: v.is_mutable()
False
```

```
>>> from sage.all import *
>>> v = sage.modules.free_module_element.make_FreeModuleElement_generic_sparse_
↪v1(QQ**Integer(3), {Integer(2):Integer(5)/Integer(2)}, Integer(3), False); v
(0, 0, 5/2)
>>> v.is_mutable()
False
```

sage.modules.free_module_element.**prepare**(*v*, *R*, *degree=None*)

Convert an object describing elements of a vector into a list of entries in a common ring.

INPUT:

- `v` – dictionary with nonnegative integers as keys, or a list or other object that can be converted by the `Sequence` constructor
- `R` – a ring containing all the entries, possibly given as `None`
- `degree` – a requested size for the list when the input is a dictionary, otherwise ignored

OUTPUT: a pair

The first item is a list of the values specified in the object `v`. If the object is a dictionary , entries are placed in the list according to the indices that were their keys in the dictionary, and the remainder of the entries are zero. The value of `degree` is assumed to be larger than any index provided in the dictionary and will be used as the number of entries in the returned list.

The second item returned is a ring that contains all of the entries in the list. If `R` is given, the entries are coerced in. Otherwise a common ring is found. For more details, see the `Sequence` object. When `v` has no elements and `R` is `None`, the ring returned is the integers.

EXAMPLES:

```
sage: from sage.modules.free_module_element import prepare
sage: prepare([1, 2/3, 5], None)
([1, 2/3, 5], Rational Field)

sage: prepare([1, 2/3, 5], RR)
([1.00000000000000, 0.666666666666667, 5.00000000000000],
 Real Field with 53 bits of precision)

sage: prepare({1: 4, 3: -2}, ZZ, 6)
([0, 4, 0, -2, 0, 0], Integer Ring)

sage: prepare({3: 1, 5: 3}, QQ, 6)
([0, 0, 0, 1, 0, 3], Rational Field)
```

(continues on next page)

(continued from previous page)

```
sage: prepare([1, 2/3, '10', 5], RR)
([1.00000000000000, 0.666666666666667, 10.0000000000000, 5.00000000000000],
 Real Field with 53 bits of precision)

sage: prepare({}, QQ, 0)
([], Rational Field)

sage: prepare([1, 2/3, '10', 5], None)
Traceback (most recent call last):
...
TypeError: unable to find a common ring for all elements
```

```
>>> from sage.all import *
>>> from sage.modules.free_module_element import prepare
>>> prepare([Integer(1), Integer(2)/Integer(3), Integer(5)], None)
([1, 2/3, 5], Rational Field)

>>> prepare([Integer(1), Integer(2)/Integer(3), Integer(5)], RR)
([1.00000000000000, 0.666666666666667, 5.00000000000000],
 Real Field with 53 bits of precision)

>>> prepare({Integer(1): Integer(4), Integer(3): -Integer(2)}, ZZ, Integer(6))
([0, 4, 0, -2, 0, 0], Integer Ring)

>>> prepare({Integer(3): Integer(1), Integer(5): Integer(3)}, QQ, Integer(6))
([0, 0, 0, 1, 0, 3], Rational Field)

>>> prepare([Integer(1), Integer(2)/Integer(3), '10', Integer(5)], RR)
([1.00000000000000, 0.666666666666667, 10.0000000000000, 5.00000000000000],
 Real Field with 53 bits of precision)

>>> prepare({}, QQ, Integer(0))
([], Rational Field)

>>> prepare([Integer(1), Integer(2)/Integer(3), '10', Integer(5)], None)
Traceback (most recent call last):
...
TypeError: unable to find a common ring for all elements
```

Some objects can be converted to sequences even if they are not always thought of as sequences.

```
sage: c = CDF(2 + 3*I)                                                          #␣
↪needs sage.symbolic
sage: prepare(c, None)                                                          #␣
↪needs sage.symbolic
([2.0, 3.0], Real Double Field)
```

```
>>> from sage.all import *
>>> c = CDF(Integer(2) + Integer(3)*I)                                           ␣
↪              # needs sage.symbolic
>>> prepare(c, None)                                                           #␣
↪needs sage.symbolic
```

(continues on next page)

(continued from previous page)

```
([2.0, 3.0], Real Double Field)
```

This checks a bug listed at Issue #10595. Without good evidence for a ring, the default is the integers.

```
sage: prepare([], None)
([], Integer Ring)
```

```
>>> from sage.all import *
>>> prepare([], None)
([], Integer Ring)
```

sage.modules.free_module_element.**random_vector**(*ring*, *degree=None*, **args*, ***kwds*)

Return a vector (or module element) with random entries.

INPUT:

- `ring` – (default: `ZZ`) the base ring for the entries
- `degree` – nonnegative integer for the number of entries in the vector
- `sparse` – (default: `False`) whether to use a sparse implementation
- `*args`, `**kwds` – additional arguments and keywords are passed to the `random_element()` method of the ring

OUTPUT:

A vector, or free module element, with `degree` elements from `ring`, chosen randomly from the ring according to the ring's `random_element()` method.

Note

See below for examples of how random elements are generated by some common base rings.

EXAMPLES:

First, module elements over the integers. The default distribution is tightly clustered around -1, 0, 1. Uniform distributions can be specified by giving bounds, though the upper bound is never met. See `sage.rings.integer_ring.IntegerRing_class.random_element()` for several other variants.

```
sage: random_vector(10).parent()
Ambient free module of rank 10 over the principal ideal domain Integer Ring
sage: random_vector(20).parent()
Ambient free module of rank 20 over the principal ideal domain Integer Ring

sage: v = random_vector(ZZ, 20, x=4)
sage: all(i in range(4) for i in v)
True

sage: v = random_vector(ZZ, 20, x=-20, y=100)
sage: all(i in range(-20, 100) for i in v)
True
```

```
>>> from sage.all import *
>>> random_vector(Integer(10)).parent()
Ambient free module of rank 10 over the principal ideal domain Integer Ring
>>> random_vector(Integer(20)).parent()
Ambient free module of rank 20 over the principal ideal domain Integer Ring

>>> v = random_vector(ZZ, Integer(20), x=Integer(4))
>>> all(i in range(Integer(4)) for i in v)
True

>>> v = random_vector(ZZ, Integer(20), x=-Integer(20), y=Integer(100))
>>> all(i in range(-Integer(20), Integer(100)) for i in v)
True
```

If the ring is not specified, the default is the integers, and parameters for the random distribution may be passed without using keywords. This is a random vector with 20 entries uniformly distributed between -20 and 100.

```
sage: random_vector(20, -20, 100).parent()
Ambient free module of rank 20 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> random_vector(Integer(20), -Integer(20), Integer(100)).parent()
Ambient free module of rank 20 over the principal ideal domain Integer Ring
```

Now over the rationals. Note that bounds on the numerator and denominator may be specified. See `sage.rings.rational_field.RationalField.random_element()` for documentation.

```
sage: random_vector(QQ, 10).parent()
Vector space of dimension 10 over Rational Field

sage: v = random_vector(QQ, 10, num_bound=15, den_bound=5)
sage: v.parent()
Vector space of dimension 10 over Rational Field
sage: all(q.numerator() <= 15 and q.denominator() <= 5 for q in v)
True
```

```
>>> from sage.all import *
>>> random_vector(QQ, Integer(10)).parent()
Vector space of dimension 10 over Rational Field

>>> v = random_vector(QQ, Integer(10), num_bound=Integer(15), den_
↪bound=Integer(5))
>>> v.parent()
Vector space of dimension 10 over Rational Field
>>> all(q.numerator() <= Integer(15) and q.denominator() <= Integer(5) for q in v)
True
```

Inexact rings may be used as well. The reals have uniform distributions, with the range $(-1, 1)$ as the default. More at: `sage.rings.real_mpfr.RealField_class.random_element()`

```
sage: v = random_vector(RR, 5)
sage: v.parent()
Vector space of dimension 5 over Real Field with 53 bits of precision
```

(continues on next page)

(continued from previous page)

```
sage: all(-1 <= r <= 1 for r in v)
True

sage: v = random_vector(RR, 5, min=8, max=14)
sage: v.parent()
Vector space of dimension 5 over Real Field with 53 bits of precision
sage: all(8 <= r <= 14 for r in v)
True
```

```
>>> from sage.all import *
>>> v = random_vector(RR, Integer(5))
>>> v.parent()
Vector space of dimension 5 over Real Field with 53 bits of precision
>>> all(-Integer(1) <= r <= Integer(1) for r in v)
True

>>> v = random_vector(RR, Integer(5), min=Integer(8), max=Integer(14))
>>> v.parent()
Vector space of dimension 5 over Real Field with 53 bits of precision
>>> all(Integer(8) <= r <= Integer(14) for r in v)
True
```

Any ring with a `random_element()` method may be used.

```
sage: F = FiniteField(23)
sage: hasattr(F, 'random_element')
True
sage: v = random_vector(F, 10)
sage: v.parent()
Vector space of dimension 10 over Finite Field of size 23
```

```
>>> from sage.all import *
>>> F = FiniteField(Integer(23))
>>> hasattr(F, 'random_element')
True
>>> v = random_vector(F, Integer(10))
>>> v.parent()
Vector space of dimension 10 over Finite Field of size 23
```

The default implementation is a dense representation, equivalent to setting `sparse=False`.

```
sage: v = random_vector(10)
sage: v.is_sparse()
False

sage: w = random_vector(ZZ, 20, sparse=True)
sage: w.is_sparse()
True
```

```
>>> from sage.all import *
>>> v = random_vector(Integer(10))
>>> v.is_sparse()
```

(continues on next page)

(continued from previous page)

```
False

>>> w = random_vector(ZZ, Integer(20), sparse=True)
>>> w.is_sparse()
True
```

The elements are chosen using the ring's `random_element` method:

```
sage: from sage.misc.randstate import current_randstate
sage: seed = current_randstate().seed()
sage: set_random_seed(seed)
sage: v1 = random_vector(ZZ, 20, distribution='1/n')
sage: v2 = random_vector(ZZ, 15, x=-1000, y=1000)
sage: v3 = random_vector(QQ, 10)
sage: v4 = random_vector(FiniteField(17), 10)
sage: v5 = random_vector(RR, 10)
sage: set_random_seed(seed)
sage: w1 = vector(ZZ.random_element(distribution='1/n') for _ in range(20))
sage: w2 = vector(ZZ.random_element(x=-1000, y=1000) for _ in range(15))
sage: w3 = vector(QQ.random_element() for _ in range(10))
sage: [v1, v2, v3] == [w1, w2, w3]
True
sage: w4 = vector(FiniteField(17).random_element() for _ in range(10))
sage: v4 == w4
True
sage: w5 = vector(RR.random_element() for _ in range(10))
sage: v5 == w5
True
```

```
>>> from sage.all import *
>>> from sage.misc.randstate import current_randstate
>>> seed = current_randstate().seed()
>>> set_random_seed(seed)
>>> v1 = random_vector(ZZ, Integer(20), distribution='1/n')
>>> v2 = random_vector(ZZ, Integer(15), x=-Integer(1000), y=Integer(1000))
>>> v3 = random_vector(QQ, Integer(10))
>>> v4 = random_vector(FiniteField(Integer(17)), Integer(10))
>>> v5 = random_vector(RR, Integer(10))
>>> set_random_seed(seed)
>>> w1 = vector(ZZ.random_element(distribution='1/n') for _ in range(Integer(20)))
>>> w2 = vector(ZZ.random_element(x=-Integer(1000), y=Integer(1000)) for _ in␣
↪range(Integer(15)))
>>> w3 = vector(QQ.random_element() for _ in range(Integer(10)))
>>> [v1, v2, v3] == [w1, w2, w3]
True
>>> w4 = vector(FiniteField(Integer(17)).random_element() for _ in␣
↪range(Integer(10)))
>>> v4 == w4
True
>>> w5 = vector(RR.random_element() for _ in range(Integer(10)))
>>> v5 == w5
True
```

Inputs get checked before constructing the vector.

```
sage: random_vector('junk')
Traceback (most recent call last):
...
TypeError: degree of a random vector must be an integer, not None

sage: random_vector('stuff', 5)
Traceback (most recent call last):
...
TypeError: elements of a vector, or module element, must come from a ring, not␣
↪stuff

sage: random_vector(ZZ, -9)
Traceback (most recent call last):
...
ValueError: degree of a random vector must be nonnegative, not -9
```

```
>>> from sage.all import *
>>> random_vector('junk')
Traceback (most recent call last):
...
TypeError: degree of a random vector must be an integer, not None

>>> random_vector('stuff', Integer(5))
Traceback (most recent call last):
...
TypeError: elements of a vector, or module element, must come from a ring, not␣
↪stuff

>>> random_vector(ZZ, -Integer(9))
Traceback (most recent call last):
...
ValueError: degree of a random vector must be nonnegative, not -9
```

sage.modules.free_module_element.**vector**(*arg0*, *arg1=None*, *arg2=None*, *sparse=None*, *immutable=False*)

Return a vector or free module element with specified entries.

CALL FORMATS:

This constructor can be called in several different ways. In each case, `sparse=True` or `sparse=False` as well as `immutable=True` or `immutable=False` can be supplied as an option. `free_module_element()` is an alias for `vector()`.

1. vector(object)
2. vector(ring, object)
3. vector(object, ring)
4. vector(ring, degree, object)
5. vector(ring, degree)

INPUT:

- `object` – list, dictionary, or other iterable containing the entries of the vector, including any object that is palatable to the `Sequence` constructor

- `ring` – a base ring (or field) for the vector space or free module, which contains all of the elements
- `degree` – integer specifying the number of entries in the vector or free module element
- `sparse` – boolean, whether the result should be a sparse vector
- `immutable` – boolean (default: `False`); whether the result should be an immutable vector

In call format 4, an error is raised if the `degree` does not match the length of `object` so this call can provide some safeguards. Note however that using this format when `object` is a dictionary is unlikely to work properly.

OUTPUT:

An element of the ambient vector space or free module with the given base ring and implied or specified dimension or rank, containing the specified entries and with correct degree.

In call format 5, no entries are specified, so the element is populated with all zeros.

If the `sparse` option is not supplied, the output will generally have a dense representation. The exception is if `object` is a dictionary, then the representation will be sparse.

EXAMPLES:

```
sage: v = vector([1,2,3]); v
(1, 2, 3)
sage: v.parent()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
sage: v = vector([1,2,3/5]); v
(1, 2, 3/5)
sage: v.parent()
Vector space of dimension 3 over Rational Field
```

```
>>> from sage.all import *
>>> v = vector([Integer(1),Integer(2),Integer(3)]); v
(1, 2, 3)
>>> v.parent()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
>>> v = vector([Integer(1),Integer(2),Integer(3)/Integer(5)]); v
(1, 2, 3/5)
>>> v.parent()
Vector space of dimension 3 over Rational Field
```

All entries must *canonically* coerce to some common ring:

```
sage: v = vector([17, GF(11)(5), 19/3]); v
Traceback (most recent call last):
...
TypeError: unable to find a common ring for all elements
```

```
>>> from sage.all import *
>>> v = vector([Integer(17), GF(Integer(11))(Integer(5)), Integer(19)/
↪Integer(3)]); v
Traceback (most recent call last):
...
TypeError: unable to find a common ring for all elements
```

```
sage: v = vector([17, GF(11)(5), 19]); v
(6, 5, 8)
sage: v.parent()
Vector space of dimension 3 over Finite Field of size 11
sage: v = vector([17, GF(11)(5), 19], QQ); v
(17, 5, 19)
sage: v.parent()
Vector space of dimension 3 over Rational Field
sage: v = vector((1,2,3), QQ); v
(1, 2, 3)
sage: v.parent()
Vector space of dimension 3 over Rational Field
sage: v = vector(QQ, (1,2,3)); v
(1, 2, 3)
sage: v.parent()
Vector space of dimension 3 over Rational Field
sage: v = vector(vector([1,2,3])); v
(1, 2, 3)
sage: v.parent()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> v = vector([Integer(17), GF(Integer(11))(Integer(5)), Integer(19)]); v
(6, 5, 8)
>>> v.parent()
Vector space of dimension 3 over Finite Field of size 11
>>> v = vector([Integer(17), GF(Integer(11))(Integer(5)), Integer(19)], QQ); v
(17, 5, 19)
>>> v.parent()
Vector space of dimension 3 over Rational Field
>>> v = vector((Integer(1),Integer(2),Integer(3)), QQ); v
(1, 2, 3)
>>> v.parent()
Vector space of dimension 3 over Rational Field
>>> v = vector(QQ, (Integer(1),Integer(2),Integer(3))); v
(1, 2, 3)
>>> v.parent()
Vector space of dimension 3 over Rational Field
>>> v = vector(vector([Integer(1),Integer(2),Integer(3)])); v
(1, 2, 3)
>>> v.parent()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

You can also use `free_module_element`, which is the same as `vector`.

```
sage: free_module_element([1/3, -4/5])
(1/3, -4/5)
```

```
>>> from sage.all import *
>>> free_module_element([Integer(1)/Integer(3), -Integer(4)/Integer(5)])
(1/3, -4/5)
```

We make a vector mod 3 out of a vector over $\mathbf{Z}$.

```
sage: vector(vector([1,2,3]), GF(3))
(1, 2, 0)
```

```
>>> from sage.all import *
>>> vector(vector([Integer(1),Integer(2),Integer(3)]), GF(Integer(3)))
(1, 2, 0)
```

The degree of a vector may be specified:

```
sage: vector(QQ, 4, [1,1/2,1/3,1/4])
(1, 1/2, 1/3, 1/4)
```

```
>>> from sage.all import *
>>> vector(QQ, Integer(4), [Integer(1),Integer(1)/Integer(2),Integer(1)/
↪Integer(3),Integer(1)/Integer(4)])
(1, 1/2, 1/3, 1/4)
```

But it is an error if the degree and size of the list of entries are mismatched:

```
sage: vector(QQ, 5, [1,1/2,1/3,1/4])
Traceback (most recent call last):
...
ValueError: incompatible degrees in vector constructor
```

```
>>> from sage.all import *
>>> vector(QQ, Integer(5), [Integer(1),Integer(1)/Integer(2),Integer(1)/
↪Integer(3),Integer(1)/Integer(4)])
Traceback (most recent call last):
...
ValueError: incompatible degrees in vector constructor
```

Providing no entries populates the vector with zeros, but of course, you must specify the degree since it is not implied. Here we use a finite field as the base ring.

```
sage: w = vector(FiniteField(7), 4); w
(0, 0, 0, 0)
sage: w.parent()
Vector space of dimension 4 over Finite Field of size 7
```

```
>>> from sage.all import *
>>> w = vector(FiniteField(Integer(7)), Integer(4)); w
(0, 0, 0, 0)
>>> w.parent()
Vector space of dimension 4 over Finite Field of size 7
```

The fastest method to construct a zero vector is to call the `zero_vector()` method directly on a free module or vector space, since vector(...) must do a small amount of type checking. Almost as fast as the `zero_vector()` method is the *zero_vector()* constructor, which defaults to the integers.

```
sage: vector(ZZ, 5)          # works fine
(0, 0, 0, 0, 0)
sage: (ZZ^5).zero_vector()   # very tiny bit faster
```

(continues on next page)

(continued from previous page)

```
(0, 0, 0, 0, 0)
sage: zero_vector(ZZ, 5)      # similar speed to vector(...)
(0, 0, 0, 0, 0)
sage: z = zero_vector(5); z
(0, 0, 0, 0, 0)
sage: z.parent()
Ambient free module of rank 5 over
the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> vector(ZZ, Integer(5))           # works fine
(0, 0, 0, 0, 0)
>>> (ZZ**Integer(5)).zero_vector()    # very tiny bit faster
(0, 0, 0, 0, 0)
>>> zero_vector(ZZ, Integer(5))      # similar speed to vector(...)
(0, 0, 0, 0, 0)
>>> z = zero_vector(Integer(5)); z
(0, 0, 0, 0, 0)
>>> z.parent()
Ambient free module of rank 5 over
the principal ideal domain Integer Ring
```

Here we illustrate the creation of sparse vectors by using a dictionary:

```
sage: vector({1:1.1, 3:3.14})
(0.000000000000000, 1.10000000000000, 0.000000000000000, 3.14000000000000)
```

```
>>> from sage.all import *
>>> vector({Integer(1):RealNumber('1.1'), Integer(3):RealNumber('3.14')})
(0.000000000000000, 1.10000000000000, 0.000000000000000, 3.14000000000000)
```

With no degree given, a dictionary of entries implicitly declares a degree by the largest index (key) present. So you can provide a terminal element (perhaps a zero?) to set the degree. But it is probably safer to just include a degree in your construction.

```
sage: v = vector(QQ, {0:1/2, 4:-6, 7:0}); v
(1/2, 0, 0, 0, -6, 0, 0, 0)
sage: v.degree()
8
sage: v.is_sparse()
True
sage: w = vector(QQ, 8, {0:1/2, 4:-6})
sage: w == v
True
```

```
>>> from sage.all import *
>>> v = vector(QQ, {Integer(0):Integer(1)/Integer(2), Integer(4):-Integer(6),␣
↪Integer(7):Integer(0)}); v
(1/2, 0, 0, 0, -6, 0, 0, 0)
>>> v.degree()
8
>>> v.is_sparse()
```

(continues on next page)

(continued from previous page)

```
True
>>> w = vector(QQ, Integer(8), {Integer(0):Integer(1)/Integer(2), Integer(4):-
↪Integer(6)})
>>> w == v
True
```

It is an error to specify a negative degree.

```
sage: vector(RR, -4, [1.0, 2.0, 3.0, 4.0])
Traceback (most recent call last):
...
ValueError: cannot specify the degree of a vector as a negative integer (-4)
```

```
>>> from sage.all import *
>>> vector(RR, -Integer(4), [RealNumber('1.0'), RealNumber('2.0'), RealNumber('3.0
↪'), RealNumber('4.0')])
Traceback (most recent call last):
...
ValueError: cannot specify the degree of a vector as a negative integer (-4)
```

It is an error to create a zero vector but not provide a ring as the first argument.

```
sage: vector('junk', 20)
Traceback (most recent call last):
...
TypeError: first argument must be base ring of zero vector, not junk
```

```
>>> from sage.all import *
>>> vector('junk', Integer(20))
Traceback (most recent call last):
...
TypeError: first argument must be base ring of zero vector, not junk
```

And it is an error to specify an index in a dictionary that is greater than or equal to a requested degree.

```
sage: vector(ZZ, 10, {3:4, 7:-2, 10:637})
Traceback (most recent call last):
...
ValueError: dictionary of entries has a key (index) exceeding the requested degree
```

```
>>> from sage.all import *
>>> vector(ZZ, Integer(10), {Integer(3):Integer(4), Integer(7):-Integer(2),␣
↪Integer(10):Integer(637)})
Traceback (most recent call last):
...
ValueError: dictionary of entries has a key (index) exceeding the requested degree
```

A 1-dimensional numpy array of type float or complex may be passed to vector. Unless an explicit ring is given, the result will be a vector in the appropriate dimensional vector space over the real double field or the complex double field. The data in the array must be contiguous, so column-wise slices of numpy matrices will raise an exception.

```
sage: # needs numpy
sage: import numpy
sage: x = numpy.random.randn(10)
sage: y = vector(x)
sage: parent(y)
Vector space of dimension 10 over Real Double Field
sage: parent(vector(RDF, x))
Vector space of dimension 10 over Real Double Field
sage: parent(vector(CDF, x))
Vector space of dimension 10 over Complex Double Field
sage: parent(vector(RR, x))
Vector space of dimension 10 over Real Field with 53 bits of precision
sage: v = numpy.random.randn(10) * complex(0,1)
sage: w = vector(v)
sage: parent(w)
Vector space of dimension 10 over Complex Double Field
```

```
>>> from sage.all import *
>>> # needs numpy
>>> import numpy
>>> x = numpy.random.randn(Integer(10))
>>> y = vector(x)
>>> parent(y)
Vector space of dimension 10 over Real Double Field
>>> parent(vector(RDF, x))
Vector space of dimension 10 over Real Double Field
>>> parent(vector(CDF, x))
Vector space of dimension 10 over Complex Double Field
>>> parent(vector(RR, x))
Vector space of dimension 10 over Real Field with 53 bits of precision
>>> v = numpy.random.randn(Integer(10)) * complex(Integer(0),Integer(1))
>>> w = vector(v)
>>> parent(w)
Vector space of dimension 10 over Complex Double Field
```

Multi-dimensional arrays are not supported:

```
sage: # needs numpy
sage: import numpy as np
sage: a = np.array([[1, 2, 3], [4, 5, 6]], np.float64)
sage: vector(a)
Traceback (most recent call last):
...
TypeError: cannot convert 2-dimensional array to a vector
```

```
>>> from sage.all import *
>>> # needs numpy
>>> import numpy as np
>>> a = np.array([[Integer(1), Integer(2), Integer(3)], [Integer(4), Integer(5),␣
↪Integer(6)]], np.float64)
>>> vector(a)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
TypeError: cannot convert 2-dimensional array to a vector
```

If any of the arguments to vector have Python type int, real, or complex, they will first be coerced to the appropriate Sage objects. This fixes Issue #3847.

```
sage: v = vector([int(0)]); v
(0)
sage: v[0].parent()
Integer Ring
sage: v = vector(range(10)); v
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
sage: v[3].parent()
Integer Ring
sage: v = vector([float(23.4), int(2), complex(2+7*I), 1]); v                    #␣
↪needs sage.symbolic
(23.4, 2.0, 2.0 + 7.0*I, 1.0)
sage: v[1].parent()                                                              #␣
↪needs sage.symbolic
Complex Double Field
```

```
>>> from sage.all import *
>>> v = vector([int(Integer(0))]); v
(0)
>>> v[Integer(0)].parent()
Integer Ring
>>> v = vector(range(Integer(10))); v
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> v[Integer(3)].parent()
Integer Ring
>>> v = vector([float(RealNumber('23.4')), int(Integer(2)),␣
↪complex(Integer(2)+Integer(7)*I), Integer(1)]); v                    # needs␣
↪sage.symbolic
(23.4, 2.0, 2.0 + 7.0*I, 1.0)
>>> v[Integer(1)].parent()                                                      ␣
↪     # needs sage.symbolic
Complex Double Field
```

If the argument is a vector, it doesn't change the base ring. This fixes Issue #6643:

```
sage: # needs sage.rings.number_field
sage: K.<sqrt3> = QuadraticField(3)
sage: u = vector(K, (1/2, sqrt3/2))
sage: vector(u).base_ring()
Number Field in sqrt3 with defining polynomial x^2 - 3 with sqrt3 = 1.
↪732050807568878?
sage: v = vector(K, (0, 1))
sage: vector(v).base_ring()
Number Field in sqrt3 with defining polynomial x^2 - 3 with sqrt3 = 1.
↪732050807568878?
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> K = QuadraticField(Integer(3), names=('sqrt3',)); (sqrt3,) = K._first_ngens(1)
>>> u = vector(K, (Integer(1)/Integer(2), sqrt3/Integer(2)))
>>> vector(u).base_ring()
Number Field in sqrt3 with defining polynomial x^2 - 3 with sqrt3 = 1.
↪732050807568878?
>>> v = vector(K, (Integer(0), Integer(1)))
>>> vector(v).base_ring()
Number Field in sqrt3 with defining polynomial x^2 - 3 with sqrt3 = 1.
↪732050807568878?
```

Constructing a vector from a numpy array behaves as expected:

```
sage: # needs numpy
sage: import numpy
sage: a = numpy.array([1,2,3])
sage: v = vector(a); v
(1, 2, 3)
sage: parent(v)
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> # needs numpy
>>> import numpy
>>> a = numpy.array([Integer(1),Integer(2),Integer(3)])
>>> v = vector(a); v
(1, 2, 3)
>>> parent(v)
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

Complex numbers can be converted naturally to a sequence of length 2. And then to a vector.

```
sage: c = CDF(2 + 3*I)                                                          #␣
↪needs sage.rings.complex_double sage.symbolic
sage: v = vector(c); v                                                          #␣
↪needs sage.rings.complex_double sage.symbolic
(2.0, 3.0)
```

```
>>> from sage.all import *
>>> c = CDF(Integer(2) + Integer(3)*I)                                          ␣
↪              # needs sage.rings.complex_double sage.symbolic
>>> v = vector(c); v                                                          #␣
↪needs sage.rings.complex_double sage.symbolic
(2.0, 3.0)
```

A generator, or other iterable, may also be supplied as input. Anything that can be converted to a `Sequence` is a possible input.

```
sage: type(i^2 for i in range(3))
<... 'generator'>
sage: v = vector(i^2 for i in range(3)); v
(0, 1, 4)
```

```
>>> from sage.all import *
>>> type(i**Integer(2) for i in range(Integer(3)))
<... 'generator'>
>>> v = vector(i**Integer(2) for i in range(Integer(3))); v
(0, 1, 4)
```

An empty list, without a ring given, will default to the integers.

```
sage: x = vector([]); x
()
sage: x.parent()
Ambient free module of rank 0 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> x = vector([]); x
()
>>> x.parent()
Ambient free module of rank 0 over the principal ideal domain Integer Ring
```

The `immutable` switch allows to create an immutable vector.

```
sage: v = vector(QQ, {0:1/2, 4:-6, 7:0}, immutable=True); v
(1/2, 0, 0, 0, -6, 0, 0, 0)
sage: v.is_immutable()
True
```

```
>>> from sage.all import *
>>> v = vector(QQ, {Integer(0):Integer(1)/Integer(2), Integer(4):-Integer(6),␣
↪Integer(7):Integer(0)}, immutable=True); v
(1/2, 0, 0, 0, -6, 0, 0, 0)
>>> v.is_immutable()
True
```

The `immutable` switch works regardless of the type of valid input to the constructor.

```
sage: v = vector(ZZ, 4, immutable=True)
sage: v.is_immutable()
True
sage: w = vector(ZZ, [1,2,3])
sage: v = vector(w, ZZ, immutable=True)
sage: v.is_immutable()
True
sage: v = vector(QQ, w, immutable=True)
sage: v.is_immutable()
True

sage: # needs numpy sage.symbolic
sage: import numpy as np
sage: w = np.array([1, 2, pi], float)
sage: v = vector(w, immutable=True)
sage: v.is_immutable()
True
sage: w = np.array([i, 2, 3], complex)
```

(continues on next page)

(continued from previous page)

```
sage: v = vector(w, immutable=True)
sage: v.is_immutable()
True
```

```
>>> from sage.all import *
>>> v = vector(ZZ, Integer(4), immutable=True)
>>> v.is_immutable()
True
>>> w = vector(ZZ, [Integer(1),Integer(2),Integer(3)])
>>> v = vector(w, ZZ, immutable=True)
>>> v.is_immutable()
True
>>> v = vector(QQ, w, immutable=True)
>>> v.is_immutable()
True

>>> # needs numpy sage.symbolic
>>> import numpy as np
>>> w = np.array([Integer(1), Integer(2), pi], float)
>>> v = vector(w, immutable=True)
>>> v.is_immutable()
True
>>> w = np.array([i, Integer(2), Integer(3)], complex)
>>> v = vector(w, immutable=True)
>>> v.is_immutable()
True
```

sage.modules.free_module_element.**zero_vector**(*arg0*, *arg1=None*)

Return a vector or free module element with a specified number of zeros.

CALL FORMATS:

1. zero_vector(degree)
2. zero_vector(ring, degree)

INPUT:

- `degree` – the number of zero entries in the vector or free module element
- `ring` – (default: `ZZ`) the base ring of the vector space or module containing the constructed zero vector

OUTPUT:

A vector or free module element with `degree` entries, all equal to zero and belonging to the ring if specified. If no ring is given, a free module element over `ZZ` is returned.

EXAMPLES:

A zero vector over the field of rationals.

```
sage: v = zero_vector(QQ, 5); v
(0, 0, 0, 0, 0)
sage: v.parent()
Vector space of dimension 5 over Rational Field
```

```
>>> from sage.all import *
>>> v = zero_vector(QQ, Integer(5)); v
(0, 0, 0, 0, 0)
>>> v.parent()
Vector space of dimension 5 over Rational Field
```

A free module zero element.

```
sage: w = zero_vector(Integers(6), 3); w
(0, 0, 0)
sage: w.parent()
Ambient free module of rank 3 over Ring of integers modulo 6
```

```
>>> from sage.all import *
>>> w = zero_vector(Integers(Integer(6)), Integer(3)); w
(0, 0, 0)
>>> w.parent()
Ambient free module of rank 3 over Ring of integers modulo 6
```

If no ring is given, the integers are used.

```
sage: u = zero_vector(9); u
(0, 0, 0, 0, 0, 0, 0, 0, 0)
sage: u.parent()
Ambient free module of rank 9 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> u = zero_vector(Integer(9)); u
(0, 0, 0, 0, 0, 0, 0, 0, 0)
>>> u.parent()
Ambient free module of rank 9 over the principal ideal domain Integer Ring
```

Non-integer degrees produce an error.

```
sage: zero_vector(5.6)
Traceback (most recent call last):
...
TypeError: Attempt to coerce non-integral RealNumber to Integer
```

```
>>> from sage.all import *
>>> zero_vector(RealNumber('5.6'))
Traceback (most recent call last):
...
TypeError: Attempt to coerce non-integral RealNumber to Integer
```

Negative degrees also give an error.

```
sage: zero_vector(-3)
Traceback (most recent call last):
...
ValueError: rank (=-3) must be nonnegative
```

```
>>> from sage.all import *
>>> zero_vector(-Integer(3))
Traceback (most recent call last):
...
ValueError: rank (=-3) must be nonnegative
```

Garbage instead of a ring will be recognized as such.

```
sage: zero_vector(x^2, 5)                                                    #␣
↪needs sage.symbolic
Traceback (most recent call last):
...
TypeError: first argument must be a ring
```

```
>>> from sage.all import *
>>> zero_vector(x**Integer(2), Integer(5))                                    ␣
↪               # needs sage.symbolic
Traceback (most recent call last):
...
TypeError: first argument must be a ring
```

2.4 Submodules and subquotients of free modules

Free modules and submodules of a free module (of finite rank) over a principal ideal domain have well-defined notion of rank, and they are implemented in *sage.modules.free_module*. Here submodules with no rank are implemented. For example, submodules of free modules over multivariate polynomial rings with more than one variables have no notion of rank.

EXAMPLES:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: N = M.submodule([vector([x - y, z]), vector([y * z, x * z])])
sage: N
Submodule of Ambient free module of rank 2 over the integral domain Multivariate␣
↪Polynomial Ring in x, y, z over Rational Field
Generated by the rows of the matrix:
[x - y     z]
[  y*z   x*z]
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_ngens(3)
>>> M = S**Integer(2)
>>> N = M.submodule([vector([x - y, z]), vector([y * z, x * z])])
>>> N
Submodule of Ambient free module of rank 2 over the integral domain Multivariate␣
↪Polynomial Ring in x, y, z over Rational Field
Generated by the rows of the matrix:
[x - y     z]
[  y*z   x*z]
```

AUTHORS:

- Kwankyu Lee (2022-05): initial version

class sage.modules.submodule.**Submodule_free_ambient**(*ambient*, *gens*, *check=True*, *already_echelonized=False*)

Bases: *Module_free_ambient*

Base class of submodules of ambient modules.

The ambient module is either a free module or a quotient of a free module by a submodule.

Note that if the ambient module is a quotient module, submodules of the quotient module are called subquotients.

INPUT:

- `ambient` – an ambient module
- `gens` – vectors of the ambient free module generating this submodule
- `check` – boolean; if `True`, vectors in `gens` are checked whether they belong to the ambient free module
- `already_echelonized` – ignored; for compatibility with other submodules

EXAMPLES:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: N = M.submodule([vector([x - y, z]), vector([y * z, x * z])])
sage: N
Submodule of Ambient free module of rank 2 over the integral domain
Multivariate Polynomial Ring in x, y, z over Rational Field
Generated by the rows of the matrix:
[x - y     z]
[  y*z   x*z]
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_ngens(3)
>>> M = S**Integer(2)
>>> N = M.submodule([vector([x - y, z]), vector([y * z, x * z])])
>>> N
Submodule of Ambient free module of rank 2 over the integral domain
Multivariate Polynomial Ring in x, y, z over Rational Field
Generated by the rows of the matrix:
[x - y     z]
[  y*z   x*z]
```

```
sage: M.coerce_map_from(N)
Coercion map:
  From: Submodule of Ambient free module of rank 2 over the integral domain
Multivariate Polynomial Ring in x, y, z over Rational Field
Generated by the rows of the matrix:
[x - y     z]
[  y*z   x*z]
  To:   Ambient free module of rank 2 over the integral domain
Multivariate Polynomial Ring in x, y, z over Rational Field
```

```
>>> from sage.all import *
>>> M.coerce_map_from(N)
```

(continues on next page)

(continued from previous page)

```
Coercion map:
  From: Submodule of Ambient free module of rank 2 over the integral domain
Multivariate Polynomial Ring in x, y, z over Rational Field
Generated by the rows of the matrix:
[x - y     z]
[  y*z   x*z]
  To:   Ambient free module of rank 2 over the integral domain
Multivariate Polynomial Ring in x, y, z over Rational Field
```

ambient_module()

Return the ambient module of `self`.

EXAMPLES:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: N = M.submodule([vector([x - y, z]), vector([y * z, x * z])])
sage: N.ambient_module() is M
True
sage: N.zero_submodule().ambient_module() is M
True
sage: Q = M / N
sage: Q.zero_submodule().ambient_module() is Q
True
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_
↪ngens(3)
>>> M = S**Integer(2)
>>> N = M.submodule([vector([x - y, z]), vector([y * z, x * z])])
>>> N.ambient_module() is M
True
>>> N.zero_submodule().ambient_module() is M
True
>>> Q = M / N
>>> Q.zero_submodule().ambient_module() is Q
True
```

gen(*i=0*)

Return the i-th generator of this module.

EXAMPLES:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
sage: N.gen(0)
(x - y, z)
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_
↪ngens(3)
```

(continues on next page)

(continued from previous page)

```
>>> M = S**Integer(2)
>>> N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
>>> N.gen(Integer(0))
(x - y, z)
```

generators_matrix()

Return the matrix defining `self`.

EXAMPLES:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
sage: N.matrix()
[x - y     z]
[  y*z   x*z]
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_
↪ngens(3)
>>> M = S**Integer(2)
>>> N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
>>> N.matrix()
[x - y     z]
[  y*z   x*z]
```

gens()

Return the generators of this submodule.

EXAMPLES:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: N = M.submodule([vector([x - y, z]), vector([y * z, x * z])])
sage: N.gens()
[(x - y, z), (y*z, x*z)]
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_
↪ngens(3)
>>> M = S**Integer(2)
>>> N = M.submodule([vector([x - y, z]), vector([y * z, x * z])])
>>> N.gens()
[(x - y, z), (y*z, x*z)]
```

matrix()

Return the matrix defining `self`.

EXAMPLES:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
```

(continues on next page)

(continued from previous page)

```
sage: N.matrix()
[x - y     z]
[  y*z   x*z]
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_
↪ngens(3)
>>> M = S**Integer(2)
>>> N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
>>> N.matrix()
[x - y     z]
[  y*z   x*z]
```

relations()

Return the relations module of the ambient module.

If the ambient module is free, then the relations module is trivial.

EXAMPLES:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: N = M.submodule([vector([x - y, z]), vector([y * z, x * z])])
sage: N.relations() == M.zero_submodule()
True

sage: Q = M.quotient(N)
sage: Q.zero_submodule().relations() == N
True
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_
↪ngens(3)
>>> M = S**Integer(2)
>>> N = M.submodule([vector([x - y, z]), vector([y * z, x * z])])
>>> N.relations() == M.zero_submodule()
True

>>> Q = M.quotient(N)
>>> Q.zero_submodule().relations() == N
True
```

2.5 Quotients of free modules

AUTHORS:

- William Stein (2009): initial version
- Kwankyu Lee (2022-05): added quotient module over domain

class sage.modules.quotient_module.**FreeModule_ambient_field_quotient**(*domain*, *sub*, *quotient_matrix*, *lift_matrix*, *inner_product_matrix=None*)

Bases: *FreeModule_ambient_field*

A quotient V/W of two vector spaces as a vector space.

To obtain V or W use `self.V()` and `self.W()`.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: k.<i> = QuadraticField(-1)
sage: A = k^3; V = A.span([[1,0,i], [2,i,0]])
sage: W = A.span([[3,i,i]])
sage: U = V/W; U
Vector space quotient V/W of dimension 1 over Number Field in i
 with defining polynomial x^2 + 1 with i = 1*I where
 V: Vector space of degree 3 and dimension 2 over Number Field in i
    with defining polynomial x^2 + 1 with i = 1*I
    Basis matrix:
    [ 1  0  i]
    [ 0  1 -2]
 W: Vector space of degree 3 and dimension 1 over Number Field in i
    with defining polynomial x^2 + 1 with i = 1*I
    Basis matrix:
    [    1 1/3*i 1/3*i]
sage: U.V()
Vector space of degree 3 and dimension 2 over Number Field in i
 with defining polynomial x^2 + 1 with i = 1*I
 Basis matrix:
 [ 1  0  i]
 [ 0  1 -2]
sage: U.W()
Vector space of degree 3 and dimension 1 over Number Field in i
 with defining polynomial x^2 + 1 with i = 1*I
 Basis matrix:
 [    1 1/3*i 1/3*i]
sage: U.quotient_map()
Vector space morphism represented by the matrix:
[  1]
[3*i]
Domain:   Vector space of degree 3 and dimension 2 over Number Field in i
          with defining polynomial x^2 + 1 with i = 1*I
          Basis matrix:
          [ 1  0  i]
          [ 0  1 -2]
Codomain: Vector space quotient V/W of dimension 1 over Number Field in i
          with defining polynomial x^2 + 1 with i = 1*I where
          V: Vector space of degree 3 and dimension 2 over Number Field in i
             with defining polynomial x^2 + 1 with i = 1*I
             Basis matrix:
             [ 1  0  i]
             [ 0  1 -2]
          W: Vector space of degree 3 and dimension 1 over Number Field in i
             with defining polynomial x^2 + 1 with i = 1*I
             Basis matrix:
```

(continues on next page)

(continued from previous page)

```
             [    1 1/3*i 1/3*i]
sage: Z = V.quotient(W)
sage: Z == U
True
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> k = QuadraticField(-Integer(1), names=('i',)); (i,) = k._first_ngens(1)
>>> A = k**Integer(3); V = A.span([[Integer(1),Integer(0),i], [Integer(2),i,
↪Integer(0)]])
>>> W = A.span([[Integer(3),i,i]])
>>> U = V/W; U
Vector space quotient V/W of dimension 1 over Number Field in i
 with defining polynomial x^2 + 1 with i = 1*I where
 V: Vector space of degree 3 and dimension 2 over Number Field in i
    with defining polynomial x^2 + 1 with i = 1*I
    Basis matrix:
    [ 1  0  i]
    [ 0  1 -2]
 W: Vector space of degree 3 and dimension 1 over Number Field in i
    with defining polynomial x^2 + 1 with i = 1*I
    Basis matrix:
    [    1 1/3*i 1/3*i]
>>> U.V()
Vector space of degree 3 and dimension 2 over Number Field in i
 with defining polynomial x^2 + 1 with i = 1*I
 Basis matrix:
 [ 1  0  i]
 [ 0  1 -2]
>>> U.W()
Vector space of degree 3 and dimension 1 over Number Field in i
 with defining polynomial x^2 + 1 with i = 1*I
 Basis matrix:
 [    1 1/3*i 1/3*i]
>>> U.quotient_map()
Vector space morphism represented by the matrix:
[  1]
[3*i]
Domain:   Vector space of degree 3 and dimension 2 over Number Field in i
          with defining polynomial x^2 + 1 with i = 1*I
          Basis matrix:
          [ 1  0  i]
          [ 0  1 -2]
Codomain: Vector space quotient V/W of dimension 1 over Number Field in i
          with defining polynomial x^2 + 1 with i = 1*I where
          V: Vector space of degree 3 and dimension 2 over Number Field in i
             with defining polynomial x^2 + 1 with i = 1*I
             Basis matrix:
             [ 1  0  i]
             [ 0  1 -2]
          W: Vector space of degree 3 and dimension 1 over Number Field in i
             with defining polynomial x^2 + 1 with i = 1*I
```

(continues on next page)

(continued from previous page)

```
            Basis matrix:
            [    1 1/3*i 1/3*i]
>>> Z = V.quotient(W)
>>> Z == U
True
```

We create three quotient spaces and compare them:

```
sage: A = QQ^2
sage: V = A.span_of_basis([[1,0], [1,1]])
sage: W0 = V.span([V.1, V.0])
sage: W1 = V.span([V.1])
sage: W2 = V.span([V.1])
sage: Q0 = V/W0
sage: Q1 = V/W1
sage: Q2 = V/W2

sage: Q0 == Q1
False
sage: Q1 == Q2
True
```

```
>>> from sage.all import *
>>> A = QQ**Integer(2)
>>> V = A.span_of_basis([[Integer(1),Integer(0)], [Integer(1),Integer(1)]])
>>> W0 = V.span([V.gen(1), V.gen(0)])
>>> W1 = V.span([V.gen(1)])
>>> W2 = V.span([V.gen(1)])
>>> Q0 = V/W0
>>> Q1 = V/W1
>>> Q2 = V/W2

>>> Q0 == Q1
False
>>> Q1 == Q2
True
```

V()

Given this quotient space $Q = V/W$, return V.

EXAMPLES:

```
sage: M = QQ^10 / [list(range(10)), list(range(2,12))]
sage: M.cover()
Vector space of dimension 10 over Rational Field
```

```
>>> from sage.all import *
>>> M = QQ**Integer(10) / [list(range(Integer(10))), list(range(Integer(2),
↪Integer(12)))]
>>> M.cover()
Vector space of dimension 10 over Rational Field
```

W()

Given this quotient space $Q = V/W$, return W.

EXAMPLES:

```
sage: M = QQ^10 / [list(range(10)), list(range(2,12))]
sage: M.relations()
Vector space of degree 10 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1 -2 -3 -4 -5 -6 -7 -8]
[ 0  1  2  3  4  5  6  7  8  9]
```

```
>>> from sage.all import *
>>> M = QQ**Integer(10) / [list(range(Integer(10))), list(range(Integer(2),
↪Integer(12)))]
>>> M.relations()
Vector space of degree 10 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1 -2 -3 -4 -5 -6 -7 -8]
[ 0  1  2  3  4  5  6  7  8  9]
```

cover()

Given this quotient space $Q = V/W$, return V.

EXAMPLES:

```
sage: M = QQ^10 / [list(range(10)), list(range(2,12))]
sage: M.cover()
Vector space of dimension 10 over Rational Field
```

```
>>> from sage.all import *
>>> M = QQ**Integer(10) / [list(range(Integer(10))), list(range(Integer(2),
↪Integer(12)))]
>>> M.cover()
Vector space of dimension 10 over Rational Field
```

lift(x)

Lift element of this quotient V/W to V by applying the fixed lift homomorphism.

The lift is a fixed homomorphism.

EXAMPLES:

```
sage: M = QQ^3 / [[1,2,3]]
sage: M.lift(M.0)
(1, 0, 0)
sage: M.lift(M.1)
(0, 1, 0)
sage: M.lift(M.0 - 2*M.1)
(1, -2, 0)
```

```
>>> from sage.all import *
>>> M = QQ**Integer(3) / [[Integer(1),Integer(2),Integer(3)]]
>>> M.lift(M.gen(0))
```

(continues on next page)

(continued from previous page)

```
(1, 0, 0)
>>> M.lift(M.gen(1))
(0, 1, 0)
>>> M.lift(M.gen(0) - Integer(2)*M.gen(1))
(1, -2, 0)
```

lift_map()

Given this quotient space $Q = V/W$, return a fixed choice of linear homomorphism (a section) from Q to V.

EXAMPLES:

```
sage: M = QQ^3 / [[1,2,3]]
sage: M.lift_map()
Vector space morphism represented by the matrix:
[1 0 0]
[0 1 0]
Domain:   Vector space quotient V/W of dimension 2 over Rational Field where
          V: Vector space of dimension 3 over Rational Field
          W: Vector space of degree 3 and dimension 1 over Rational Field
             Basis matrix:
             [1 2 3]
Codomain: Vector space of dimension 3 over Rational Field
```

```
>>> from sage.all import *
>>> M = QQ**Integer(3) / [[Integer(1),Integer(2),Integer(3)]]
>>> M.lift_map()
Vector space morphism represented by the matrix:
[1 0 0]
[0 1 0]
Domain:   Vector space quotient V/W of dimension 2 over Rational Field where
          V: Vector space of dimension 3 over Rational Field
          W: Vector space of degree 3 and dimension 1 over Rational Field
             Basis matrix:
             [1 2 3]
Codomain: Vector space of dimension 3 over Rational Field
```

quotient_map()

Given this quotient space $Q = V/W$, return the natural quotient map from V to Q.

EXAMPLES:

```
sage: M = QQ^3 / [[1,2,3]]
sage: M.quotient_map()
Vector space morphism represented by the matrix:
[   1    0]
[   0    1]
[-1/3 -2/3]
Domain:   Vector space of dimension 3 over Rational Field
Codomain: Vector space quotient V/W of dimension 2 over Rational Field where
          V: Vector space of dimension 3 over Rational Field
          W: Vector space of degree 3 and dimension 1 over Rational Field
          Basis matrix:
```

(continues on next page)

(continued from previous page)

```
          [1 2 3]

sage: M.quotient_map()( (QQ^3)([1,2,3]) )
(0, 0)
```

```
>>> from sage.all import *
>>> M = QQ**Integer(3) / [[Integer(1),Integer(2),Integer(3)]]
>>> M.quotient_map()
Vector space morphism represented by the matrix:
[   1    0]
[   0    1]
[-1/3 -2/3]
Domain:   Vector space of dimension 3 over Rational Field
Codomain: Vector space quotient V/W of dimension 2 over Rational Field where
          V: Vector space of dimension 3 over Rational Field
          W: Vector space of degree 3 and dimension 1 over Rational Field
          Basis matrix:
          [1 2 3]

>>> M.quotient_map()( (QQ**Integer(3))([Integer(1),Integer(2),Integer(3)]) )
(0, 0)
```

relations()

Given this quotient space $Q = V/W$, return W.

EXAMPLES:

```
sage: M = QQ^10 / [list(range(10)), list(range(2,12))]
sage: M.relations()
Vector space of degree 10 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1 -2 -3 -4 -5 -6 -7 -8]
[ 0  1  2  3  4  5  6  7  8  9]
```

```
>>> from sage.all import *
>>> M = QQ**Integer(10) / [list(range(Integer(10))), list(range(Integer(2),
↪Integer(12)))]
>>> M.relations()
Vector space of degree 10 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1 -2 -3 -4 -5 -6 -7 -8]
[ 0  1  2  3  4  5  6  7  8  9]
```

class sage.modules.quotient_module.**QuotientModule_free_ambient**(*module*, *sub*)

Bases: *Module_free_ambient*

Quotients of ambient free modules by a submodule.

INPUT:

- `module` – an ambient free module
- `sub` – a submodule of `module`

EXAMPLES:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
sage: M.quotient_module(N)
Quotient module by Submodule of Ambient free module of rank 2 over
 the integral domain Multivariate Polynomial Ring in x, y, z over Rational Field
 Generated by the rows of the matrix:
 [x - y     z]
 [  y*z   x*z]
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_ngens(3)
>>> M = S**Integer(2)
>>> N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
>>> M.quotient_module(N)
Quotient module by Submodule of Ambient free module of rank 2 over
 the integral domain Multivariate Polynomial Ring in x, y, z over Rational Field
 Generated by the rows of the matrix:
 [x - y     z]
 [  y*z   x*z]
```

V()

Given this quotient space $Q = V/W$, return V.

EXAMPLES:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
sage: Q = M.quotient_module(N)
sage: Q.cover() is M
True
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_
↪ngens(3)
>>> M = S**Integer(2)
>>> N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
>>> Q = M.quotient_module(N)
>>> Q.cover() is M
True
```

W()

Given this quotient space $Q = V/W$, return W.

EXAMPLES:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
sage: Q = M.quotient_module(N)
sage: Q.relations() is N
True
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_
↪ngens(3)
>>> M = S**Integer(2)
>>> N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
>>> Q = M.quotient_module(N)
>>> Q.relations() is N
True
```

ambient_module()

Return `self`, since `self` is ambient.

EXAMPLES:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
sage: Q = M.quotient_module(N)
sage: Q.ambient_module() is Q
True
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_
↪ngens(3)
>>> M = S**Integer(2)
>>> N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
>>> Q = M.quotient_module(N)
>>> Q.ambient_module() is Q
True
```

cover()

Given this quotient space $Q = V/W$, return V.

EXAMPLES:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
sage: Q = M.quotient_module(N)
sage: Q.cover() is M
True
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_
↪ngens(3)
>>> M = S**Integer(2)
>>> N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
>>> Q = M.quotient_module(N)
>>> Q.cover() is M
True
```

free_cover()

Given this quotient space $Q = V/W$, return the free module that covers V.

EXAMPLES:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
sage: Q = M / N
sage: NQ = Q.submodule([Q([1,x])])
sage: QNQ = Q / NQ
sage: QNQ.free_cover() is Q.free_cover() is M
True
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_
↪ngens(3)
>>> M = S**Integer(2)
>>> N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
>>> Q = M / N
>>> NQ = Q.submodule([Q([Integer(1),x])])
>>> QNQ = Q / NQ
>>> QNQ.free_cover() is Q.free_cover() is M
True
```

Note that this is different than the immediate cover:

```
sage: QNQ.cover() is Q
True
sage: QNQ.cover() is QNQ.free_cover()
False
```

```
>>> from sage.all import *
>>> QNQ.cover() is Q
True
>>> QNQ.cover() is QNQ.free_cover()
False
```

free_relations()

Given this quotient space $Q = V/W$, return the submodule that generates all relations of Q.

When V is a free module, then this returns W. Otherwise this returns the union of W lifted to the cover of V and the relations of V (repeated until W is a submodule of a free module).

EXAMPLES:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
sage: Q = M / N
sage: NQ = Q.submodule([Q([1, x])])
sage: QNQ = Q / NQ
sage: QNQ.free_relations()
Submodule of Ambient free module of rank 2 over the integral domain
Multivariate Polynomial Ring in x, y, z over Rational Field
Generated by the rows of the matrix:
[    1     x]
```

(continues on next page)

(continued from previous page)

```
[x - y     z]
[  y*z   x*z]
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_
↪ngens(3)
>>> M = S**Integer(2)
>>> N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
>>> Q = M / N
>>> NQ = Q.submodule([Q([Integer(1), x])])
>>> QNQ = Q / NQ
>>> QNQ.free_relations()
Submodule of Ambient free module of rank 2 over the integral domain
Multivariate Polynomial Ring in x, y, z over Rational Field
Generated by the rows of the matrix:
[    1     x]
[x - y     z]
[  y*z   x*z]
```

Note that this is different than the defining relations:

```
sage: QNQ.relations() is NQ
True
sage: QNQ.relations() == QNQ.free_relations()
False
```

```
>>> from sage.all import *
>>> QNQ.relations() is NQ
True
>>> QNQ.relations() == QNQ.free_relations()
False
```

gen(*i=0*)

Return the i-th generator of this module.

EXAMPLES:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
sage: Q = M.quotient_module(N)
sage: Q.gen(0)
(1, 0)
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_
↪ngens(3)
>>> M = S**Integer(2)
>>> N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
>>> Q = M.quotient_module(N)
>>> Q.gen(Integer(0))
(1, 0)
```

gens()

Return the generators of this module.

EXAMPLES:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
sage: Q = M.quotient_module(N)
sage: Q.gens()
((1, 0), (0, 1))
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_
↪ngens(3)
>>> M = S**Integer(2)
>>> N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
>>> Q = M.quotient_module(N)
>>> Q.gens()
((1, 0), (0, 1))
```

relations()

Given this quotient space $Q = V/W$, return W.

EXAMPLES:

```
sage: S.<x,y,z> = PolynomialRing(QQ)
sage: M = S**2
sage: N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
sage: Q = M.quotient_module(N)
sage: Q.relations() is N
True
```

```
>>> from sage.all import *
>>> S = PolynomialRing(QQ, names=('x', 'y', 'z',)); (x, y, z,) = S._first_
↪ngens(3)
>>> M = S**Integer(2)
>>> N = M.submodule([vector([x - y, z]), vector([y*z, x*z])])
>>> Q = M.quotient_module(N)
>>> Q.relations() is N
True
```

See also sage.tensor.modules.finite_rank_free_module.

CHAPTER

THREE

MODULES WITH BASIS

3.1 Concrete classes related to modules with a distinguished basis

This module provides concrete classes for various constructions related to modules with a distinguished basis:

- `morphism` – Concrete classes for morphisms of modules with basis

See also

The category `ModulesWithBasis`

3.2 Cell modules

class sage.modules.with_basis.cell_module.**CellModule**(*A*, *mu*, ***kwds*)

Bases: `CombinatorialFreeModule`

A cell module.

Let R be a commutative ring. Let A be a cellular R-algebra with cell datum (Λ, i, M, C). A *cell module* $W(\lambda)$ is the R-module given by $R\{C_s \mid s \in M(\lambda)\}$ with an action of $a \in A$ given by $aC_s = \sum_{u \in M(\lambda)} r_a(u,s) C_u$, where $r_a(u,s)$ is the same as those given by the cellular condition:

$$aC_{st}^{\lambda} = \sum_{u \in M(\lambda)} r_a(u,s) C_{ut}^{\lambda} + \sum_{\substack{\mu < \lambda \\ x,y \in M(\mu)}} R C_{xy}^{\mu}.$$

INPUT:

- `A` – a cellular algebra
- `mu` – an element of the cellular poset of `A`

See also

`CellularBasis`

AUTHORS:

- 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 Element

Bases: *IndexedFreeModuleElement*

bilinear_form(*x*, *y*)

Return the bilinear form on `x` and `y`.

The cell module $W(\lambda)$ has a canonical bilinear form $\Phi_\lambda : W(\lambda) \times W(\lambda) \to W(\lambda)$ given by

$$C_{ss}^\lambda C_{tt}^\lambda = \Phi_\lambda(C_s, C_t) C_{st}^\lambda + \sum_{\substack{\mu<\lambda \\ x,y\in M(\mu)}} RC_{xy}^\mu.$$

EXAMPLES:

```
sage: S = SymmetricGroupAlgebra(QQ, 3)
sage: W = S.cell_module([2,1])
sage: elt = W.an_element(); elt
2*W[[1, 2], [3]] + 2*W[[1, 3], [2]]
sage: W.bilinear_form(elt, elt)
8
```

```
>>> from sage.all import *
>>> S = SymmetricGroupAlgebra(QQ, Integer(3))
>>> W = S.cell_module([Integer(2),Integer(1)])
>>> elt = W.an_element(); elt
2*W[[1, 2], [3]] + 2*W[[1, 3], [2]]
>>> W.bilinear_form(elt, elt)
8
```

bilinear_form_matrix(*ordering=None*)

Return the matrix corresponding to the bilinear form of `self`.

INPUT:

- `ordering` – (optional) an ordering of the indices

EXAMPLES:

```
sage: S = SymmetricGroupAlgebra(QQ, 3)
sage: W = S.cell_module([2,1])
sage: W.bilinear_form_matrix()
[1 0]
[0 1]
```

```
>>> from sage.all import *
>>> S = SymmetricGroupAlgebra(QQ, Integer(3))
>>> W = S.cell_module([Integer(2),Integer(1)])
>>> W.bilinear_form_matrix()
[1 0]
[0 1]
```

cellular_algebra()

Return the cellular algebra of `self`.

EXAMPLES:

```
sage: S = SymmetricGroupAlgebra(QQ, 3)
sage: W = S.cell_module([2,1])
sage: W.cellular_algebra() is S.cellular_basis()
True
sage: S.has_coerce_map_from(W.cellular_algebra())
True
```

```
>>> from sage.all import *
>>> S = SymmetricGroupAlgebra(QQ, Integer(3))
>>> W = S.cell_module([Integer(2),Integer(1)])
>>> W.cellular_algebra() is S.cellular_basis()
True
>>> S.has_coerce_map_from(W.cellular_algebra())
True
```

nonzero_bilinear_form()

Return `True` if the bilinear form of `self` is nonzero.

EXAMPLES:

```
sage: S = SymmetricGroupAlgebra(QQ, 3)
sage: W = S.cell_module([2,1])
sage: W.nonzero_bilinear_form()
True
```

```
>>> from sage.all import *
>>> S = SymmetricGroupAlgebra(QQ, Integer(3))
>>> W = S.cell_module([Integer(2),Integer(1)])
>>> W.nonzero_bilinear_form()
True
```

radical()

Return the radical of `self`.

Let $W(\lambda)$ denote a cell module. The *radical* of $W(\lambda)$ is defined as

$$\operatorname{rad}(\lambda) := \{x \in W(\lambda) \mid \Phi_\lambda(x, y)\},$$

and note that it is a submodule of $W(\lambda)$.

EXAMPLES:

```
sage: S = SymmetricGroupAlgebra(QQ, 3)
sage: W = S.cell_module([2,1])
sage: R = W.radical(); R
Radical of Cell module indexed by [2, 1] of Cellular basis of
 Symmetric group algebra of order 3 over Rational Field
sage: R.basis()
Finite family {}
```

```
>>> from sage.all import *
>>> S = SymmetricGroupAlgebra(QQ, Integer(3))
>>> W = S.cell_module([Integer(2),Integer(1)])
>>> R = W.radical(); R
Radical of Cell module indexed by [2, 1] of Cellular basis of
 Symmetric group algebra of order 3 over Rational Field
>>> R.basis()
Finite family {}
```

radical_basis()

Return a basis of the radical of `self`.

EXAMPLES:

```
sage: S = SymmetricGroupAlgebra(QQ, 3)
sage: W = S.cell_module([2,1])
sage: W.radical_basis()
()
```

```
>>> from sage.all import *
>>> S = SymmetricGroupAlgebra(QQ, Integer(3))
>>> W = S.cell_module([Integer(2),Integer(1)])
>>> W.radical_basis()
()
```

simple_module()

Return the corresponding simple module of `self`.

Let $W(\lambda)$ denote a cell module. The simple module $L(\lambda)$ is defined as $W(\lambda)/\operatorname{rad}(\lambda)$, where $\operatorname{rad}(\lambda)$ is the radical of the bilinear form Φ_λ.

See also

radical()

EXAMPLES:

```
sage: S = SymmetricGroupAlgebra(QQ, 3)
sage: W = S.cell_module([2,1])
sage: L = W.simple_module(); L
Simple module indexed by [2, 1] of Cellular basis of
 Symmetric group algebra of order 3 over Rational Field
sage: L.has_coerce_map_from(W)
True
```

```
>>> from sage.all import *
>>> S = SymmetricGroupAlgebra(QQ, Integer(3))
>>> W = S.cell_module([Integer(2),Integer(1)])
>>> L = W.simple_module(); L
Simple module indexed by [2, 1] of Cellular basis of
 Symmetric group algebra of order 3 over Rational Field
>>> L.has_coerce_map_from(W)
True
```

class sage.modules.with_basis.cell_module.**SimpleModule**(*submodule*)

Bases: *QuotientModuleWithBasis*

A simple module of a cellular algebra.

Let $W(\lambda)$ denote a cell module. The simple module $L(\lambda)$ is defined as $W(\lambda)/\operatorname{rad}(\lambda)$, where $\operatorname{rad}(\lambda)$ is the radical of the bilinear form Φ_λ.

class Element

Bases: *IndexedFreeModuleElement*

3.3 An element in an indexed free module

AUTHORS:

- Travis Scrimshaw (03-2017): Moved code from `sage.combinat.free_module`.
- Travis Scrimshaw (29-08-2022): Implemented `copy` as the identity map.

class sage.modules.with_basis.indexed_element.**IndexedFreeModuleElement**

Bases: ModuleElement

Element class for CombinatorialFreeModule.

monomial_coefficients(*copy=True*)

Return the internal dictionary which has the combinatorial objects indexing the basis as keys and their corresponding coefficients as values.

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 = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']
sage: d = f.monomial_coefficients()
sage: d['a']
1
sage: d['c']
3
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(QQ, ['a','b','c'])
>>> B = F.basis()
>>> f = B['a'] + Integer(3)*B['c']
>>> d = f.monomial_coefficients()
>>> d['a']
1
>>> d['c']
3
```

To run through the monomials of an element, it is better to use the idiom:

```
sage: for (t,c) in f:
....:     print("{} {}".format(t,c))
a 1
c 3
```

```
>>> from sage.all import *
>>> for (t,c) in f:
...     print("{} {}".format(t,c))
a 1
c 3
```

```
sage: # needs sage.combinat
sage: s = SymmetricFunctions(QQ).schur()
sage: a = s([2,1])+2*s([3,2])
sage: d = a.monomial_coefficients()
sage: type(d)
<... 'dict'>
sage: d[ Partition([2,1]) ]
1
sage: d[ Partition([3,2]) ]
2
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> s = SymmetricFunctions(QQ).schur()
>>> a = s([Integer(2),Integer(1)])+Integer(2)*s([Integer(3),Integer(2)])
>>> d = a.monomial_coefficients()
>>> type(d)
<... 'dict'>
>>> d[ Partition([Integer(2),Integer(1)]) ]
1
>>> d[ Partition([Integer(3),Integer(2)]) ]
2
```

to_vector(*new_base_ring=None*, *order=None*, *sparse=False*)

Return `self` as a vector.

INPUT:

- `new_base_ring` – a ring (default: `None`)
- `order` – (optional) an ordering of the support of `self`
- `sparse` – boolean (default: `False`); whether to return a sparse vector or a dense vector

OUTPUT: a *FreeModule()* vector

Warning

This will crash/run forever if `self` is infinite dimensional!

See also

- *vector()*
- CombinatorialFreeModule.get_order()
- CombinatorialFreeModule.from_vector()
- CombinatorialFreeModule._dense_free_module()

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] - 3*B['c']
sage: f._vector_()
(1, 0, -3)
```

```
>>> from sage.all import *
>>> F = CombinatorialFreeModule(QQ, ['a','b','c'])
>>> B = F.basis()
>>> f = B['a'] - Integer(3)*B['c']
>>> f._vector_()
(1, 0, -3)
```

One can use equivalently:

```
sage: f.to_vector()
(1, 0, -3)
sage: vector(f)
(1, 0, -3)
```

```
>>> from sage.all import *
>>> f.to_vector()
(1, 0, -3)
>>> vector(f)
(1, 0, -3)
```

More examples:

```
sage: # needs sage.combinat
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: a = 2*QS3([1,2,3]) + 4*QS3([3,2,1])
sage: a._vector_()
(2, 0, 0, 0, 0, 4)
sage: a.to_vector()
(2, 0, 0, 0, 0, 4)
sage: vector(a)
(2, 0, 0, 0, 0, 4)
sage: a == QS3.from_vector(a.to_vector())
True
sage: a.to_vector(sparse=True)
(2, 0, 0, 0, 0, 4)
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> QS3 = SymmetricGroupAlgebra(QQ, Integer(3))
>>> a = Integer(2)*QS3([Integer(1),Integer(2),Integer(3)]) +␣
↪Integer(4)*QS3([Integer(3),Integer(2),Integer(1)])
>>> a._vector_()
(2, 0, 0, 0, 0, 4)
>>> a.to_vector()
(2, 0, 0, 0, 0, 4)
>>> vector(a)
(2, 0, 0, 0, 0, 4)
>>> a == QS3.from_vector(a.to_vector())
True
>>> a.to_vector(sparse=True)
(2, 0, 0, 0, 0, 4)
```

If `new_base_ring` is specified, then a vector over `new_base_ring` is returned:

```
sage: a._vector_(RDF)                                                    #␣
↪needs sage.combinat
(2.0, 0.0, 0.0, 0.0, 0.0, 4.0)
```

```
>>> from sage.all import *
>>> a._vector_(RDF)                                                      #␣
↪needs sage.combinat
(2.0, 0.0, 0.0, 0.0, 0.0, 4.0)
```

Note

Issue #13406: the current implementation has been optimized, at the price of breaking the encapsulation for FreeModule elements creation, with the following use case as metric, on a 2008' Macbook Pro:

```
   sage: F = CombinatorialFreeModule(QQ, range(10))
   sage: f = F.an_element()
   sage: %timeit f._vector_()   # not tested
   625 loops, best of 3: 17.5 micros per loop

Other use cases may call for different or further
optimizations.
```

3.4 Invariant modules

class sage.modules.with_basis.invariant.**FiniteDimensionalInvariantModule**(*M*, *S*, *action=<built-in function mul>*, *side='left'*, **args*, ***kwargs*)

Bases: *SubmoduleWithBasis*

The invariant submodule under a semigroup action.

When a semigroup S acts on a module M, the invariant module is the set of elements $m \in M$ such that $s \cdot m = m$ for all $s \in S$:

$$M^S := \{m \in M : s \cdot m = m, \forall s \in S\}.$$

INPUT:

- `M` – a module in the category of `FiniteDimensionalModulesWithBasis`
- `S` – a semigroup in the category of `FinitelyGeneratedSemigroups`
- `action` – (default: `operator.mul`) the action of `S` on `M`
- `side` – (default: `'left'`) the side on which `S` acts

EXAMPLES:

First, we create the invariant defined by the cyclic group action on the free module with basis $\{1, 2, 3\}$:

```
sage: G = CyclicPermutationGroup(3)
sage: M = CombinatorialFreeModule(QQ, [1,2,3], prefix='M')
sage: action = lambda g, m: M.monomial(g(m))  # cyclically permute coordinates
```

```
>>> from sage.all import *
>>> G = CyclicPermutationGroup(Integer(3))
>>> M = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3)], prefix='M
↪')
>>> action = lambda g, m: M.monomial(g(m))  # cyclically permute coordinates
```

In order to give the module an action of `G`, we create a *Representation*:

```
sage: from sage.modules.with_basis.representation import Representation
sage: R = Representation(G, M, action)
sage: I = R.invariant_module()
```

```
>>> from sage.all import *
>>> from sage.modules.with_basis.representation import Representation
>>> R = Representation(G, M, action)
>>> I = R.invariant_module()
```

Then we can lift the basis from the invariant to the original module:

```
sage: [I.lift(b) for b in I.basis()]
[M[1] + M[2] + M[3]]
```

```
>>> from sage.all import *
>>> [I.lift(b) for b in I.basis()]
[M[1] + M[2] + M[3]]
```

The we could also have the action be a right-action, instead of the default left-action:

```
sage: def rt_action(g, m): return M.monomial(g(m))  # cyclically permute␣
↪coordinates
sage: R = Representation(G, M, rt_action, side='right')  # same as last but on␣
↪right
sage: g = G.an_element(); g
```

(continues on next page)

(continued from previous page)

```
(1,2,3)
sage: r = R.an_element(); r
2*M[1] + 2*M[2] + 3*M[3]
sage: R.side()
'right'
```

```
>>> from sage.all import *
>>> def rt_action(g, m): return M.monomial(g(m))  # cyclically permute coordinates
>>> R = Representation(G, M, rt_action, side='right')  # same as last but on right
>>> g = G.an_element(); g
(1,2,3)
>>> r = R.an_element(); r
2*M[1] + 2*M[2] + 3*M[3]
>>> R.side()
'right'
```

So now we can see that multiplication with `g` on the right sends `M[1]` to `M[2]` and so on:

```
sage: r * g
3*M[1] + 2*M[2] + 2*M[3]
sage: I = R.invariant_module()
sage: [I.lift(b) for b in I.basis()]
[M[1] + M[2] + M[3]]
```

```
>>> from sage.all import *
>>> r * g
3*M[1] + 2*M[2] + 2*M[3]
>>> I = R.invariant_module()
>>> [I.lift(b) for b in I.basis()]
[M[1] + M[2] + M[3]]
```

Now we will take the regular representation of the symmetric group on three elements to be the module, and compute its invariant submodule:

```
sage: G = SymmetricGroup(3)
sage: R = G.regular_representation(QQ)
sage: I = R.invariant_module()
sage: [I.lift(b).to_vector() for b in I.basis()]
[(1, 1, 1, 1, 1, 1)]
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(3))
>>> R = G.regular_representation(QQ)
>>> I = R.invariant_module()
>>> [I.lift(b).to_vector() for b in I.basis()]
[(1, 1, 1, 1, 1, 1)]
```

We can also check the scalar multiplication by elements of the base ring (for this example, the rational field):

```
sage: [I.lift(3*b).to_vector() for b in I.basis()]
[(3, 3, 3, 3, 3, 3)]
```

```
>>> from sage.all import *
>>> [I.lift(Integer(3)*b).to_vector() for b in I.basis()]
[(3, 3, 3, 3, 3, 3)]
```

A more subtle example is the invariant submodule of a skew-commutative module, for example the exterior algebra $E[x_0, x_1, x_2]$ generated by three elements:

```
sage: G = CyclicPermutationGroup(3)
sage: M = algebras.Exterior(QQ, 'x', 3)
sage: def cyclic_ext_action(g, m):
....:     # cyclically permute generators
....:     return M.prod([M.monomial(FrozenBitset([g(j+1)-1])) for j in m])
```

```
>>> from sage.all import *
>>> G = CyclicPermutationGroup(Integer(3))
>>> M = algebras.Exterior(QQ, 'x', Integer(3))
>>> def cyclic_ext_action(g, m):
...     # cyclically permute generators
...     return M.prod([M.monomial(FrozenBitset([g(j+Integer(1))-Integer(1)])) for␣
↪j in m])
```

If you care about being able to exploit the algebra structure of the exterior algebra (i.e. if you want to multiply elements together), you should make sure the representation knows it is also an algebra with the semigroup action being by algebra endomorphisms:

```
sage: cat = Algebras(QQ).WithBasis().FiniteDimensional()
sage: R = Representation(G, M, cyclic_ext_action, category=cat)
sage: I = R.invariant_module()
```

```
>>> from sage.all import *
>>> cat = Algebras(QQ).WithBasis().FiniteDimensional()
>>> R = Representation(G, M, cyclic_ext_action, category=cat)
>>> I = R.invariant_module()
```

We can express the basis in the ambient algebra ($E[x_0, x_1, x_2]$):

```
sage: [I.lift(b) for b in I.basis()]
[1, x0 + x1 + x2, x0*x1 - x0*x2 + x1*x2, x0*x1*x2]
```

```
>>> from sage.all import *
>>> [I.lift(b) for b in I.basis()]
[1, x0 + x1 + x2, x0*x1 - x0*x2 + x1*x2, x0*x1*x2]
```

or we can express the basis intrinsicallly to the invariant `I`:

```
sage: B = I.basis()
sage: m = 3*B[0] + 2*B[1] + 7*B[3]
```

```
>>> from sage.all import *
>>> B = I.basis()
>>> m = Integer(3)*B[Integer(0)] + Integer(2)*B[Integer(1)] +␣
↪Integer(7)*B[Integer(3)]
```

This lifts to the exterior algebra:

```
sage: I.lift(m)
3 + 2*x0 + 7*x0*x1*x2 + 2*x1 + 2*x2
```

```
>>> from sage.all import *
>>> I.lift(m)
3 + 2*x0 + 7*x0*x1*x2 + 2*x1 + 2*x2
```

We can also check using the invariant element `m` that arithmetic works:

```
sage: m^2
9*B[0] + 12*B[1] + 42*B[3]
sage: m+m
6*B[0] + 4*B[1] + 14*B[3]
```

```
>>> from sage.all import *
>>> m**Integer(2)
9*B[0] + 12*B[1] + 42*B[3]
>>> m+m
6*B[0] + 4*B[1] + 14*B[3]
```

To see the actual elements expressed in the exterior algebra, we lift them again:

```
sage: I.lift(m+m)
6 + 4*x0 + 14*x0*x1*x2 + 4*x1 + 4*x2
sage: 7*m
21*B[0] + 14*B[1] + 49*B[3]
sage: I.lift(7*m)
21 + 14*x0 + 49*x0*x1*x2 + 14*x1 + 14*x2
```

```
>>> from sage.all import *
>>> I.lift(m+m)
6 + 4*x0 + 14*x0*x1*x2 + 4*x1 + 4*x2
>>> Integer(7)*m
21*B[0] + 14*B[1] + 49*B[3]
>>> I.lift(Integer(7)*m)
21 + 14*x0 + 49*x0*x1*x2 + 14*x1 + 14*x2
```

The classic example of an invariant module is the module of symmetric functions, which is the invariant module of polynomials whose variables are acted upon by permutation. We can create a module isomorphic to the homogeneous component of a a polynomial ring in n variable of a fixed degree d by looking at weak compositions of d of length n, which we consider as the exponent vector. For example, $x^2yz \in \mathbf{Q}[x, y, z]$ would have the exponent vector $(2, 1, 1)$. The vector $(2, 1, 1)$ is a weak composition of 4, with length 3, and so we can think of it as being in the degree-4 homogeneous component of a polynomial ring in three variables:

```
sage: C = IntegerVectors(4, length=3, min_part=0)  # representing degree-4␣
↪monomials
sage: M = CombinatorialFreeModule(QQ, C)  # isomorphic to deg-4 homog. polynomials
sage: G = SymmetricGroup(3)
sage: def perm_action(g, x): return M.monomial(C(g(list(x))))
sage: perm_action(G((1,2,3)), C([4,3,2]))
B[[3, 2, 4]]
```

(continues on next page)

(continued from previous page)

```
sage: R = Representation(G, M, perm_action)
sage: I = R.invariant_module()
sage: [I.lift(b) for b in I.basis()]
[B[[0, 0, 4]] + B[[0, 4, 0]] + B[[4, 0, 0]],
 B[[0, 1, 3]] + B[[0, 3, 1]] + B[[1, 0, 3]]
 + B[[1, 3, 0]] + B[[3, 0, 1]] + B[[3, 1, 0]],
 B[[0, 2, 2]] + B[[2, 0, 2]] + B[[2, 2, 0]],
 B[[1, 1, 2]] + B[[1, 2, 1]] + B[[2, 1, 1]]]
```

```
>>> from sage.all import *
>>> C = IntegerVectors(Integer(4), length=Integer(3), min_part=Integer(0))  #␣
↪representing degree-4 monomials
>>> M = CombinatorialFreeModule(QQ, C)  # isomorphic to deg-4 homog. polynomials
>>> G = SymmetricGroup(Integer(3))
>>> def perm_action(g, x): return M.monomial(C(g(list(x))))
>>> perm_action(G((Integer(1),Integer(2),Integer(3))), C([Integer(4),Integer(3),
↪Integer(2)]))
B[[3, 2, 4]]
>>> R = Representation(G, M, perm_action)
>>> I = R.invariant_module()
>>> [I.lift(b) for b in I.basis()]
[B[[0, 0, 4]] + B[[0, 4, 0]] + B[[4, 0, 0]],
 B[[0, 1, 3]] + B[[0, 3, 1]] + B[[1, 0, 3]]
 + B[[1, 3, 0]] + B[[3, 0, 1]] + B[[3, 1, 0]],
 B[[0, 2, 2]] + B[[2, 0, 2]] + B[[2, 2, 0]],
 B[[1, 1, 2]] + B[[1, 2, 1]] + B[[2, 1, 1]]]
```

These are the monomial symmetric functions, which are a well-known basis for the symmetric functions. For comparison:

```
sage: Sym = SymmetricFunctions(QQ)
sage: m = Sym.monomial()
sage: [m[mu].expand(3) for mu in Partitions(4)]
[x0^4 + x1^4 + x2^4,
 x0^3*x1 + x0*x1^3 + x0^3*x2 + x1^3*x2 + x0*x2^3 + x1*x2^3,
 x0^2*x1^2 + x0^2*x2^2 + x1^2*x2^2,
 x0^2*x1*x2 + x0*x1^2*x2 + x0*x1*x2^2,
 0]
```

```
>>> from sage.all import *
>>> Sym = SymmetricFunctions(QQ)
>>> m = Sym.monomial()
>>> [m[mu].expand(Integer(3)) for mu in Partitions(Integer(4))]
[x0^4 + x1^4 + x2^4,
 x0^3*x1 + x0*x1^3 + x0^3*x2 + x1^3*x2 + x0*x2^3 + x1*x2^3,
 x0^2*x1^2 + x0^2*x2^2 + x1^2*x2^2,
 x0^2*x1*x2 + x0*x1^2*x2 + x0*x1*x2^2,
 0]
```

Note

The current implementation works when S is a finitely-generated semigroup, and when M is a finite-dimensional free module with a distinguished basis.

Todo

Extend this to have multiple actions, including actions on both sides.

Todo

Extend when M does not have a basis and S is a permutation group using:

- arXiv 0812.3082
- https://www.dmtcs.org/pdfpapers/dmAA0123.pdf

class Element

Bases: *IndexedFreeModuleElement*

construction()

Return the functorial construction of `self`.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(3)
sage: R = G.regular_representation(); R
Left Regular Representation of Cyclic group of order 3 as a permutation group␣
↪over Integer Ring
sage: I = R.invariant_module()
sage: I.construction()
(EquivariantSubobjectConstructionFunctor,
Left Regular Representation of Cyclic group of order 3 as a permutation group␣
↪over Integer Ring)
```

```
>>> from sage.all import *
>>> G = CyclicPermutationGroup(Integer(3))
>>> R = G.regular_representation(); R
Left Regular Representation of Cyclic group of order 3 as a permutation group␣
↪over Integer Ring
>>> I = R.invariant_module()
>>> I.construction()
(EquivariantSubobjectConstructionFunctor,
Left Regular Representation of Cyclic group of order 3 as a permutation group␣
↪over Integer Ring)
```

semigroup()

Return the semigroup S whose action `self` is invariant under.

EXAMPLES:

```
sage: G = SymmetricGroup(3)
sage: M = CombinatorialFreeModule(QQ, [1,2,3], prefix='M')
```

(continues on next page)

(continued from previous page)

```
sage: def action(g, x): return M.monomial(g(x))
sage: I = M.invariant_module(G, action_on_basis=action)
sage: I.semigroup()
Symmetric group of order 3! as a permutation group
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(3))
>>> M = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3)],␣
↪prefix='M')
>>> def action(g, x): return M.monomial(g(x))
>>> I = M.invariant_module(G, action_on_basis=action)
>>> I.semigroup()
Symmetric group of order 3! as a permutation group
```

semigroup_representation()

Return the ambient space of `self`.

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, range(3)); x = X.basis()
sage: Y = X.submodule((x[0]-x[1], x[1]-x[2]))
sage: Y.ambient() is X
True
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, range(Integer(3))); x = X.basis()
>>> Y = X.submodule((x[Integer(0)]-x[Integer(1)], x[Integer(1)]-
↪x[Integer(2)]))
>>> Y.ambient() is X
True
```

class sage.modules.with_basis.invariant.**FiniteDimensionalTwistedInvariantModule**(*M*, *G*, *chi*, *action=<built-in function mul>*, *side='left'*, ***kwargs*)

Bases: *SubmoduleWithBasis*

Construct the χ-twisted invariant submodule of M.

When a group G acts on a module M, the *χ-twisted invariant submodule* of M is the isotypic component of the representation M corresponding to the irreducible character χ.

For more information, see [Sta1979].

INPUT:

- `M` – a module in the category of `FiniteDimensionalModulesWithBasis` and whose base ring contains all the values passed to `chi` and $1/|G|$
- `G` – a finitely generated group
- `chi` – list/tuple of the character values of the irreducible representation onto which you want to project. The order of values of *chi* must agree with the order of `G.conjugacy_classes()`

- `action` – (default: `operator.mul`) the action of `G` on `M`
- `side` – (default: `'left'`) the side on which `G` acts

Warning

The current implementation does not check if `chi` is irreducible. Passing character values of non-irreducible representations may lead to mathematically incorrect results.

EXAMPLES:

Suppose that the symmetric group S_3 acts on a four dimensional vector space by permuting the first three coordinates only:

```
sage: M = CombinatorialFreeModule(QQ, [1,2,3,4], prefix='M')
sage: G = SymmetricGroup(3)
sage: action = lambda g,x: M.term(g(x))
```

```
>>> from sage.all import *
>>> M = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3),Integer(4)],
↪ prefix='M')
>>> G = SymmetricGroup(Integer(3))
>>> action = lambda g,x: M.term(g(x))
```

The trivial representation corresponds to the usual invariant module, so trying to create the twisted invariant module when there is no twist returns a *FiniteDimensionalInvariantModule*:

```
sage: chi = ClassFunction(G, (1,1,1))
sage: T = M.twisted_invariant_module(G, chi, action_on_basis=action)
sage: type(T)
<class 'sage.modules.with_basis.invariant.FiniteDimensionalInvariantModule_with_
↪category'>
```

```
>>> from sage.all import *
>>> chi = ClassFunction(G, (Integer(1),Integer(1),Integer(1)))
>>> T = M.twisted_invariant_module(G, chi, action_on_basis=action)
>>> type(T)
<class 'sage.modules.with_basis.invariant.FiniteDimensionalInvariantModule_with_
↪category'>
```

In this case, there are two copies of the trivial representation, one coming from the first three coordinates and the other coming from the fact that S_3 does not touch the fourth coordinate:

```
sage: T.basis()
Finite family {0: B[0], 1: B[1]}
sage: [T.lift(b) for b in T.basis()]
[M[1] + M[2] + M[3], M[4]]
```

```
>>> from sage.all import *
>>> T.basis()
Finite family {0: B[0], 1: B[1]}
>>> [T.lift(b) for b in T.basis()]
[M[1] + M[2] + M[3], M[4]]
```

The character values of the standard representation are $2, 0, -1$:

```
sage: chi = ClassFunction(G, [2,0,-1])
sage: T = M.twisted_invariant_module(G, chi, action_on_basis=action)
sage: type(T)
<class 'sage.modules.with_basis.invariant.FiniteDimensionalTwistedInvariantModule_
↪with_category'>
sage: T.basis()
Finite family {0: B[0], 1: B[1]}
sage: [T.lift(b) for b in T.basis()]
[M[1] - M[3], M[2] - M[3]]
```

```
>>> from sage.all import *
>>> chi = ClassFunction(G, [Integer(2),Integer(0),-Integer(1)])
>>> T = M.twisted_invariant_module(G, chi, action_on_basis=action)
>>> type(T)
<class 'sage.modules.with_basis.invariant.FiniteDimensionalTwistedInvariantModule_
↪with_category'>
>>> T.basis()
Finite family {0: B[0], 1: B[1]}
>>> [T.lift(b) for b in T.basis()]
[M[1] - M[3], M[2] - M[3]]
```

The permutation representation is the direct sum of the standard representation with the trivial representation, and the action on the basis element `B[4]` is itself a copy of the trivial representation, so the sign representation does not appear in the decomposition:

```
sage: T = M.twisted_invariant_module(G, [1,-1,1], action_on_basis=action)
sage: T.basis()
Finite family {}
```

```
>>> from sage.all import *
>>> T = M.twisted_invariant_module(G, [Integer(1),-Integer(1),Integer(1)], action_
↪on_basis=action)
>>> T.basis()
Finite family {}
```

We can also get two copies of the standard representation by looking at two copies of the permutation representation, found by reduction modulo three on the indices of a six-dimensional module:

```
sage: M = CombinatorialFreeModule(QQ, [0,1,2,3,4,5], prefix='M')
sage: action = lambda g,x: M.term(g(x%3 + 1)-1 + (x>=3)*3)
sage: T = M.twisted_invariant_module(G, [2,0,-1], action_on_basis=action)
sage: T.basis()
Finite family {0: B[0], 1: B[1], 2: B[2], 3: B[3]}
sage: [T.lift(b) for b in T.basis()]
[M[0] - M[2], M[1] - M[2], M[3] - M[5], M[4] - M[5]]

sage: T = M.twisted_invariant_module(G, [1,1,1], action_on_basis=action)
sage: T.basis()
Finite family {0: B[0], 1: B[1]}
sage: [T.lift(b) for b in T.basis()]
[M[0] + M[1] + M[2], M[3] + M[4] + M[5]]
```

```
>>> from sage.all import *
>>> M = CombinatorialFreeModule(QQ, [Integer(0),Integer(1),Integer(2),Integer(3),
↪Integer(4),Integer(5)], prefix='M')
>>> action = lambda g,x: M.term(g(x%Integer(3) + Integer(1))-Integer(1) + (x>
↪=Integer(3))*Integer(3))
>>> T = M.twisted_invariant_module(G, [Integer(2),Integer(0),-Integer(1)], action_
↪on_basis=action)
>>> T.basis()
Finite family {0: B[0], 1: B[1], 2: B[2], 3: B[3]}
>>> [T.lift(b) for b in T.basis()]
[M[0] - M[2], M[1] - M[2], M[3] - M[5], M[4] - M[5]]

>>> T = M.twisted_invariant_module(G, [Integer(1),Integer(1),Integer(1)], action_
↪on_basis=action)
>>> T.basis()
Finite family {0: B[0], 1: B[1]}
>>> [T.lift(b) for b in T.basis()]
[M[0] + M[1] + M[2], M[3] + M[4] + M[5]]
```

There are still no copies of the sign representation:

```
sage: T = M.twisted_invariant_module(G, [1,-1,1], action_on_basis=action)
sage: T.basis()
Finite family {}
```

```
>>> from sage.all import *
>>> T = M.twisted_invariant_module(G, [Integer(1),-Integer(1),Integer(1)], action_
↪on_basis=action)
>>> T.basis()
Finite family {}
```

The trivial representation also contains no copies of the sign representation:

```
sage: R = G.trivial_representation(QQ)
sage: T = R.twisted_invariant_module([1,-1,1])
sage: T.basis()
Finite family {}
```

```
>>> from sage.all import *
>>> R = G.trivial_representation(QQ)
>>> T = R.twisted_invariant_module([Integer(1),-Integer(1),Integer(1)])
>>> T.basis()
Finite family {}
```

The regular representation contains two copies of the standard representation and one copy each of the trivial and the sign:

```
sage: R = G.regular_representation(QQ)
sage: std = R.twisted_invariant_module([2,0,-1])
sage: std.basis()
Finite family {0: B[0], 1: B[1], 2: B[2], 3: B[3]}
sage: [std.lift(b) for b in std.basis()]
[() - (1,2,3), -(1,2,3) + (1,3,2), (2,3) - (1,2), -(1,2) + (1,3)]
```

(continues on next page)

(continued from previous page)

```
sage: triv = R.twisted_invariant_module([1,1,1])
sage: triv.basis()
Finite family {0: B[0]}
sage: [triv.lift(b) for b in triv.basis()]
[() + (2,3) + (1,2) + (1,2,3) + (1,3,2) + (1,3)]

sage: sgn = R.twisted_invariant_module([1,-1,1])
sage: sgn.basis()
Finite family {0: B[0]}
sage: [sgn.lift(b) for b in sgn.basis()]
[() - (2,3) - (1,2) + (1,2,3) + (1,3,2) - (1,3)]
```

```
>>> from sage.all import *
>>> R = G.regular_representation(QQ)
>>> std = R.twisted_invariant_module([Integer(2),Integer(0),-Integer(1)])
>>> std.basis()
Finite family {0: B[0], 1: B[1], 2: B[2], 3: B[3]}
>>> [std.lift(b) for b in std.basis()]
[() - (1,2,3), -(1,2,3) + (1,3,2), (2,3) - (1,2), -(1,2) + (1,3)]

>>> triv = R.twisted_invariant_module([Integer(1),Integer(1),Integer(1)])
>>> triv.basis()
Finite family {0: B[0]}
>>> [triv.lift(b) for b in triv.basis()]
[() + (2,3) + (1,2) + (1,2,3) + (1,3,2) + (1,3)]

>>> sgn = R.twisted_invariant_module([Integer(1),-Integer(1),Integer(1)])
>>> sgn.basis()
Finite family {0: B[0]}
>>> [sgn.lift(b) for b in sgn.basis()]
[() - (2,3) - (1,2) + (1,2,3) + (1,3,2) - (1,3)]
```

For the next example, we construct a twisted invariant by the character for the 2 dimensional representation of S_3 on the natural action on the exterior algebra. While S_3 acts by automorphisms, the twisted invariants do not form an algebra in this case:

```
sage: G = SymmetricGroup(3); G.rename('S3')
sage: E = algebras.Exterior(QQ, 'x', 3); E.rename('E')
sage: def action(g, m): return E.prod([E.monomial(FrozenBitset([g(j+1)-1])) for j␣
↪in m])
sage: from sage.modules.with_basis.representation import Representation
sage: EA = Representation(G, E, action, category=Algebras(QQ).WithBasis().
↪FiniteDimensional())
sage: T = EA.twisted_invariant_module([2,0,-1])
sage: t = T.an_element(); t
2*B[0] + 2*B[1] + 3*B[2]
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(3)); G.rename('S3')
>>> E = algebras.Exterior(QQ, 'x', Integer(3)); E.rename('E')
>>> def action(g, m): return E.prod([E.monomial(FrozenBitset([g(j+Integer(1))-
```

(continues on next page)

(continued from previous page)

```
↪Integer(1)])) for j in m])
>>> from sage.modules.with_basis.representation import Representation
>>> EA = Representation(G, E, action, category=Algebras(QQ).WithBasis().
↪FiniteDimensional())
>>> T = EA.twisted_invariant_module([Integer(2),Integer(0),-Integer(1)])
>>> t = T.an_element(); t
2*B[0] + 2*B[1] + 3*B[2]
```

We can still get meaningful information about the product by taking the product in the ambient space:

```
sage: T.lift(t) * T.lift(t)
-36*x0*x1*x2
```

```
>>> from sage.all import *
>>> T.lift(t) * T.lift(t)
-36*x0*x1*x2
```

We can see this does not lie in this twisted invariant algebra:

```
sage: T.retract(T.lift(t) * T.lift(t))
Traceback (most recent call last):
...
ValueError: -36*x0*x1*x2 is not in the image

sage: [T.lift(b) for b in T.basis()]
[x0 - x2, x1 - x2, x0*x1 - x1*x2, x0*x2 + x1*x2]
```

```
>>> from sage.all import *
>>> T.retract(T.lift(t) * T.lift(t))
Traceback (most recent call last):
...
ValueError: -36*x0*x1*x2 is not in the image

>>> [T.lift(b) for b in T.basis()]
[x0 - x2, x1 - x2, x0*x1 - x1*x2, x0*x2 + x1*x2]
```

It happens to be in the trivial isotypic component (equivalently in the usual invariant algebra) but Sage does not know this.

```
sage: G.rename(); E.rename()  # reset the names
```

```
>>> from sage.all import *
>>> G.rename(); E.rename()  # reset the names
```

Todo

- Replace `G` by `S` in `FinitelyGeneratedSemigroups`
- Allow for `chi` to be a *Representation*
- Add check for irreducibility of `chi`

class Element

Bases: *IndexedFreeModuleElement*

project(*x*)

Project `x` in the ambient module onto `self`.

EXAMPLES:

The standard representation is the orthogonal complement of the trivial representation inside of the permutation representation, so the basis for the trivial representation projects to 0:

```
sage: M = CombinatorialFreeModule(QQ, [1,2,3]); M.rename('M')
sage: B = M.basis()
sage: G = SymmetricGroup(3); G.rename('S3')
sage: def action(g, x): return M.term(g(x))
sage: T = M.twisted_invariant_module(G, [2,0,-1], action_on_basis=action)
sage: m = B[1] + B[2] + B[3]
sage: parent(m)
M
sage: t = T.project(m); t
0
sage: parent(t)
Twist of (S3)-invariant submodule of M by character [2, 0, -1]

sage: G.rename(); M.rename()  # reset names
```

```
>>> from sage.all import *
>>> M = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3)]); M.
↪rename('M')
>>> B = M.basis()
>>> G = SymmetricGroup(Integer(3)); G.rename('S3')
>>> def action(g, x): return M.term(g(x))
>>> T = M.twisted_invariant_module(G, [Integer(2),Integer(0),-Integer(1)],␣
↪action_on_basis=action)
>>> m = B[Integer(1)] + B[Integer(2)] + B[Integer(3)]
>>> parent(m)
M
>>> t = T.project(m); t
0
>>> parent(t)
Twist of (S3)-invariant submodule of M by character [2, 0, -1]

>>> G.rename(); M.rename()  # reset names
```

project_ambient(*x*)

Project `x` in the ambient representation onto the submodule of the ambient representation to which `self` is isomorphic as a module.

Note

The image of `self.project_ambient` is not in `self` but rather is in `self.ambient()`.

EXAMPLES:

```
sage: M = CombinatorialFreeModule(QQ, [1,2,3]); M.rename('M')
sage: B = M.basis()
sage: G = SymmetricGroup(3); G.rename('S3')
sage: def action(g, x): return M.term(g(x))
sage: T = M.twisted_invariant_module(G, [2,0,-1], action_on_basis=action)
```

```
>>> from sage.all import *
>>> M = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3)]); M.
↪rename('M')
>>> B = M.basis()
>>> G = SymmetricGroup(Integer(3)); G.rename('S3')
>>> def action(g, x): return M.term(g(x))
>>> T = M.twisted_invariant_module(G, [Integer(2),Integer(0),-Integer(1)],␣
↪action_on_basis=action)
```

To compare with `self.project`, we can inspect the parents. The image of `self.project` is in `self`, while the image of `self.project_ambient` is in `self._ambient`:

```
sage: t = T.project(B[1] + B[2] + B[3]); t
0
sage: parent(t)
Twist of (S3)-invariant submodule of M by character [2, 0, -1]
sage: s = T.project_ambient(B[1] + B[2] + B[3]); s
0
sage: parent(s)
Representation of S3 indexed by {1, 2, 3} over Rational Field
```

```
>>> from sage.all import *
>>> t = T.project(B[Integer(1)] + B[Integer(2)] + B[Integer(3)]); t
0
>>> parent(t)
Twist of (S3)-invariant submodule of M by character [2, 0, -1]
>>> s = T.project_ambient(B[Integer(1)] + B[Integer(2)] + B[Integer(3)]); s
0
>>> parent(s)
Representation of S3 indexed by {1, 2, 3} over Rational Field
```

Note that because of the construction of `T`, `self._ambient` is an instance of *Representation*, but you still may pass elements of `M`, which is an instance of `CombinatorialFreeModule`, because the underlying `Representation` is built off of `M` and we can cannonically construct elements of the `Representation` from elements of `M`.

```
sage: G.rename(); M.rename()  # reset names
```

```
>>> from sage.all import *
>>> G.rename(); M.rename()  # reset names
```

projection_matrix()

Return the matrix defining the projection map from the ambient representation onto `self`.

EXAMPLES:

```
sage: M = CombinatorialFreeModule(QQ, [1,2,3])
sage: def action(g, x): return(M.term(g(x)))
sage: G = SymmetricGroup(3)
```

```
>>> from sage.all import *
>>> M = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3)])
>>> def action(g, x): return(M.term(g(x)))
>>> G = SymmetricGroup(Integer(3))
```

If the matrix A has columns form a basis for the subspace onto which we are trying to project, then we can find the projection matrix via the formula $P = A(A^TA)^{-1}A^T$. Recall that the standard representation twisted invariant has basis `(B[1] - B[3], B[2] - B[3])`, hence:

```
sage: A = Matrix([[1,0],[0,1],[-1,-1]])
sage: P = A*(A.transpose()*A).inverse()*A.transpose()
sage: T = M.twisted_invariant_module(G, [2,0,-1], action_on_basis=action)
sage: P == T.projection_matrix()
True
```

```
>>> from sage.all import *
>>> A = Matrix([[Integer(1),Integer(0)],[Integer(0),Integer(1)],[-Integer(1),-
↪Integer(1)]])
>>> P = A*(A.transpose()*A).inverse()*A.transpose()
>>> T = M.twisted_invariant_module(G, [Integer(2),Integer(0),-Integer(1)],␣
↪action_on_basis=action)
>>> P == T.projection_matrix()
True
```

Moreover, since there is no component of the sign representation in this representation, the projection matrix is just the zero matrix:

```
sage: T = M.twisted_invariant_module(G, [1,-1,1], action_on_basis=action)
sage: T.projection_matrix()
[0 0 0]
[0 0 0]
[0 0 0]
```

```
>>> from sage.all import *
>>> T = M.twisted_invariant_module(G, [Integer(1),-Integer(1),Integer(1)],␣
↪action_on_basis=action)
>>> T.projection_matrix()
[0 0 0]
[0 0 0]
[0 0 0]
```

3.5 Morphisms of modules with a basis

This module contains a hierarchy of classes for morphisms of modules with a basis (category `Modules.WithBasis`):

- *ModuleMorphism*
- *ModuleMorphismByLinearity*

- *ModuleMorphismFromMatrix*
- *ModuleMorphismFromFunction*
- *TriangularModuleMorphism*
- *TriangularModuleMorphismByLinearity*
- *TriangularModuleMorphismFromFunction*

These are internal classes; it is recommended *not* to use them directly, and instead to construct morphisms through the ModulesWithBasis.ParentMethods.module_morphism() method of the domain, or through the homset. See the former for an overview of the possible arguments.

EXAMPLES:

We construct a morphism through the method ModulesWithBasis.ParentMethods.module_morphism(), by specifying the image of each element of the distinguished basis:

```
sage: X = CombinatorialFreeModule(QQ, [1,2,3]);   x = X.basis()
sage: Y = CombinatorialFreeModule(QQ, [1,2,3,4]); y = Y.basis()
sage: on_basis = lambda i: Y.monomial(i) + 2*Y.monomial(i+1)

sage: phi1 = X.module_morphism(on_basis, codomain=Y)
sage: phi1(x[1])
B[1] + 2*B[2]

sage: phi1
Generic morphism:
  From: Free module generated by {1, 2, 3} over Rational Field
  To:   Free module generated by {1, 2, 3, 4} over Rational Field
sage: phi1.parent()
Set of Morphisms from Free module generated by {1, 2, 3} over Rational Field to Free␣
↪module generated by {1, 2, 3, 4} over Rational Field in Category of finite␣
↪dimensional vector spaces with basis over Rational Field
sage: phi1.__class__
<class 'sage.modules.with_basis.morphism.ModuleMorphismByLinearity_with_category'>
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3)]);   x = X.
↪basis()
>>> Y = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3),Integer(4)]); y␣
↪= Y.basis()
>>> on_basis = lambda i: Y.monomial(i) + Integer(2)*Y.monomial(i+Integer(1))

>>> phi1 = X.module_morphism(on_basis, codomain=Y)
>>> phi1(x[Integer(1)])
B[1] + 2*B[2]

>>> phi1
Generic morphism:
  From: Free module generated by {1, 2, 3} over Rational Field
  To:   Free module generated by {1, 2, 3, 4} over Rational Field
>>> phi1.parent()
Set of Morphisms from Free module generated by {1, 2, 3} over Rational Field to Free␣
↪module generated by {1, 2, 3, 4} over Rational Field in Category of finite␣
↪dimensional vector spaces with basis over Rational Field
```

(continues on next page)

(continued from previous page)

```
>>> phi1.__class__
<class 'sage.modules.with_basis.morphism.ModuleMorphismByLinearity_with_category'>
```

Constructing the same morphism from the homset:

```
sage: H = Hom(X,Y)
sage: phi2 = H(on_basis=on_basis)
sage: phi1 == phi2
True
```

```
>>> from sage.all import *
>>> H = Hom(X,Y)
>>> phi2 = H(on_basis=on_basis)
>>> phi1 == phi2
True
```

Constructing the same morphism directly using the class; no backward compatibility is guaranteed in this case:

```
sage: from sage.modules.with_basis.morphism import ModuleMorphismByLinearity
sage: phi3 = ModuleMorphismByLinearity(X, on_basis, codomain=Y)
sage: phi3 == phi1
True
```

```
>>> from sage.all import *
>>> from sage.modules.with_basis.morphism import ModuleMorphismByLinearity
>>> phi3 = ModuleMorphismByLinearity(X, on_basis, codomain=Y)
>>> phi3 == phi1
True
```

Warning

The hierarchy of classes implemented in this module is one of the first non-trivial hierarchies of classes for morphisms. It is hitting a couple scaling issues:

- There are many independent properties from which module morphisms can get code (being defined by linearity, from a matrix, or a function; being triangular, being diagonal, …). How to mitigate the class hierarchy growth?

  This will become even more stringent as more properties are added (e.g. being defined from generators for an algebra morphism, …)

  Categories, whose primary purpose is to provide infrastructure for handling such large hierarchy of classes, can't help at this point: there is no category whose morphisms are triangular morphisms, and it's not clear such a category would be sensible.

- How to properly handle `__init__` method calls and multiple inheritance?

- Who should be in charge of setting the default category: the classes themselves, or `ModulesWithBasis.ParentMethods.module_morphism()`?

Because of this, the hierarchy of classes, and the specific APIs, is likely to be refactored as better infrastructure and best practices emerge.

AUTHORS:

- Nicolas M. Thiery (2008-2015)

- Jason Bandlow and Florent Hivert (2010): Triangular Morphisms
- Christian Stump (2010): Issue #9648 module_morphism's to a wider class of codomains

Before Issue #8678, this hierarchy of classes used to be in sage.categories.modules_with_basis; see Issue #8678 for the complete log.

class sage.modules.with_basis.morphism.**DiagonalModuleMorphism**(*domain*, *diagonal*, *codomain=None*, *category=None*)

Bases: *ModuleMorphismByLinearity*

A class for diagonal module morphisms.

See ModulesWithBasis.ParentMethods.module_morphism().

INPUT:

- domain, codomain – two modules with basis F and G, respectively
- diagonal – a function d

Assumptions:

- domain and codomain have the same base ring R,
- their respective bases F and G have the same index set I,
- d is a function $I \to R$.

Return the diagonal module morphism from domain to codomain sending $F(i) \mapsto d(i)G(i)$ for all $i \in I$.

By default, codomain is currently assumed to be domain. (Todo: make a consistent choice with *ModuleMorphism.)

Todo

- Implement an optimized _call_() function.
- Generalize to a mapcoeffs.
- Generalize to a mapterms.

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename('X')
sage: phi = X.module_morphism(diagonal=factorial, codomain=X)
sage: x = X.basis()
sage: phi(x[1]), phi(x[2]), phi(x[3])
(B[1], 2*B[2], 6*B[3])
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, [Integer(1), Integer(2), Integer(3)]); X.
↪rename('X')
>>> phi = X.module_morphism(diagonal=factorial, codomain=X)
>>> x = X.basis()
>>> phi(x[Integer(1)]), phi(x[Integer(2)]), phi(x[Integer(3)])
(B[1], 2*B[2], 6*B[3])
```

class sage.modules.with_basis.morphism.**ModuleMorphism**(*domain*, *codomain=None*, *category=None*, *affine=False*)

Bases: Morphism

The top abstract base class for module with basis morphisms.

INPUT:

- domain – a parent in ModulesWithBasis(...)
- codomain – a parent in Modules(...)
- category – a category or None (default: None)
- affine – whether we define an affine module morphism (default: False)

Construct a module morphism from domain to codomain in the category category. By default, the category is the first of Modules(R).WithBasis().FiniteDimensional(), Modules(R).WithBasis(), Modules(R), CommutativeAdditiveMonoids() that contains both the domain and the codomain. If initializing an affine morphism, then $Sets()$ is used instead.

See also

- ModulesWithBasis.ParentMethods.module_morphism() for usage information and examples;
- *sage.modules.with_basis.morphism* for a technical overview of the classes for module morphisms;
- *ModuleMorphismFromFunction* and *TriangularModuleMorphism*.

The role of this class is minimal: it provides an __init__() method which:

- handles the choice of the default category
- handles the proper inheritance from categories by updating the class of self upon construction.

class sage.modules.with_basis.morphism.**ModuleMorphismByLinearity**(*domain*, *on_basis=None*, *codomain=None*, *category=None*, *position=0*, *zero=None*)

Bases: *ModuleMorphism*

A class for module morphisms obtained by extending a function by linearity.

INPUT:

- domain, codomain, category – as for *ModuleMorphism*
- on_basis – a function which accepts indices of the basis of domain as position-th argument
- **codomain – a parent in Modules(...)**
  (default: on_basis.codomain())
- position – nonnegative integer (default: 0)
- zero – the zero of the codomain (defaults: codomain.zero())

See also

- ModulesWithBasis.ParentMethods.module_morphism() for usage information and examples;

- *sage.modules.with_basis.morphism* for a technical overview of the classes for module morphisms;
- *ModuleMorphismFromFunction* and *TriangularModuleMorphism*.

Note

`on_basis` may alternatively be provided in derived classes by passing `None` as argument, and implementing or setting the attribute `_on_basis`

on_basis()

Return the action of this morphism on basis elements, as per `ModulesWithBasis.Homsets.ElementMethods.on_basis()`.

OUTPUT:

- a function from the indices of the basis of the domain to the codomain

EXAMPLES:

```
sage: X = CombinatorialFreeModule(ZZ, [-2, -1, 1, 2])
sage: Y = CombinatorialFreeModule(ZZ, [1, 2])
sage: phi_on_basis = Y.monomial * abs
sage: phi = sage.modules.with_basis.morphism.ModuleMorphismByLinearity(X, on_
↪basis = phi_on_basis, codomain=Y)
sage: x = X.basis()
sage: phi.on_basis()(-2)
B[2]
sage: phi.on_basis() == phi_on_basis
True
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(ZZ, [-Integer(2), -Integer(1), Integer(1),␣
↪Integer(2)])
>>> Y = CombinatorialFreeModule(ZZ, [Integer(1), Integer(2)])
>>> phi_on_basis = Y.monomial * abs
>>> phi = sage.modules.with_basis.morphism.ModuleMorphismByLinearity(X, on_
↪basis = phi_on_basis, codomain=Y)
>>> x = X.basis()
>>> phi.on_basis()(-Integer(2))
B[2]
>>> phi.on_basis() == phi_on_basis
True
```

class sage.modules.with_basis.morphism.**ModuleMorphismFromFunction**(*domain*, *function*, *codomain=None*, *category=None*)

Bases: *ModuleMorphism*, `SetMorphism`

A class for module morphisms implemented by a plain function.

INPUT:

- `domain`, `codomain`, `category` – as for *ModuleMorphism*

- `function` – any function or callable from domain to codomain

See also

- `ModulesWithBasis.ParentMethods.module_morphism()` for usage information and examples;
- *`sage.modules.with_basis.morphism`* for a technical overview of the classes for module morphisms;
- *`ModuleMorphismFromFunction`* and *`TriangularModuleMorphism`*.

class sage.modules.with_basis.morphism.**ModuleMorphismFromMatrix**(*domain*, *matrix*, *codomain=None*, *category=None*, *side='left'*)

Bases: *`ModuleMorphismByLinearity`*

A class for module morphisms built from a matrix in the distinguished bases of the domain and codomain.

See also

- `ModulesWithBasis.ParentMethods.module_morphism()`
- `ModulesWithBasis.FiniteDimensional.MorphismMethods.matrix()`

INPUT:

- `domain`, `codomain` – two finite dimensional modules over the same base ring R with basis F and G, respectively
- `matrix` – a matrix with base ring R and dimensions matching that of F and G, respectively
- `side` – `'left'` or `'right'` (default: `'left'`)

  If `side` is "left", this morphism is considered as acting on the left; i.e. each column of the matrix represents the image of an element of the basis of the domain.
- `category` – a category or `None` (default: `None`)

EXAMPLES:

```
sage: X = CombinatorialFreeModule(ZZ, [1,2]); X.rename('X'); x = X.basis()
sage: Y = CombinatorialFreeModule(ZZ, [3,4]); Y.rename('Y'); y = Y.basis()
sage: m = matrix([[1,2],[3,5]])
sage: phi = X.module_morphism(matrix=m, codomain=Y)
sage: phi.parent()
Set of Morphisms from X to Y in Category of finite dimensional modules with basis␣
↪over Integer Ring
sage: phi.__class__
<class 'sage.modules.with_basis.morphism.ModuleMorphismFromMatrix_with_category'>
sage: phi(x[1])
B[3] + 3*B[4]
sage: phi(x[2])
2*B[3] + 5*B[4]

sage: m = matrix([[1,2],[3,5]])
```

(continues on next page)

(continued from previous page)

```
sage: phi = X.module_morphism(matrix=m, codomain=Y, side='right',
....:                         category=Modules(ZZ).WithBasis())
sage: phi.parent()
Set of Morphisms from X to Y
in Category of modules with basis over Integer Ring
sage: phi(x[1])
B[3] + 2*B[4]
sage: phi(x[2])
3*B[3] + 5*B[4]
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(ZZ, [Integer(1),Integer(2)]); X.rename('X'); x =␣
↪X.basis()
>>> Y = CombinatorialFreeModule(ZZ, [Integer(3),Integer(4)]); Y.rename('Y'); y =␣
↪Y.basis()
>>> m = matrix([[Integer(1),Integer(2)],[Integer(3),Integer(5)]])
>>> phi = X.module_morphism(matrix=m, codomain=Y)
>>> phi.parent()
Set of Morphisms from X to Y in Category of finite dimensional modules with basis␣
↪over Integer Ring
>>> phi.__class__
<class 'sage.modules.with_basis.morphism.ModuleMorphismFromMatrix_with_category'>
>>> phi(x[Integer(1)])
B[3] + 3*B[4]
>>> phi(x[Integer(2)])
2*B[3] + 5*B[4]

>>> m = matrix([[Integer(1),Integer(2)],[Integer(3),Integer(5)]])
>>> phi = X.module_morphism(matrix=m, codomain=Y, side='right',
...                         category=Modules(ZZ).WithBasis())
>>> phi.parent()
Set of Morphisms from X to Y
in Category of modules with basis over Integer Ring
>>> phi(x[Integer(1)])
B[3] + 2*B[4]
>>> phi(x[Integer(2)])
3*B[3] + 5*B[4]
```

Todo

Possibly implement rank, addition, multiplication, matrix, etc, from the stored matrix.

class sage.modules.with_basis.morphism.**PointwiseInverseFunction**(f)

Bases: SageObject

A class for pointwise inverse functions.

The pointwise inverse function of a function f is the function sending every x to $1/f(x)$.

EXAMPLES:

```
sage: from sage.modules.with_basis.morphism import PointwiseInverseFunction
sage: f = PointwiseInverseFunction(factorial)
sage: f(0), f(1), f(2), f(3)
(1, 1, 1/2, 1/6)
```

```
>>> from sage.all import *
>>> from sage.modules.with_basis.morphism import PointwiseInverseFunction
>>> f = PointwiseInverseFunction(factorial)
>>> f(Integer(0)), f(Integer(1)), f(Integer(2)), f(Integer(3))
(1, 1, 1/2, 1/6)
```

pointwise_inverse()

class sage.modules.with_basis.morphism.**TriangularModuleMorphism**(*triangular='upper'*, *unitriangular=False*, *key=None*, *inverse=None*, *inverse_on_support=<built-in function identity>*, *invertible=None*)

Bases: *ModuleMorphism*

An abstract class for triangular module morphisms.

Let X and Y be modules over the same base ring, with distinguished bases F indexed by I and G indexed by J, respectively.

A module morphism ϕ from X to Y is *triangular* if its representing matrix in the distinguished bases of X and Y is upper triangular (echelon form).

More precisely, ϕ is *upper triangular* w.r.t. a total order $<$ on J if, for any $j \in J$, there exists at most one index $i \in I$ such that the leading support of $\phi(F_i)$ is j (see `leading_support()`). We denote by $r(j)$ this index, setting $r(j)$ to `None` if it does not exist.

Lower triangular morphisms are defined similarly, taking the trailing support instead (see `trailing_support()`).

A triangular morphism is *unitriangular* if all its pivots (i.e. coefficient of j in each $\phi(F[r(j)])$) are 1.

INPUT:

- `domain` – a module with basis X
- `codomain` – a module with basis Y (default: X)
- `category` – a category, as for *ModuleMorphism*
- `triangular` – `'upper'` or `'lower'` (default: `'upper'`)
- `unitriangular` – boolean (default: `False`) As a shorthand, one may use `unitriangular='lower'` for `triangular='lower', unitriangular=True`.
- `key` – a comparison key on J (default: the usual comparison of elements of J)
- `inverse_on_support` – a function $J \to I \cup \{None\}$ implementing r (default: the identity function). If set to "compute", the values of $r(j)$ are precomputed by running through the index set I of the basis of the domain. This of course requires the domain to be finite dimensional.
- `invertible` – boolean or `None` (default: `None`); can be set to specify that ϕ is known to be (or not to be) invertible. If the domain and codomain share the same indexing set, this is by default automatically set to `True` if `inverse_on_support` is the identity, or in the finite dimensional case.

See also

- `ModulesWithBasis.ParentMethods.module_morphism()` for usage information and examples;
- *`sage.modules.with_basis.morphism`* for a technical overview of the classes for module morphisms;
- *`ModuleMorphismFromFunction`* and *`TriangularModuleMorphism`*.

OUTPUT: a morphism from X to Y

Warning

This class is meant to be used as a complement for a concrete morphism class. In particular, the `__init__()` method focuses on setting up the data structure describing the triangularity of the morphism. It purposely does *not* call `ModuleMorphism.__init__()` which should be called (directly or indirectly) beforehand.

EXAMPLES:

We construct and invert an upper unitriangular module morphism between two free $\mathbf{Q}$-modules:

```
sage: I = range(1,200)
sage: X = CombinatorialFreeModule(QQ, I); X.rename('X'); x = X.basis()
sage: Y = CombinatorialFreeModule(QQ, I); Y.rename('Y'); y = Y.basis()
sage: ut = Y.sum_of_monomials * divisors   # This * is map composition.
sage: phi = X.module_morphism(ut, unitriangular='upper', codomain=Y)
sage: phi(x[2])
B[1] + B[2]
sage: phi(x[6])
B[1] + B[2] + B[3] + B[6]
sage: phi(x[30])
B[1] + B[2] + B[3] + B[5] + B[6] + B[10] + B[15] + B[30]
sage: phi.preimage(y[2])
-B[1] + B[2]
sage: phi.preimage(y[6])
B[1] - B[2] - B[3] + B[6]
sage: phi.preimage(y[30])
-B[1] + B[2] + B[3] + B[5] - B[6] - B[10] - B[15] + B[30]
sage: (phi^-1)(y[30])
-B[1] + B[2] + B[3] + B[5] - B[6] - B[10] - B[15] + B[30]
```

```
>>> from sage.all import *
>>> I = range(Integer(1),Integer(200))
>>> X = CombinatorialFreeModule(QQ, I); X.rename('X'); x = X.basis()
>>> Y = CombinatorialFreeModule(QQ, I); Y.rename('Y'); y = Y.basis()
>>> ut = Y.sum_of_monomials * divisors   # This * is map composition.
>>> phi = X.module_morphism(ut, unitriangular='upper', codomain=Y)
>>> phi(x[Integer(2)])
B[1] + B[2]
>>> phi(x[Integer(6)])
B[1] + B[2] + B[3] + B[6]
>>> phi(x[Integer(30)])
```

(continues on next page)

(continued from previous page)

```
B[1] + B[2] + B[3] + B[5] + B[6] + B[10] + B[15] + B[30]
>>> phi.preimage(y[Integer(2)])
-B[1] + B[2]
>>> phi.preimage(y[Integer(6)])
B[1] - B[2] - B[3] + B[6]
>>> phi.preimage(y[Integer(30)])
-B[1] + B[2] + B[3] + B[5] - B[6] - B[10] - B[15] + B[30]
>>> (phi**-Integer(1))(y[Integer(30)])
-B[1] + B[2] + B[3] + B[5] - B[6] - B[10] - B[15] + B[30]
```

A lower triangular (but not unitriangular) morphism:

```
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename('X'); x = X.basis()
sage: def lt(i): return sum(j*x[j] for j in range(i, 4))
sage: phi = X.module_morphism(lt, triangular='lower', codomain=X)
sage: phi(x[2])
2*B[2] + 3*B[3]
sage: phi.preimage(x[2])
1/2*B[2] - 1/2*B[3]
sage: phi(phi.preimage(x[2]))
B[2]
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, [Integer(1), Integer(2), Integer(3)]); X.
↪rename('X'); x = X.basis()
>>> def lt(i): return sum(j*x[j] for j in range(i, Integer(4)))
>>> phi = X.module_morphism(lt, triangular='lower', codomain=X)
>>> phi(x[Integer(2)])
2*B[2] + 3*B[3]
>>> phi.preimage(x[Integer(2)])
1/2*B[2] - 1/2*B[3]
>>> phi(phi.preimage(x[Integer(2)]))
B[2]
```

Using the `key` keyword, we can use triangularity even if the map becomes triangular only after a permutation of the basis:

```
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename('X'); x = X.basis()
sage: def ut(i): return (x[1] + x[2] if i == 1 else x[2] + (x[3] if i == 3 else␣
↪0))
sage: perm = [0, 2, 1, 3]
sage: phi = X.module_morphism(ut, triangular='upper', codomain=X,
....:                         key=lambda a: perm[a])
sage: [phi(x[i]) for i in range(1, 4)]
[B[1] + B[2], B[2], B[2] + B[3]]
sage: [phi.preimage(x[i]) for i in range(1, 4)]
[B[1] - B[2], B[2], -B[2] + B[3]]
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, [Integer(1), Integer(2), Integer(3)]); X.
↪rename('X'); x = X.basis()
>>> def ut(i): return (x[Integer(1)] + x[Integer(2)] if i == Integer(1) else␣
```

(continues on next page)

(continued from previous page)

```
↪x[Integer(2)] + (x[Integer(3)] if i == Integer(3) else Integer(0)))
>>> perm = [Integer(0), Integer(2), Integer(1), Integer(3)]
>>> phi = X.module_morphism(ut, triangular='upper', codomain=X,
...                         key=lambda a: perm[a])
>>> [phi(x[i]) for i in range(Integer(1), Integer(4))]
[B[1] + B[2], B[2], B[2] + B[3]]
>>> [phi.preimage(x[i]) for i in range(Integer(1), Integer(4))]
[B[1] - B[2], B[2], -B[2] + B[3]]
```

The same works in the lower-triangular case:

```
sage: def lt(i): return (x[1] + x[2] + x[3] if i == 2 else x[i])
sage: phi = X.module_morphism(lt, triangular='lower', codomain=X,
....:                         key=lambda a: perm[a])
sage: [phi(x[i]) for i in range(1, 4)]
[B[1], B[1] + B[2] + B[3], B[3]]
sage: [phi.preimage(x[i]) for i in range(1, 4)]
[B[1], -B[1] + B[2] - B[3], B[3]]
```

```
>>> from sage.all import *
>>> def lt(i): return (x[Integer(1)] + x[Integer(2)] + x[Integer(3)] if i ==␣
↪Integer(2) else x[i])
>>> phi = X.module_morphism(lt, triangular='lower', codomain=X,
...                         key=lambda a: perm[a])
>>> [phi(x[i]) for i in range(Integer(1), Integer(4))]
[B[1], B[1] + B[2] + B[3], B[3]]
>>> [phi.preimage(x[i]) for i in range(Integer(1), Integer(4))]
[B[1], -B[1] + B[2] - B[3], B[3]]
```

An injective but not surjective morphism cannot be inverted, but the `inverse_on_support` keyword allows Sage to find a partial inverse:

```
sage: X = CombinatorialFreeModule(QQ, [1,2,3]); x = X.basis()
sage: Y = CombinatorialFreeModule(QQ, [1,2,3,4,5]); y = Y.basis()
sage: ult = lambda i: sum(  y[j] for j in range(i+1,6)  )
sage: phi = X.module_morphism(ult, unitriangular='lower', codomain=Y,
....:         inverse_on_support=lambda i: i-1 if i in [2,3,4] else None)
sage: phi(x[2])
B[3] + B[4] + B[5]
sage: phi.preimage(y[3])
B[2] - B[3]
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3)]); x = X.
↪basis()
>>> Y = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5)]); y = Y.basis()
>>> ult = lambda i: sum(  y[j] for j in range(i+Integer(1),Integer(6))  )
>>> phi = X.module_morphism(ult, unitriangular='lower', codomain=Y,
...         inverse_on_support=lambda i: i-Integer(1) if i in [Integer(2),
↪Integer(3),Integer(4)] else None)
>>> phi(x[Integer(2)])
```

(continues on next page)

(continued from previous page)

```
B[3] + B[4] + B[5]
>>> phi.preimage(y[Integer(3)])
B[2] - B[3]
```

The `inverse_on_support` keyword can also be used if the bases of the domain and the codomain are identical but one of them has to be permuted in order to render the morphism triangular. For example:

```
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename('X'); x = X.basis()
sage: def ut(i):
....:     return (x[3] if i == 1 else x[1] if i == 2
....:             else x[1] + x[2])
sage: def perm(i):
....:     return (2 if i == 1 else 3 if i == 2 else 1)
sage: phi = X.module_morphism(ut, triangular='upper', codomain=X,
....:                         inverse_on_support=perm)
sage: [phi(x[i]) for i in range(1, 4)]
[B[3], B[1], B[1] + B[2]]
sage: [phi.preimage(x[i]) for i in range(1, 4)]
[B[2], -B[2] + B[3], B[1]]
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, [Integer(1), Integer(2), Integer(3)]); X.
↪rename('X'); x = X.basis()
>>> def ut(i):
...     return (x[Integer(3)] if i == Integer(1) else x[Integer(1)] if i ==␣
↪Integer(2)
...             else x[Integer(1)] + x[Integer(2)])
>>> def perm(i):
...     return (Integer(2) if i == Integer(1) else Integer(3) if i == Integer(2)␣
↪else Integer(1))
>>> phi = X.module_morphism(ut, triangular='upper', codomain=X,
...                         inverse_on_support=perm)
>>> [phi(x[i]) for i in range(Integer(1), Integer(4))]
[B[3], B[1], B[1] + B[2]]
>>> [phi.preimage(x[i]) for i in range(Integer(1), Integer(4))]
[B[2], -B[2] + B[3], B[1]]
```

The same works if the permutation induces lower triangularity:

```
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename('X'); x = X.basis()
sage: def lt(i):
....:     return (x[3] if i == 1 else x[2] if i == 2
....:             else x[1] + x[2])
sage: def perm(i):
....:     return 4 - i
sage: phi = X.module_morphism(lt, triangular='lower', codomain=X,
....:                         inverse_on_support=perm)
sage: [phi(x[i]) for i in range(1, 4)]
[B[3], B[2], B[1] + B[2]]
sage: [phi.preimage(x[i]) for i in range(1, 4)]
[-B[2] + B[3], B[2], B[1]]
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, [Integer(1), Integer(2), Integer(3)]); X.
↪rename('X'); x = X.basis()
>>> def lt(i):
...     return (x[Integer(3)] if i == Integer(1) else x[Integer(2)] if i ==␣
↪Integer(2)
...             else x[Integer(1)] + x[Integer(2)])
>>> def perm(i):
...     return Integer(4) - i
>>> phi = X.module_morphism(lt, triangular='lower', codomain=X,
...                         inverse_on_support=perm)
>>> [phi(x[i]) for i in range(Integer(1), Integer(4))]
[B[3], B[2], B[1] + B[2]]
>>> [phi.preimage(x[i]) for i in range(Integer(1), Integer(4))]
[-B[2] + B[3], B[2], B[1]]
```

In the finite dimensional case, one can ask Sage to recover `inverse_on_support` by a precomputation:

```
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); x = X.basis()
sage: Y = CombinatorialFreeModule(QQ, [1, 2, 3, 4]); y = Y.basis()
sage: ut = lambda i: sum(  y[j] for j in range(1,i+2) )
sage: phi = X.module_morphism(ut, triangular='upper', codomain=Y,
....:                         inverse_on_support='compute')
sage: tx = "{} {} {}"
sage: for j in Y.basis().keys():
....:     i = phi._inverse_on_support(j)
....:     print(tx.format(j, i, phi(x[i]) if i is not None else None))
1 None None
2 1 B[1] + B[2]
3 2 B[1] + B[2] + B[3]
4 3 B[1] + B[2] + B[3] + B[4]
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, [Integer(1), Integer(2), Integer(3)]); x = X.
↪basis()
>>> Y = CombinatorialFreeModule(QQ, [Integer(1), Integer(2), Integer(3),␣
↪Integer(4)]); y = Y.basis()
>>> ut = lambda i: sum(  y[j] for j in range(Integer(1),i+Integer(2)) )
>>> phi = X.module_morphism(ut, triangular='upper', codomain=Y,
...                         inverse_on_support='compute')
>>> tx = "{} {} {}"
>>> for j in Y.basis().keys():
...     i = phi._inverse_on_support(j)
...     print(tx.format(j, i, phi(x[i]) if i is not None else None))
1 None None
2 1 B[1] + B[2]
3 2 B[1] + B[2] + B[3]
4 3 B[1] + B[2] + B[3] + B[4]
```

The `inverse_on_basis` and `key` keywords can be combined:

```
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); X.rename('X')
sage: x = X.basis()
```

(continues on next page)

(continued from previous page)

```
sage: def ut(i):
....:     return (2*x[2] + 3*x[3] if i == 1
....:             else x[1] + x[2] + x[3] if i == 2
....:             else 4*x[2])
sage: def perm(i):
....:     return (2 if i == 1 else 3 if i == 2 else 1)
sage: perverse_key = lambda a: (a - 2) % 3
sage: phi = X.module_morphism(ut, triangular='upper', codomain=X,
....:                         inverse_on_support=perm, key=perverse_key)
sage: [phi(x[i]) for i in range(1, 4)]
[2*B[2] + 3*B[3], B[1] + B[2] + B[3], 4*B[2]]
sage: [phi.preimage(x[i]) for i in range(1, 4)]
[-1/3*B[1] + B[2] - 1/12*B[3], 1/4*B[3], 1/3*B[1] - 1/6*B[3]]
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, [Integer(1), Integer(2), Integer(3)]); X.
↪rename('X')
>>> x = X.basis()
>>> def ut(i):
...     return (Integer(2)*x[Integer(2)] + Integer(3)*x[Integer(3)] if i ==␣
↪Integer(1)
...             else x[Integer(1)] + x[Integer(2)] + x[Integer(3)] if i ==␣
↪Integer(2)
...             else Integer(4)*x[Integer(2)])
>>> def perm(i):
...     return (Integer(2) if i == Integer(1) else Integer(3) if i == Integer(2)␣
↪else Integer(1))
>>> perverse_key = lambda a: (a - Integer(2)) % Integer(3)
>>> phi = X.module_morphism(ut, triangular='upper', codomain=X,
...                         inverse_on_support=perm, key=perverse_key)
>>> [phi(x[i]) for i in range(Integer(1), Integer(4))]
[2*B[2] + 3*B[3], B[1] + B[2] + B[3], 4*B[2]]
>>> [phi.preimage(x[i]) for i in range(Integer(1), Integer(4))]
[-1/3*B[1] + B[2] - 1/12*B[3], 1/4*B[3], 1/3*B[1] - 1/6*B[3]]
```

cokernel_basis_indices()

Return the indices of the natural monomial basis of the cokernel of `self`.

INPUT:

- `self` – a triangular morphism over a field or a unitriangular morphism over a ring, with a finite dimensional codomain.

OUTPUT:

A list E of indices of the basis $(B_e)_e$ of the codomain of `self` so that $(B_e)_{e\in E}$ forms a basis of a supplementary of the image set of `self`.

Thinking of this triangular morphism as a row echelon matrix, this returns the complementary of the characteristic columns. Namely E is the set of indices which do not appear as leading support of some element of the image set of `self`.

EXAMPLES:

```
sage: X = CombinatorialFreeModule(ZZ, [1,2,3]); x = X.basis()
sage: Y = CombinatorialFreeModule(ZZ, [1,2,3,4,5]); y = Y.basis()
sage: uut = lambda i: sum(  y[j] for j in range(i+1,6)  ) # uni-upper
sage: phi = X.module_morphism(uut, unitriangular='upper', codomain=Y,
....:       inverse_on_support=lambda i: i-1 if i in [2,3,4] else None)
sage: phi.cokernel_basis_indices()
[1, 5]

sage: phi = X.module_morphism(uut, triangular='upper', codomain=Y,
....:       inverse_on_support=lambda i: i-1 if i in [2,3,4] else None)
sage: phi.cokernel_basis_indices()
Traceback (most recent call last):
...
NotImplementedError: cokernel_basis_indices for a triangular but not␣
↪unitriangular morphism over a ring

sage: Y = CombinatorialFreeModule(ZZ, NN); y = Y.basis()
sage: phi = X.module_morphism(uut, unitriangular='upper', codomain=Y,
....:       inverse_on_support=lambda i: i-1 if i in [2,3,4] else None)
sage: phi.cokernel_basis_indices()
Traceback (most recent call last):
...
NotImplementedError: cokernel_basis_indices implemented only for morphisms␣
↪with a finite dimensional codomain
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(ZZ, [Integer(1),Integer(2),Integer(3)]); x =␣
↪X.basis()
>>> Y = CombinatorialFreeModule(ZZ, [Integer(1),Integer(2),Integer(3),
↪Integer(4),Integer(5)]); y = Y.basis()
>>> uut = lambda i: sum(  y[j] for j in range(i+Integer(1),Integer(6))  ) #␣
↪uni-upper
>>> phi = X.module_morphism(uut, unitriangular='upper', codomain=Y,
...       inverse_on_support=lambda i: i-Integer(1) if i in [Integer(2),
↪Integer(3),Integer(4)] else None)
>>> phi.cokernel_basis_indices()
[1, 5]

>>> phi = X.module_morphism(uut, triangular='upper', codomain=Y,
...       inverse_on_support=lambda i: i-Integer(1) if i in [Integer(2),
↪Integer(3),Integer(4)] else None)
>>> phi.cokernel_basis_indices()
Traceback (most recent call last):
...
NotImplementedError: cokernel_basis_indices for a triangular but not␣
↪unitriangular morphism over a ring

>>> Y = CombinatorialFreeModule(ZZ, NN); y = Y.basis()
>>> phi = X.module_morphism(uut, unitriangular='upper', codomain=Y,
...       inverse_on_support=lambda i: i-Integer(1) if i in [Integer(2),
↪Integer(3),Integer(4)] else None)
>>> phi.cokernel_basis_indices()
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
NotImplementedError: cokernel_basis_indices implemented only for morphisms␣
↪with a finite dimensional codomain
```

cokernel_projection(*category=None*)

Return a projection on the co-kernel of `self`.

INPUT:

- `category` – the category of the result

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, [1,2,3]); x = X.basis()
sage: Y = CombinatorialFreeModule(QQ, [1,2,3,4,5]); y = Y.basis()
sage: lt = lambda i: sum(  y[j] for j in range(i+1,6)  ) # lower
sage: phi = X.module_morphism(lt, triangular='lower', codomain=Y,
....:       inverse_on_support=lambda i: i-1 if i in [2,3,4] else None)
sage: phipro = phi.cokernel_projection()
sage: phipro(y[1] + y[2])
B[1]
sage: all(phipro(phi(x)).is_zero() for x in X.basis())
True
sage: phipro(y[1])
B[1]
sage: phipro(y[4])
-B[5]
sage: phipro(y[5])
B[5]
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3)]); x =␣
↪X.basis()
>>> Y = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3),
↪Integer(4),Integer(5)]); y = Y.basis()
>>> lt = lambda i: sum(  y[j] for j in range(i+Integer(1),Integer(6))  ) #␣
↪lower
>>> phi = X.module_morphism(lt, triangular='lower', codomain=Y,
...       inverse_on_support=lambda i: i-Integer(1) if i in [Integer(2),
↪Integer(3),Integer(4)] else None)
>>> phipro = phi.cokernel_projection()
>>> phipro(y[Integer(1)] + y[Integer(2)])
B[1]
>>> all(phipro(phi(x)).is_zero() for x in X.basis())
True
>>> phipro(y[Integer(1)])
B[1]
>>> phipro(y[Integer(4)])
-B[5]
>>> phipro(y[Integer(5)])
B[5]
```

coreduced(*y*)

Return y reduced w.r.t. the image of `self`.

INPUT:

- `self` – a triangular morphism over a field, or a unitriangular morphism over a ring
- `y` – an element of the codomain of `self`

Suppose that `self` is a morphism from X to Y. Then, for any $y \in Y$, the call `self.coreduced(y)` returns a normal form for y in the quotient Y/I where I is the image of `self`.

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, [1,2,3]); x = X.basis()
sage: Y = CombinatorialFreeModule(QQ, [1,2,3,4,5]); y = Y.basis()
sage: ult = lambda i: sum(  y[j] for j in range(i+1,6)  )
sage: phi = X.module_morphism(ult, unitriangular='lower', codomain=Y,
....:         inverse_on_support=lambda i: i-1 if i in [2,3,4] else None)
sage: [phi(v) for v in X.basis()]
[B[2] + B[3] + B[4] + B[5],
        B[3] + B[4] + B[5],
               B[4] + B[5]]
sage: [phi.coreduced(y[1]-2*y[4])]
[B[1] + 2*B[5]]
sage: [phi.coreduced(v) for v in y]
[B[1], 0, 0, -B[5], B[5]]
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3)]); x =␣
↪X.basis()
>>> Y = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3),
↪Integer(4),Integer(5)]); y = Y.basis()
>>> ult = lambda i: sum(  y[j] for j in range(i+Integer(1),Integer(6))  )
>>> phi = X.module_morphism(ult, unitriangular='lower', codomain=Y,
...         inverse_on_support=lambda i: i-Integer(1) if i in [Integer(2),
↪Integer(3),Integer(4)] else None)
>>> [phi(v) for v in X.basis()]
[B[2] + B[3] + B[4] + B[5],
        B[3] + B[4] + B[5],
               B[4] + B[5]]
>>> [phi.coreduced(y[Integer(1)]-Integer(2)*y[Integer(4)])]
[B[1] + 2*B[5]]
>>> [phi.coreduced(v) for v in y]
[B[1], 0, 0, -B[5], B[5]]
```

Now with a non unitriangular morphism:

```
sage: lt = lambda i: sum( j*y[j] for j in range(i+1,6)  )
sage: phi = X.module_morphism(lt, triangular='lower', codomain=Y,
....:         inverse_on_support=lambda i: i-1 if i in [2,3,4] else None)
sage: [phi(v) for v in X.basis()]
[2*B[2] + 3*B[3] + 4*B[4] + 5*B[5],
          3*B[3] + 4*B[4] + 5*B[5],
                   4*B[4] + 5*B[5]]
sage: [phi.coreduced(y[1]-2*y[4])]
```

(continues on next page)

(continued from previous page)

```
[B[1] + 5/2*B[5]]
sage: [phi.coreduced(v) for v in y]
[B[1], 0, 0, -5/4*B[5], B[5]]
```

```
>>> from sage.all import *
>>> lt = lambda i: sum( j*y[j] for j in range(i+Integer(1),Integer(6))  )
>>> phi = X.module_morphism(lt, triangular='lower', codomain=Y,
...       inverse_on_support=lambda i: i-Integer(1) if i in [Integer(2),
↪Integer(3),Integer(4)] else None)
>>> [phi(v) for v in X.basis()]
[2*B[2] + 3*B[3] + 4*B[4] + 5*B[5],
          3*B[3] + 4*B[4] + 5*B[5],
                   4*B[4] + 5*B[5]]
>>> [phi.coreduced(y[Integer(1)]-Integer(2)*y[Integer(4)])]
[B[1] + 5/2*B[5]]
>>> [phi.coreduced(v) for v in y]
[B[1], 0, 0, -5/4*B[5], B[5]]
```

For general rings, this method is only implemented for unitriangular morphisms:

```
sage: X = CombinatorialFreeModule(ZZ, [1,2,3]); x = X.basis()
sage: Y = CombinatorialFreeModule(ZZ, [1,2,3,4,5]); y = Y.basis()
sage: phi = X.module_morphism(ult, unitriangular='lower', codomain=Y,
....:       inverse_on_support=lambda i: i-1 if i in [2,3,4] else None)
sage: [phi.coreduced(y[1]-2*y[4])]
[B[1] + 2*B[5]]
sage: [phi.coreduced(v) for v in y]
[B[1], 0, 0, -B[5], B[5]]

sage: phi = X.module_morphism(lt, triangular='lower', codomain=Y,
....:       inverse_on_support=lambda i: i-1 if i in [2,3,4] else None)
sage: [phi.coreduced(v) for v in y]
Traceback (most recent call last):
...
NotImplementedError: coreduce for a triangular but not unitriangular morphism␣
↪over a ring
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(ZZ, [Integer(1),Integer(2),Integer(3)]); x =␣
↪X.basis()
>>> Y = CombinatorialFreeModule(ZZ, [Integer(1),Integer(2),Integer(3),
↪Integer(4),Integer(5)]); y = Y.basis()
>>> phi = X.module_morphism(ult, unitriangular='lower', codomain=Y,
...       inverse_on_support=lambda i: i-Integer(1) if i in [Integer(2),
↪Integer(3),Integer(4)] else None)
>>> [phi.coreduced(y[Integer(1)]-Integer(2)*y[Integer(4)])]
[B[1] + 2*B[5]]
>>> [phi.coreduced(v) for v in y]
[B[1], 0, 0, -B[5], B[5]]

>>> phi = X.module_morphism(lt, triangular='lower', codomain=Y,
...       inverse_on_support=lambda i: i-Integer(1) if i in [Integer(2),
```

(continues on next page)

(continued from previous page)

```
↪Integer(3),Integer(4)] else None)
>>> [phi.coreduced(v) for v in y]
Traceback (most recent call last):
...
NotImplementedError: coreduce for a triangular but not unitriangular morphism␣
↪over a ring
```

Note

Before Issue #8678 this method used to be called co_reduced.

preimage(f)

Return the preimage of f under `self`.

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); x = X.basis()
sage: Y = CombinatorialFreeModule(QQ, [1, 2, 3]); y = Y.basis()
sage: ult = lambda i: sum(  y[j] for j in range(i,4)  ) # uni-lower
sage: phi = X.module_morphism(ult, triangular='lower', codomain=Y)
sage: phi.preimage(y[1] + y[2])
B[1] - B[3]
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, [Integer(1), Integer(2), Integer(3)]); x␣
↪= X.basis()
>>> Y = CombinatorialFreeModule(QQ, [Integer(1), Integer(2), Integer(3)]); y␣
↪= Y.basis()
>>> ult = lambda i: sum(  y[j] for j in range(i,Integer(4))  ) # uni-lower
>>> phi = X.module_morphism(ult, triangular='lower', codomain=Y)
>>> phi.preimage(y[Integer(1)] + y[Integer(2)])
B[1] - B[3]
```

The morphism need not be surjective. In the following example, the codomain is of larger dimension than the domain:

```
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); x = X.basis()
sage: Y = CombinatorialFreeModule(QQ, [1, 2, 3, 4]); y = Y.basis()
sage: lt = lambda i: sum(  y[j] for j in range(i,5)  )
sage: phi = X.module_morphism(lt, triangular='lower', codomain=Y)
sage: phi.preimage(y[1] + y[2])
B[1] - B[3]
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, [Integer(1), Integer(2), Integer(3)]); x␣
↪= X.basis()
>>> Y = CombinatorialFreeModule(QQ, [Integer(1), Integer(2), Integer(3),␣
↪Integer(4)]); y = Y.basis()
>>> lt = lambda i: sum(  y[j] for j in range(i,Integer(5))  )
>>> phi = X.module_morphism(lt, triangular='lower', codomain=Y)
```

(continues on next page)

(continued from previous page)

```
>>> phi.preimage(y[Integer(1)] + y[Integer(2)])
B[1] - B[3]
```

Here are examples using `inverse_on_support` to handle a morphism that shifts the leading indices by 1:

```
sage: X = CombinatorialFreeModule(QQ, [1, 2, 3]); x = X.basis()
sage: Y = CombinatorialFreeModule(QQ, [1, 2, 3, 4, 5]); y = Y.basis()
sage: lt = lambda i: sum(  y[j] for j in range(i+1,6)  ) # lower
sage: phi = X.module_morphism(lt, triangular='lower', codomain=Y,
....:         inverse_on_support=lambda i: i-1 if i in [2,3,4] else None)
sage: phi(x[1])
B[2] + B[3] + B[4] + B[5]
sage: phi(x[3])
B[4] + B[5]
sage: phi.preimage(y[2] + y[3])
B[1] - B[3]
sage: phi(phi.preimage(y[2] + y[3])) == y[2] + y[3]
True
sage: el = x[1] + 3*x[2] + 2*x[3]
sage: phi.preimage(phi(el)) == el
True

sage: phi.preimage(y[1])
Traceback (most recent call last):
...
ValueError: B[1] is not in the image
sage: phi.preimage(y[4])
Traceback (most recent call last):
...
ValueError: B[4] is not in the image
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, [Integer(1), Integer(2), Integer(3)]); x␣
↪= X.basis()
>>> Y = CombinatorialFreeModule(QQ, [Integer(1), Integer(2), Integer(3),␣
↪Integer(4), Integer(5)]); y = Y.basis()
>>> lt = lambda i: sum(  y[j] for j in range(i+Integer(1),Integer(6))  ) #␣
↪lower
>>> phi = X.module_morphism(lt, triangular='lower', codomain=Y,
...         inverse_on_support=lambda i: i-Integer(1) if i in [Integer(2),
↪Integer(3),Integer(4)] else None)
>>> phi(x[Integer(1)])
B[2] + B[3] + B[4] + B[5]
>>> phi(x[Integer(3)])
B[4] + B[5]
>>> phi.preimage(y[Integer(2)] + y[Integer(3)])
B[1] - B[3]
>>> phi(phi.preimage(y[Integer(2)] + y[Integer(3)])) == y[Integer(2)] +␣
↪y[Integer(3)]
True
>>> el = x[Integer(1)] + Integer(3)*x[Integer(2)] + Integer(2)*x[Integer(3)]
>>> phi.preimage(phi(el)) == el
```

(continues on next page)

(continued from previous page)

```
True

>>> phi.preimage(y[Integer(1)])
Traceback (most recent call last):
...
ValueError: B[1] is not in the image
>>> phi.preimage(y[Integer(4)])
Traceback (most recent call last):
...
ValueError: B[4] is not in the image
```

Over a base ring like **Z**, the morphism need not be surjective even when the dimensions match:

```
sage: X = CombinatorialFreeModule(ZZ, [1, 2, 3]); x = X.basis()
sage: Y = CombinatorialFreeModule(ZZ, [1, 2, 3]); y = Y.basis()
sage: lt = lambda i: sum( 2* y[j] for j in range(i,4)  ) # lower
sage: phi = X.module_morphism(lt, triangular='lower', codomain=Y)
sage: phi.preimage(2*y[1] + 2*y[2])
B[1] - B[3]
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(ZZ, [Integer(1), Integer(2), Integer(3)]); x␣
↪= X.basis()
>>> Y = CombinatorialFreeModule(ZZ, [Integer(1), Integer(2), Integer(3)]); y␣
↪= Y.basis()
>>> lt = lambda i: sum( Integer(2)* y[j] for j in range(i,Integer(4))  ) #␣
↪lower
>>> phi = X.module_morphism(lt, triangular='lower', codomain=Y)
>>> phi.preimage(Integer(2)*y[Integer(1)] + Integer(2)*y[Integer(2)])
B[1] - B[3]
```

The error message in case of failure could be more specific though:

```
sage: phi.preimage(y[1] + y[2])
Traceback (most recent call last):
  ...
TypeError: no conversion of this rational to integer
```

```
>>> from sage.all import *
>>> phi.preimage(y[Integer(1)] + y[Integer(2)])
Traceback (most recent call last):
  ...
TypeError: no conversion of this rational to integer
```

section()

Return the section (partial inverse) of `self`.

This returns a partial triangular morphism which is a section of `self`. The section morphism raises a `ValueError` if asked to apply on an element which is not in the image of `self`.

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, [1,2,3]); x = X.basis()
sage: X.rename('X')
sage: Y = CombinatorialFreeModule(QQ, [1,2,3,4,5]); y = Y.basis()
sage: ult = lambda i: sum(  y[j] for j in range(i+1,6)  ) # uni-lower
sage: phi = X.module_morphism(ult, triangular='lower', codomain=Y,
....:       inverse_on_support=lambda i: i-1 if i in [2,3,4] else None)
sage: ~phi
Traceback (most recent call last):
...
ValueError: Morphism not known to be invertible;
see the invertible option of module_morphism
sage: phiinv = phi.section()
sage: list(map(phiinv*phi, X.basis().list())) == X.basis().list()
True
sage: phiinv(Y.basis()[1])
Traceback (most recent call last):
...
ValueError: B[1] is not in the image
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3)]); x =␣
↪X.basis()
>>> X.rename('X')
>>> Y = CombinatorialFreeModule(QQ, [Integer(1),Integer(2),Integer(3),
↪Integer(4),Integer(5)]); y = Y.basis()
>>> ult = lambda i: sum(  y[j] for j in range(i+Integer(1),Integer(6))  ) #␣
↪uni-lower
>>> phi = X.module_morphism(ult, triangular='lower', codomain=Y,
...       inverse_on_support=lambda i: i-Integer(1) if i in [Integer(2),
↪Integer(3),Integer(4)] else None)
>>> ~phi
Traceback (most recent call last):
...
ValueError: Morphism not known to be invertible;
see the invertible option of module_morphism
>>> phiinv = phi.section()
>>> list(map(phiinv*phi, X.basis().list())) == X.basis().list()
True
>>> phiinv(Y.basis()[Integer(1)])
Traceback (most recent call last):
...
ValueError: B[1] is not in the image
```

class sage.modules.with_basis.morphism.**TriangularModuleMorphismByLinearity**(*domain*, *on_basis*, *codomain=None*, *category=None*, ***keywords*)

Bases: *ModuleMorphismByLinearity*, *TriangularModuleMorphism*

A concrete class for triangular module morphisms obtained by extending a function by linearity.

See also

- ModulesWithBasis.ParentMethods.module_morphism() for usage information and examples;
- *sage.modules.with_basis.morphism* for a technical overview of the classes for module morphisms;
- *ModuleMorphismByLinearity* and *TriangularModuleMorphism*.

class sage.modules.with_basis.morphism.**TriangularModuleMorphismFromFunction**(*domain*, *function*, *codomain=None*, *category=None*, ***keywords*)

Bases: *ModuleMorphismFromFunction*, *TriangularModuleMorphism*

A concrete class for triangular module morphisms implemented by a function.

See also

- ModulesWithBasis.ParentMethods.module_morphism() for usage information and examples;
- *sage.modules.with_basis.morphism* for a technical overview of the classes for module morphisms;
- *ModuleMorphismFromFunction* and *TriangularModuleMorphism*.

sage.modules.with_basis.morphism.**pointwise_inverse_function**(f)

Return the function $x \mapsto 1/f(x)$.

INPUT:

- f – a function

EXAMPLES:

```
sage: from sage.modules.with_basis.morphism import pointwise_inverse_function
sage: def f(x): return x
sage: g = pointwise_inverse_function(f)
sage: g(1), g(2), g(3)
(1, 1/2, 1/3)
```

```
>>> from sage.all import *
>>> from sage.modules.with_basis.morphism import pointwise_inverse_function
>>> def f(x): return x
>>> g = pointwise_inverse_function(f)
>>> g(Integer(1)), g(Integer(2)), g(Integer(3))
(1, 1/2, 1/3)
```

pointwise_inverse_function() is an involution:

```
sage: f is pointwise_inverse_function(g)
True
```

```
>>> from sage.all import *
>>> f is pointwise_inverse_function(g)
True
```

Todo

This has nothing to do here!!! Should there be a library for pointwise operations on functions somewhere in Sage?

3.6 Quotients of modules with basis

class sage.modules.with_basis.subquotient.**QuotientModuleWithBasis**(*submodule*, *category*, **args*, ***opts*)

Bases: CombinatorialFreeModule

A class for quotients of a module with basis by a submodule.

INPUT:

- submodule – a submodule of self
- category – a category (default: ModulesWithBasis(submodule.base_ring()))

submodule should be a free submodule admitting a basis in unitriangular echelon form. Typically submodule is a *SubmoduleWithBasis* as returned by Modules.WithBasis.ParentMethods.submodule().

The lift method should have a method .cokernel_basis_indices that computes the indexing set of a subset B of the basis of self that spans some supplementary of submodule in self (typically the non characteristic columns of the aforementioned echelon form). submodule should further implement a submodule.reduce(x) method that returns the unique element in the span of B which is equivalent to x modulo submodule.

This is meant to be constructed via Modules.WithBasis.FiniteDimensional.ParentMethods.quotient_module()

This differs from sage.rings.quotient_ring.QuotientRing in the following ways:

- submodule needs not be an ideal. If it is, the transportation of the ring structure is taken care of by the Subquotients categories.
- Thanks to .cokernel_basis_indices, we know the indices of a basis of the quotient, and elements are represented directly in the free module spanned by those indices rather than by wrapping elements of the ambient space.

There is room for sharing more code between those two implementations and generalizing them. See Issue #18204.

See also

- Modules.WithBasis.ParentMethods.submodule()
- Modules.WithBasis.FiniteDimensional.ParentMethods.quotient_module()
- *SubmoduleWithBasis*
- sage.rings.quotient_ring.QuotientRing

ambient()

Return the ambient space of `self`.

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, range(3), prefix='x'); x = X.basis()
sage: Y = X.quotient_module((x[0]-x[1], x[1]-x[2]))
sage: Y.ambient() is X
True
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, range(Integer(3)), prefix='x'); x = X.
↪basis()
>>> Y = X.quotient_module((x[Integer(0)]-x[Integer(1)], x[Integer(1)]-
↪x[Integer(2)]))
>>> Y.ambient() is X
True
```

lift(x)

Lift `x` to the ambient space of `self`.

INPUT:

- `x` – an element of `self`

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, range(3), prefix='x'); x = X.basis()
sage: Y = X.quotient_module((x[0]-x[1], x[1]-x[2]));         y = Y.basis()
sage: Y.lift(y[2])
x[2]
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, range(Integer(3)), prefix='x'); x = X.
↪basis()
>>> Y = X.quotient_module((x[Integer(0)]-x[Integer(1)], x[Integer(1)]-
↪x[Integer(2)]));         y = Y.basis()
>>> Y.lift(y[Integer(2)])
x[2]
```

retract(x)

Retract an element of the ambient space by projecting it back to `self`.

INPUT:

- `x` – an element of the ambient space of `self`

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, range(3), prefix='x'); x = X.basis()
sage: Y = X.quotient_module((x[0]-x[1], x[1]-x[2]));         y = Y.basis()
sage: Y.print_options(prefix='y')
sage: Y.retract(x[0])
y[2]
sage: Y.retract(x[1])
y[2]
```

(continues on next page)

(continued from previous page)

```
sage: Y.retract(x[2])
y[2]
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, range(Integer(3)), prefix='x'); x = X.
↪basis()
>>> Y = X.quotient_module((x[Integer(0)]-x[Integer(1)], x[Integer(1)]-
↪x[Integer(2)])); y = Y.basis()
>>> Y.print_options(prefix='y')
>>> Y.retract(x[Integer(0)])
y[2]
>>> Y.retract(x[Integer(1)])
y[2]
>>> Y.retract(x[Integer(2)])
y[2]
```

class sage.modules.with_basis.subquotient.**SubmoduleWithBasis**(*basis*, *support_order*, *ambient*, *unitriangular*, *category*, **args*, ***opts*)

Bases: CombinatorialFreeModule

A base class for submodules of a ModuleWithBasis spanned by a (possibly infinite) basis in echelon form.

INPUT:

- `basis` – a family of elements in echelon form in some `module with basis` V, or data that can be converted into such a family
- `support_order` – an ordering of the support of `basis` expressed in `ambient` given as a list
- `unitriangular` – if the lift morphism is unitriangular
- `ambient` – the ambient space V
- `category` – a category

Further arguments are passed down to CombinatorialFreeModule.

This is meant to be constructed via `Modules.WithBasis.ParentMethods.submodule()`.

See also

- `Modules.WithBasis.ParentMethods.submodule()`
- *QuotientModuleWithBasis*

ambient()

Return the ambient space of `self`.

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, range(3)); x = X.basis()
sage: Y = X.submodule((x[0]-x[1], x[1]-x[2]))
sage: Y.ambient() is X
True
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, range(Integer(3))); x = X.basis()
>>> Y = X.submodule((x[Integer(0)]-x[Integer(1)], x[Integer(1)]-
↪x[Integer(2)]))
>>> Y.ambient() is X
True
```

intersection(*other*)

Return the intersection of `self` and `other`.

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, range(4)); x = X.basis()
sage: F = X.submodule([x[0]-x[1], x[1]-x[2], x[2]-x[3]])
sage: G = X.submodule([x[0]-x[2]])
sage: H = X.submodule([x[0]-x[1], x[2]])
sage: FG = F & G; FG
Free module generated by {0} over Rational Field
sage: [FG.lift(b) for b in FG.basis()]
[B[0] - B[2]]
sage: FH = F & H; FH
Free module generated by {0} over Rational Field
sage: [FH.lift(b) for b in FH.basis()]
[B[0] - B[1]]
sage: GH = G & H; GH
Free module generated by {} over Rational Field
sage: [GH.lift(b) for b in GH.basis()]
[]

sage: F.intersection(X) is F
True
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, range(Integer(4))); x = X.basis()
>>> F = X.submodule([x[Integer(0)]-x[Integer(1)], x[Integer(1)]-x[Integer(2)],
↪ x[Integer(2)]-x[Integer(3)]])
>>> G = X.submodule([x[Integer(0)]-x[Integer(2)]])
>>> H = X.submodule([x[Integer(0)]-x[Integer(1)], x[Integer(2)]])
>>> FG = F & G; FG
Free module generated by {0} over Rational Field
>>> [FG.lift(b) for b in FG.basis()]
[B[0] - B[2]]
>>> FH = F & H; FH
Free module generated by {0} over Rational Field
>>> [FH.lift(b) for b in FH.basis()]
[B[0] - B[1]]
>>> GH = G & H; GH
Free module generated by {} over Rational Field
>>> [GH.lift(b) for b in GH.basis()]
[]

>>> F.intersection(X) is F
True
```

is_equal_subspace(*other*)

Return whether `self` is an equal submodule to `other`.

Note

This is the mathematical notion of equality (as sets that are isomorphic as vector spaces), which is weaker than the == which takes into account things like the support order.

INPUT:

- `other` – another submodule of the same ambient module or the ambient module itself

EXAMPLES:

```
sage: R.<z> = LaurentPolynomialRing(QQ)
sage: X = CombinatorialFreeModule(R, range(4)); x = X.basis()
sage: F = X.submodule([x[0]-x[1], z*x[1]-z*x[2], z^2*x[2]-z^2*x[3]])
sage: G = X.submodule([x[0]-x[1], x[1]-x[2], x[2]-x[3]])
sage: H = X.submodule([x[0]-x[1], x[1]-x[2], x[2]-x[3]], support_order=(3,2,1,
↪0))
sage: F.is_equal_subspace(F)
True
sage: F == G
False
sage: F.is_equal_subspace(G)
True
sage: F.is_equal_subspace(H)
True
sage: G == H  # different support orders
False
sage: G.is_equal_subspace(H)
True
```

```
>>> from sage.all import *
>>> R = LaurentPolynomialRing(QQ, names=('z',)); (z,) = R._first_ngens(1)
>>> X = CombinatorialFreeModule(R, range(Integer(4))); x = X.basis()
>>> F = X.submodule([x[Integer(0)]-x[Integer(1)], z*x[Integer(1)]-
↪z*x[Integer(2)], z**Integer(2)*x[Integer(2)]-z**Integer(2)*x[Integer(3)]])
>>> G = X.submodule([x[Integer(0)]-x[Integer(1)], x[Integer(1)]-x[Integer(2)],
↪ x[Integer(2)]-x[Integer(3)]])
>>> H = X.submodule([x[Integer(0)]-x[Integer(1)], x[Integer(1)]-x[Integer(2)],
↪ x[Integer(2)]-x[Integer(3)]], support_order=(Integer(3),Integer(2),
↪Integer(1),Integer(0)))
>>> F.is_equal_subspace(F)
True
>>> F == G
False
>>> F.is_equal_subspace(G)
True
>>> F.is_equal_subspace(H)
True
>>> G == H  # different support orders
False
```

(continues on next page)

(continued from previous page)

```
>>> G.is_equal_subspace(H)
True
```

```
sage: X = CombinatorialFreeModule(QQ, ZZ); x = X.basis()
sage: F = X.submodule([x[0]-x[1], x[1]-x[3]])
sage: G = X.submodule([x[0]-x[1], x[2]])
sage: H = X.submodule([x[0]+x[1], x[1]+3*x[2]])
sage: Hp = X.submodule([x[0]+x[1], x[1]+3*x[2]], prefix='Hp')
sage: F.is_equal_subspace(X)
False
sage: F.is_equal_subspace(G)
False
sage: G.is_equal_subspace(H)
False
sage: H == Hp
False
sage: H.is_equal_subspace(Hp)
True
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, ZZ); x = X.basis()
>>> F = X.submodule([x[Integer(0)]-x[Integer(1)], x[Integer(1)]-
↪x[Integer(3)]])
>>> G = X.submodule([x[Integer(0)]-x[Integer(1)], x[Integer(2)]])
>>> H = X.submodule([x[Integer(0)]+x[Integer(1)],␣
↪x[Integer(1)]+Integer(3)*x[Integer(2)]])
>>> Hp = X.submodule([x[Integer(0)]+x[Integer(1)],␣
↪x[Integer(1)]+Integer(3)*x[Integer(2)]], prefix='Hp')
>>> F.is_equal_subspace(X)
False
>>> F.is_equal_subspace(G)
False
>>> G.is_equal_subspace(H)
False
>>> H == Hp
False
>>> H.is_equal_subspace(Hp)
True
```

is_submodule(*other*)

Return whether `self` is a submodule of `other`.

INPUT:

- `other` – another submodule of the same ambient module or the ambient module itself

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, range(4)); x = X.basis()
sage: F = X.submodule([x[0]-x[1], x[1]-x[2], x[2]-x[3]])
sage: G = X.submodule([x[0]-x[2]])
sage: H = X.submodule([x[0]-x[1], x[2]])
sage: F.is_submodule(X)
```

(continues on next page)

(continued from previous page)

```
True
sage: G.is_submodule(F)
True
sage: H.is_submodule(F)
False
sage: H.is_submodule(G)
False
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, range(Integer(4))); x = X.basis()
>>> F = X.submodule([x[Integer(0)]-x[Integer(1)], x[Integer(1)]-x[Integer(2)],
↪ x[Integer(2)]-x[Integer(3)]])
>>> G = X.submodule([x[Integer(0)]-x[Integer(2)]])
>>> H = X.submodule([x[Integer(0)]-x[Integer(1)], x[Integer(2)]])
>>> F.is_submodule(X)
True
>>> G.is_submodule(F)
True
>>> H.is_submodule(F)
False
>>> H.is_submodule(G)
False
```

Infinite dimensional examples:

```
sage: X = CombinatorialFreeModule(QQ, ZZ); x = X.basis()
sage: F = X.submodule([x[0]-x[1], x[1]-x[2], x[2]-x[3]])
sage: G = X.submodule([x[0]-x[2]])
sage: H = X.submodule([x[0]-x[1]])
sage: F.is_submodule(X)
True
sage: G.is_submodule(F)
True
sage: H.is_submodule(F)
True
sage: H.is_submodule(G)
False
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, ZZ); x = X.basis()
>>> F = X.submodule([x[Integer(0)]-x[Integer(1)], x[Integer(1)]-x[Integer(2)],
↪ x[Integer(2)]-x[Integer(3)]])
>>> G = X.submodule([x[Integer(0)]-x[Integer(2)]])
>>> H = X.submodule([x[Integer(0)]-x[Integer(1)]])
>>> F.is_submodule(X)
True
>>> G.is_submodule(F)
True
>>> H.is_submodule(F)
True
>>> H.is_submodule(G)
False
```

Different ambient spaces:

```
sage: X = CombinatorialFreeModule(QQ, range(4)); x = X.basis()
sage: F = X.submodule([x[0]-x[1], x[1]-x[2], x[2]-x[3]])
sage: Y = CombinatorialFreeModule(QQ, range(6)); y = Y.basis()
sage: G = Y.submodule([y[0]-y[1], y[1]-y[2], y[2]-y[3]])
sage: F.is_submodule(G)
False
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, range(Integer(4))); x = X.basis()
>>> F = X.submodule([x[Integer(0)]-x[Integer(1)], x[Integer(1)]-x[Integer(2)],
↪ x[Integer(2)]-x[Integer(3)]])
>>> Y = CombinatorialFreeModule(QQ, range(Integer(6))); y = Y.basis()
>>> G = Y.submodule([y[Integer(0)]-y[Integer(1)], y[Integer(1)]-y[Integer(2)],
↪ y[Integer(2)]-y[Integer(3)]])
>>> F.is_submodule(G)
False
```

lift()

The lift (embedding) map from `self` to the ambient space.

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, range(3), prefix='x');              x =␣
↪X.basis()
sage: Y = X.submodule((x[0]-x[1], x[1]-x[2]), already_echelonized=True); y =␣
↪Y.basis()
sage: Y.lift
Generic morphism:
  From: Free module generated by {0, 1} over Rational Field
  To:   Free module generated by {0, 1, 2} over Rational Field
sage: [ Y.lift(u) for u in y ]
[x[0] - x[1], x[1] - x[2]]
sage: (y[0] + y[1]).lift()
x[0] - x[2]
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, range(Integer(3)), prefix='x');          ␣
↪  x = X.basis()
>>> Y = X.submodule((x[Integer(0)]-x[Integer(1)], x[Integer(1)]-
↪x[Integer(2)]), already_echelonized=True); y = Y.basis()
>>> Y.lift
Generic morphism:
  From: Free module generated by {0, 1} over Rational Field
  To:   Free module generated by {0, 1, 2} over Rational Field
>>> [ Y.lift(u) for u in y ]
[x[0] - x[1], x[1] - x[2]]
>>> (y[Integer(0)] + y[Integer(1)]).lift()
x[0] - x[2]
```

reduce()

The reduce map.

This map reduces elements of the ambient space modulo this submodule.

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, range(3), prefix='x'); x = X.basis()
sage: Y = X.submodule((x[0]-x[1], x[1]-x[2]), already_echelonized=True)
sage: Y.reduce
Generic endomorphism of Free module generated by {0, 1, 2} over Rational Field
sage: Y.reduce(x[1])
x[2]
sage: Y.reduce(2*x[0] + x[1])
3*x[2]
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, range(Integer(3)), prefix='x'); x = X.
↪basis()
>>> Y = X.submodule((x[Integer(0)]-x[Integer(1)], x[Integer(1)]-
↪x[Integer(2)]), already_echelonized=True)
>>> Y.reduce
Generic endomorphism of Free module generated by {0, 1, 2} over Rational Field
>>> Y.reduce(x[Integer(1)])
x[2]
>>> Y.reduce(Integer(2)*x[Integer(0)] + x[Integer(1)])
3*x[2]
```

retract()

The retract map from the ambient space.

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, range(3), prefix='x'); x = X.basis()
sage: Y = X.submodule((x[0]-x[1], x[1]-x[2]), already_echelonized=True)
sage: Y.print_options(prefix='y')
sage: Y.retract
Generic morphism:
  From: Free module generated by {0, 1, 2} over Rational Field
  To:   Free module generated by {0, 1} over Rational Field
sage: Y.retract(x[0] - x[2])
y[0] + y[1]
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, range(Integer(3)), prefix='x'); x = X.
↪basis()
>>> Y = X.submodule((x[Integer(0)]-x[Integer(1)], x[Integer(1)]-
↪x[Integer(2)]), already_echelonized=True)
>>> Y.print_options(prefix='y')
>>> Y.retract
Generic morphism:
  From: Free module generated by {0, 1, 2} over Rational Field
  To:   Free module generated by {0, 1} over Rational Field
>>> Y.retract(x[Integer(0)] - x[Integer(2)])
y[0] + y[1]
```

subspace(*gens*, **args*, ***opts*)

The submodule of the ambient space spanned by a finite set of generators `gens` (as a submodule).

INPUT:

- gens – list or family of elements of self

For additional optional arguments, see ModulesWithBasis.ParentMethods.submodule().

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, range(4), prefix='X'); x = X.basis()
sage: F = X.submodule([x[0]-x[1], x[1]-x[2], x[2]-x[3]], prefix='F'); f = F.
↪basis()
sage: U = F.submodule([f[0] + 2*f[1] - 5*f[2], f[1] + 2*f[2]]); U
Free module generated by {0, 1} over Rational Field
sage: [U.lift(u) for u in U.basis()]
[F[0] - 9*F[2], F[1] + 2*F[2]]
sage: V = F.subspace([f[0] + 2*f[1] - 5*f[2], f[1] + 2*f[2]]); V
Free module generated by {0, 1} over Rational Field
sage: [V.lift(u) for u in V.basis()]
[X[0] - 9*X[2] + 8*X[3], X[1] + 2*X[2] - 3*X[3]]
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, range(Integer(4)), prefix='X'); x = X.
↪basis()
>>> F = X.submodule([x[Integer(0)]-x[Integer(1)], x[Integer(1)]-x[Integer(2)],
↪ x[Integer(2)]-x[Integer(3)]], prefix='F'); f = F.basis()
>>> U = F.submodule([f[Integer(0)] + Integer(2)*f[Integer(1)] -␣
↪Integer(5)*f[Integer(2)], f[Integer(1)] + Integer(2)*f[Integer(2)]]); U
Free module generated by {0, 1} over Rational Field
>>> [U.lift(u) for u in U.basis()]
[F[0] - 9*F[2], F[1] + 2*F[2]]
>>> V = F.subspace([f[Integer(0)] + Integer(2)*f[Integer(1)] -␣
↪Integer(5)*f[Integer(2)], f[Integer(1)] + Integer(2)*f[Integer(2)]]); V
Free module generated by {0, 1} over Rational Field
>>> [V.lift(u) for u in V.basis()]
[X[0] - 9*X[2] + 8*X[3], X[1] + 2*X[2] - 3*X[3]]
```

subspace_sum(*other*)

Return the sum of self and other.

EXAMPLES:

```
sage: X = CombinatorialFreeModule(QQ, range(4)); x = X.basis()
sage: F = X.submodule([x[0]-x[1], x[1]-x[2], x[2]-x[3]])
sage: G = X.submodule([x[0]-x[2]])
sage: H = X.submodule([x[0]-x[1], x[2]])
sage: FG = F + G; FG
Free module generated by {0, 1, 2} over Rational Field
sage: [FG.lift(b) for b in FG.basis()]
[B[0] - B[3], B[1] - B[3], B[2] - B[3]]
sage: FH = F + H; FH
Free module generated by {0, 1, 2, 3} over Rational Field
sage: [FH.lift(b) for b in FH.basis()]
[B[0], B[1], B[2], B[3]]
sage: GH = G + H; GH
Free module generated by {0, 1, 2} over Rational Field
sage: [GH.lift(b) for b in GH.basis()]
[B[0], B[1], B[2]]
```

```
>>> from sage.all import *
>>> X = CombinatorialFreeModule(QQ, range(Integer(4))); x = X.basis()
>>> F = X.submodule([x[Integer(0)]-x[Integer(1)], x[Integer(1)]-x[Integer(2)],
↪ x[Integer(2)]-x[Integer(3)]])
>>> G = X.submodule([x[Integer(0)]-x[Integer(2)]])
>>> H = X.submodule([x[Integer(0)]-x[Integer(1)], x[Integer(2)]])
>>> FG = F + G; FG
Free module generated by {0, 1, 2} over Rational Field
>>> [FG.lift(b) for b in FG.basis()]
[B[0] - B[3], B[1] - B[3], B[2] - B[3]]
>>> FH = F + H; FH
Free module generated by {0, 1, 2, 3} over Rational Field
>>> [FH.lift(b) for b in FH.basis()]
[B[0], B[1], B[2], B[3]]
>>> GH = G + H; GH
Free module generated by {0, 1, 2} over Rational Field
>>> [GH.lift(b) for b in GH.basis()]
[B[0], B[1], B[2]]
```

3.7 Representations of a semigroup

AUTHORS:

- Travis Scrimshaw (2015-11-21): initial version
- Siddharth Singh (2020-03-21): signed representation
- Travis Scrimshaw (2024-02-17): tensor products

class sage.modules.with_basis.representation.**NaturalMatrixRepresentation**(*semigroup*, *base_ring*)

Bases: *Representation*

The natural representation of a matrix semigroup.

A matrix semigroup is defined by its representation on a (finite dimensional) vector space V, which is called the *natural representation*.

INPUT:

- `matrix_semigroup` – a matrix semigroup
- `base_ring` – (optional) the base ring; the default is the base ring of the semigroup

class sage.modules.with_basis.representation.**QuotientRepresentation**(**args*, ***kwds*)

Bases: *Representation_abstract*, *QuotientModuleWithBasis*

The quotient of a representation by another representation, which admits a natural structure of a representation.

Element

alias of *Element*

class sage.modules.with_basis.representation.**ReflectionRepresentation**(*W*, *base_ring*)

Bases: *Representation_abstract*, CombinatorialFreeModule

The reflection representation of a Coxeter group.

This is the canonical faithful representation of a Coxeter group.

EXAMPLES:

```
sage: W = CoxeterGroup(['B', 4])
sage: R = W.reflection_representation()
sage: all(g.matrix() == R.representation_matrix(g) for g in W)
True
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(['B', Integer(4)])
>>> R = W.reflection_representation()
>>> all(g.matrix() == R.representation_matrix(g) for g in W)
True
```

class sage.modules.with_basis.representation.**RegularRepresentation**(*semigroup*, *base_ring*, *side='left'*)

Bases: *Representation*

The regular representation of a semigroup.

The left regular representation of a semigroup S over a commutative ring R is the semigroup ring $R[S]$ equipped with the left S-action $xb_y = b_{xy}$, where $(b_z)_{z \in S}$ is the natural basis of $R[S]$ and $x, y \in S$.

INPUT:

- `semigroup` – a semigroup
- `base_ring` – the base ring for the representation
- `side` – (default: `'left'`) whether this is a `'left'` or `'right'` representation

REFERENCES:

- Wikipedia article Regular_representation

class sage.modules.with_basis.representation.**Representation**(*semigroup*, *module*, *on_basis*, *side='left'*, ***kwargs*)

Bases: *Representation_abstract*, CombinatorialFreeModule

Representation of a semigroup.

INPUT:

- `semigroup` – a semigroup
- `module` – a module with a basis
- `on_basis` – function which takes as input `g`, `m`, where `g` is an element of the semigroup and `m` is an element of the indexing set for the basis, and returns the result of `g` acting on `m`
- `side` – (default: `'left'`) whether this is a `'left'` or `'right'` representation

EXAMPLES:

We construct the sign representation of a symmetric group:

```
sage: G = SymmetricGroup(4)
sage: M = CombinatorialFreeModule(QQ, ['v'])
sage: on_basis = lambda g,m: M.term(m, g.sign())
sage: R = G.representation(M, on_basis)
sage: x = R.an_element(); x
2*B['v']
```

(continues on next page)

(continued from previous page)

```
sage: c,s = G.gens()
sage: c,s
((1,2,3,4), (1,2))
sage: c * x
-2*B['v']
sage: s * x
-2*B['v']
sage: c * s * x
2*B['v']
sage: (c * s) * x
2*B['v']
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(4))
>>> M = CombinatorialFreeModule(QQ, ['v'])
>>> on_basis = lambda g,m: M.term(m, g.sign())
>>> R = G.representation(M, on_basis)
>>> x = R.an_element(); x
2*B['v']
>>> c,s = G.gens()
>>> c,s
((1,2,3,4), (1,2))
>>> c * x
-2*B['v']
>>> s * x
-2*B['v']
>>> c * s * x
2*B['v']
>>> (c * s) * x
2*B['v']
```

This extends naturally to the corresponding group algebra:

```
sage: A = G.algebra(QQ)
sage: s,c = A.algebra_generators()
sage: c,s
((1,2,3,4), (1,2))
sage: c * x
-2*B['v']
sage: s * x
-2*B['v']
sage: c * s * x
2*B['v']
sage: (c * s) * x
2*B['v']
sage: (c + s) * x
-4*B['v']
```

```
>>> from sage.all import *
>>> A = G.algebra(QQ)
>>> s,c = A.algebra_generators()
>>> c,s
```

(continues on next page)

(continued from previous page)

```
((1,2,3,4), (1,2))
>>> c * x
-2*B['v']
>>> s * x
-2*B['v']
>>> c * s * x
2*B['v']
>>> (c * s) * x
2*B['v']
>>> (c + s) * x
-4*B['v']
```

REFERENCES:

- Wikipedia article Group_representation

product_by_coercion(*left*, *right*)

Return the product of `left` and `right` by passing to `self._module` and then building a new element of `self`.

EXAMPLES:

```
sage: G = groups.permutation.KleinFour()
sage: E = algebras.Exterior(QQ,'e',4)
sage: on_basis = lambda g,m: E.monomial(m) # the trivial representation
sage: R = G.representation(E, on_basis)
sage: r = R.an_element(); r
1 + 2*e0 + 3*e1 + e1*e2
sage: g = G.an_element();
sage: g * r == r  # indirect doctest
True
sage: r * r  # indirect doctest
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *:
 'Representation of The Klein 4 group of order 4, as a permutation
 group indexed by Subsets of {0,1,...,3} over Rational Field' and
 'Representation of The Klein 4 group of order 4, as a permutation
 group indexed by Subsets of {0,1,...,3} over Rational Field'

sage: from sage.categories.algebras import Algebras
sage: category = Algebras(QQ).FiniteDimensional().WithBasis()
sage: T = G.representation(E, on_basis, category=category)
sage: t = T.an_element(); t
1 + 2*e0 + 3*e1 + e1*e2
sage: g * t == t  # indirect doctest
True
sage: t * t  # indirect doctest
1 + 4*e0 + 4*e0*e1*e2 + 6*e1 + 2*e1*e2
```

```
>>> from sage.all import *
>>> G = groups.permutation.KleinFour()
>>> E = algebras.Exterior(QQ,'e',Integer(4))
```

(continues on next page)

(continued from previous page)

```
>>> on_basis = lambda g,m: E.monomial(m) # the trivial representation
>>> R = G.representation(E, on_basis)
>>> r = R.an_element(); r
1 + 2*e0 + 3*e1 + e1*e2
>>> g = G.an_element();
>>> g * r == r  # indirect doctest
True
>>> r * r  # indirect doctest
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *:
 'Representation of The Klein 4 group of order 4, as a permutation
 group indexed by Subsets of {0,1,...,3} over Rational Field' and
 'Representation of The Klein 4 group of order 4, as a permutation
 group indexed by Subsets of {0,1,...,3} over Rational Field'

>>> from sage.categories.algebras import Algebras
>>> category = Algebras(QQ).FiniteDimensional().WithBasis()
>>> T = G.representation(E, on_basis, category=category)
>>> t = T.an_element(); t
1 + 2*e0 + 3*e1 + e1*e2
>>> g * t == t  # indirect doctest
True
>>> t * t  # indirect doctest
1 + 4*e0 + 4*e0*e1*e2 + 6*e1 + 2*e1*e2
```

class sage.modules.with_basis.representation.**Representation_Exterior**(*rep*, *degree=None*, *category=None*, ***options*)

Bases: *Representation_abstract*, CombinatorialFreeModule

The exterior power representation (in a fixed degree).

class sage.modules.with_basis.representation.**Representation_ExteriorAlgebra**(*rep*, *degree=None*, *category=None*, ***options*)

Bases: *Representation_Exterior*

The exterior algebra representation.

one_basis()

Return the basis element indexing 1 in `self` if it exists.

EXAMPLES:

```
sage: S3 = SymmetricGroup(3)
sage: L = S3.regular_representation(side='left')
sage: E = L.exterior_power()
sage: E.one_basis()
0
sage: E0 = L.exterior_power(0)
```

(continues on next page)

(continued from previous page)

```
sage: E0.one_basis()
0
```

```
>>> from sage.all import *
>>> S3 = SymmetricGroup(Integer(3))
>>> L = S3.regular_representation(side='left')
>>> E = L.exterior_power()
>>> E.one_basis()
0
>>> E0 = L.exterior_power(Integer(0))
>>> E0.one_basis()
0
```

product_on_basis(*x*, *y*)

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

EXAMPLES:

```
sage: S3 = SymmetricGroup(3)
sage: L = S3.regular_representation(side='left')
sage: E = L.exterior_power()
sage: B = list(E.basis())
sage: B[:7]
[1, (), (1,3,2), (1,2,3), (2,3), (1,3), (1,2)]
sage: B[2] * B[4]  # indirect doctest
(1,3,2)*(2,3)
```

```
>>> from sage.all import *
>>> S3 = SymmetricGroup(Integer(3))
>>> L = S3.regular_representation(side='left')
>>> E = L.exterior_power()
>>> B = list(E.basis())
>>> B[:Integer(7)]
[1, (), (1,3,2), (1,2,3), (2,3), (1,3), (1,2)]
>>> B[Integer(2)] * B[Integer(4)]  # indirect doctest
(1,3,2)*(2,3)
```

class sage.modules.with_basis.representation.**Representation_Symmetric**(*rep*, *degree*, ***options*)

Bases: *Representation_abstract*, CombinatorialFreeModule

The symmetric power representation in a fixed degree.

class sage.modules.with_basis.representation.**Representation_Tensor**(*reps*, ***options*)

Bases: *Representation_abstract*, CombinatorialFreeModule_Tensor

Tensor product of representations.

class Element

Bases: *Element*

class sage.modules.with_basis.representation.**Representation_abstract**(*semigroup*, *side*, *algebra=None*)

Bases: object

Abstract base class for representations of semigroups.

INPUT:

- `semigroup` – a semigroup
- `side` – (default: `'left'`) whether this is a `'left'` or `'right'` representation
- `algebra` – (default: `semigroup.algebra(self.base_ring())`) the semigroup algebra

Note

This class should come before `CombinatorialFreeModule` in the MRO in order for tensor products to use the correct class.

class Element

Bases: *IndexedFreeModuleElement*

Tensor

alias of *Representation_Tensor*

brauer_character()

Return the Brauer character of `self`.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(GF(2), 5)
sage: SM = SGA.specht_module([3, 2])
sage: SM.brauer_character()
(5, -1, 0)
sage: SM.simple_module().brauer_character()
(4, -2, -1)

sage: T = SM.subrepresentation([])
sage: T.brauer_character()
(0, 0, 0)

sage: W = CoxeterGroup(['D', 4], implementation='permutation')
sage: R = W.reflection_representation(GF(2))
sage: R.brauer_character()
(4, 1)
sage: T = R.subrepresentation([])
sage: T.brauer_character()
(0, 0)
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(2)), Integer(5))
>>> SM = SGA.specht_module([Integer(3), Integer(2)])
>>> SM.brauer_character()
(5, -1, 0)
>>> SM.simple_module().brauer_character()
(4, -2, -1)

>>> T = SM.subrepresentation([])
>>> T.brauer_character()
(0, 0, 0)
```

(continues on next page)

(continued from previous page)

```
>>> W = CoxeterGroup(['D', Integer(4)], implementation='permutation')
>>> R = W.reflection_representation(GF(Integer(2)))
>>> R.brauer_character()
(4, 1)
>>> T = R.subrepresentation([])
>>> T.brauer_character()
(0, 0)
```

character()

Return the character of `self`.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 5)
sage: SM = SGA.specht_module([3,2])
sage: SM.character()
(5, 1, 1, -1, 1, -1, 0)
sage: matrix(SGA.specht_module(la).character() for la in Partitions(5))
[ 1  1  1  1  1  1  1]
[ 4  2  0  1 -1  0 -1]
[ 5  1  1 -1  1 -1  0]
[ 6  0 -2  0  0  0  1]
[ 5 -1  1 -1 -1  1  0]
[ 4 -2  0  1  1  0 -1]
[ 1 -1  1  1 -1 -1  1]

sage: SGA = SymmetricGroupAlgebra(QQ, SymmetricGroup(5))
sage: SM = SGA.specht_module([3,2])
sage: SM.character()
Character of Symmetric group of order 5! as a permutation group
sage: SM.character().values()
[5, 1, 1, -1, 1, -1, 0]
sage: matrix(SGA.specht_module(la).character().values() for la in␣
↪reversed(Partitions(5)))
[ 1 -1  1  1 -1 -1  1]
[ 4 -2  0  1  1  0 -1]
[ 5 -1  1 -1 -1  1  0]
[ 6  0 -2  0  0  0  1]
[ 5  1  1 -1  1 -1  0]
[ 4  2  0  1 -1  0 -1]
[ 1  1  1  1  1  1  1]
sage: SGA.group().character_table()
[ 1 -1  1  1 -1 -1  1]
[ 4 -2  0  1  1  0 -1]
[ 5 -1  1 -1 -1  1  0]
[ 6  0 -2  0  0  0  1]
[ 5  1  1 -1  1 -1  0]
[ 4  2  0  1 -1  0 -1]
[ 1  1  1  1  1  1  1]
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(QQ, Integer(5))
```

(continues on next page)

(continued from previous page)

```
>>> SM = SGA.specht_module([Integer(3),Integer(2)])
>>> SM.character()
(5, 1, 1, -1, 1, -1, 0)
>>> matrix(SGA.specht_module(la).character() for la in Partitions(Integer(5)))
[ 1  1  1  1  1  1  1]
[ 4  2  0  1 -1  0 -1]
[ 5  1  1 -1  1 -1  0]
[ 6  0 -2  0  0  0  1]
[ 5 -1  1 -1 -1  1  0]
[ 4 -2  0  1  1  0 -1]
[ 1 -1  1  1 -1 -1  1]

>>> SGA = SymmetricGroupAlgebra(QQ, SymmetricGroup(Integer(5)))
>>> SM = SGA.specht_module([Integer(3),Integer(2)])
>>> SM.character()
Character of Symmetric group of order 5! as a permutation group
>>> SM.character().values()
[5, 1, 1, -1, 1, -1, 0]
>>> matrix(SGA.specht_module(la).character().values() for la in␣
↪reversed(Partitions(Integer(5))))
[ 1 -1  1  1 -1 -1  1]
[ 4 -2  0  1  1  0 -1]
[ 5 -1  1 -1 -1  1  0]
[ 6  0 -2  0  0  0  1]
[ 5  1  1 -1  1 -1  0]
[ 4  2  0  1 -1  0 -1]
[ 1  1  1  1  1  1  1]
>>> SGA.group().character_table()
[ 1 -1  1  1 -1 -1  1]
[ 4 -2  0  1  1  0 -1]
[ 5 -1  1 -1 -1  1  0]
[ 6  0 -2  0  0  0  1]
[ 5  1  1 -1  1 -1  0]
[ 4  2  0  1 -1  0 -1]
[ 1  1  1  1  1  1  1]
```

composition_factors()

Return the composition factors of `self`.

Given a composition series $V = V_0 \subseteq V_1 \subseteq \cdots \subseteq V_\ell = 0$, the composition factor S_i is defined as V_i/V_{i+1}.

EXAMPLES:

The algorithm used here uses random elements, so we set the seed for testing purposes:

```
sage: set_random_seed(0)
sage: SGA = SymmetricGroupAlgebra(GF(3), 6)
sage: SM = SGA.specht_module([4, 1, 1])
sage: CF = SM.composition_factors()
sage: CF
(Quotient representation with basis
  {[[1, 2, 5, 6], [3], [4]], [[1, 2, 4, 6], [3], [5]], [[1, 2, 3, 6], [4],␣
↪[5]],
```

(continues on next page)

(continued from previous page)

```
   [[1, 2, 4, 5], [3], [6]], [[1, 2, 3, 5], [4], [6]], [[1, 2, 3, 4], [5],␣
↪[6]]}
  of Specht module of [4, 1, 1] over Finite Field of size 3,
 Subrepresentation with basis {0, 1, 2, 3} of Specht module
  of [4, 1, 1] over Finite Field of size 3)
sage: x = SGA.an_element()
sage: v = CF[1].an_element(); v
2*B[0] + 2*B[1]
sage: x * v
B[1] + B[2]
```

```
>>> from sage.all import *
>>> set_random_seed(Integer(0))
>>> SGA = SymmetricGroupAlgebra(GF(Integer(3)), Integer(6))
>>> SM = SGA.specht_module([Integer(4), Integer(1), Integer(1)])
>>> CF = SM.composition_factors()
>>> CF
(Quotient representation with basis
  {[[1, 2, 5, 6], [3], [4]], [[1, 2, 4, 6], [3], [5]], [[1, 2, 3, 6], [4],␣
↪[5]],
   [[1, 2, 4, 5], [3], [6]], [[1, 2, 3, 5], [4], [6]], [[1, 2, 3, 4], [5],␣
↪[6]]}
  of Specht module of [4, 1, 1] over Finite Field of size 3,
 Subrepresentation with basis {0, 1, 2, 3} of Specht module
  of [4, 1, 1] over Finite Field of size 3)
>>> x = SGA.an_element()
>>> v = CF[Integer(1)].an_element(); v
2*B[0] + 2*B[1]
>>> x * v
B[1] + B[2]
```

We reproduce the decomposition matrix for S_5 over $\mathbf{F}_2$:

```
sage: SGA = SymmetricGroupAlgebra(GF(2), 5)
sage: simples = [SGA.simple_module(la).brauer_character()
....:            for la in SGA.simple_module_parameterization()]
sage: D = []
sage: for la in Partitions(5):
....:     SM = SGA.specht_module(la)
....:     data = [CF.brauer_character() for CF in SM.composition_factors()]
....:     D.append([data.count(bc) for bc in simples])
sage: matrix(D)
[1 0 0]
[0 1 0]
[1 0 1]
[2 0 1]
[1 0 1]
[0 1 0]
[1 0 0]
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(2)), Integer(5))
```

(continues on next page)

(continued from previous page)

```
>>> simples = [SGA.simple_module(la).brauer_character()
...            for la in SGA.simple_module_parameterization()]
>>> D = []
>>> for la in Partitions(Integer(5)):
...     SM = SGA.specht_module(la)
...     data = [CF.brauer_character() for CF in SM.composition_factors()]
...     D.append([data.count(bc) for bc in simples])
>>> matrix(D)
[1 0 0]
[0 1 0]
[1 0 1]
[2 0 1]
[1 0 1]
[0 1 0]
[1 0 0]
```

composition_series()

Return a composition series of `self`.

EXAMPLES:

The algorithm used here uses random elements, so we set the seed for testing purposes:

```
sage: set_random_seed(0)
sage: G = groups.permutation.Dihedral(5)
sage: CFM = CombinatorialFreeModule(GF(2), [1, 2, 3, 4, 5],
....:                               bracket=False, prefix='e')
sage: CFM.an_element()
e3
sage: R = G.representation(CFM, lambda g, i: CFM.basis()[g(i)], side='right')
sage: CS = R.composition_series()
sage: len(CS)
3
sage: [[R(b) for b in F.basis()] for F in CS]
[[e1, e2, e3, e4, e5], [e1 + e5, e2 + e5, e3 + e5, e4 + e5], []]
sage: [F.brauer_character() for F in CS]
[(5, 0, 0), (4, -1, -1), (0, 0, 0)]
sage: [F.brauer_character() for F in R.composition_factors()]
[(1, 1, 1), (4, -1, -1)]
sage: Reg = G.regular_representation(GF(2))
sage: simple_brauer_chars = set([F.brauer_character()
....:                            for F in Reg.composition_factors()])
sage: sorted(simple_brauer_chars)
[(1, 1, 1), (4, -1, -1)]
```

```
>>> from sage.all import *
>>> set_random_seed(Integer(0))
>>> G = groups.permutation.Dihedral(Integer(5))
>>> CFM = CombinatorialFreeModule(GF(Integer(2)), [Integer(1), Integer(2),␣
↪Integer(3), Integer(4), Integer(5)],
...                               bracket=False, prefix='e')
>>> CFM.an_element()
e3
```

(continues on next page)

(continued from previous page)

```
>>> R = G.representation(CFM, lambda g, i: CFM.basis()[g(i)], side='right')
>>> CS = R.composition_series()
>>> len(CS)
3
>>> [[R(b) for b in F.basis()] for F in CS]
[[e1, e2, e3, e4, e5], [e1 + e5, e2 + e5, e3 + e5, e4 + e5], []]
>>> [F.brauer_character() for F in CS]
[(5, 0, 0), (4, -1, -1), (0, 0, 0)]
>>> [F.brauer_character() for F in R.composition_factors()]
[(1, 1, 1), (4, -1, -1)]
>>> Reg = G.regular_representation(GF(Integer(2)))
>>> simple_brauer_chars = set([F.brauer_character()
...                            for F in Reg.composition_factors()])
>>> sorted(simple_brauer_chars)
[(1, 1, 1), (4, -1, -1)]
```

exterior_power(*degree=None*)

Return the exterior power of `self`.

INPUT:

- `degree` – (optional) if given, then only consider the given degree

EXAMPLES:

```
sage: DC3 = groups.permutation.DiCyclic(3)
sage: L = DC3.regular_representation(QQ, side='left')
sage: E5 = L.exterior_power(5)
sage: E5
Exterior power representation of Left Regular Representation of
 Dicyclic group of order 12 as a permutation group over Rational Field
 in degree 5
sage: L.exterior_power()
Exterior algebra representation of Left Regular Representation of
 Dicyclic group of order 12 as a permutation group over Rational Field
```

```
>>> from sage.all import *
>>> DC3 = groups.permutation.DiCyclic(Integer(3))
>>> L = DC3.regular_representation(QQ, side='left')
>>> E5 = L.exterior_power(Integer(5))
>>> E5
Exterior power representation of Left Regular Representation of
 Dicyclic group of order 12 as a permutation group over Rational Field
 in degree 5
>>> L.exterior_power()
Exterior algebra representation of Left Regular Representation of
 Dicyclic group of order 12 as a permutation group over Rational Field
```

find_subrepresentation()

Return a nontrivial (not `self` or the trivial module) submodule of `self` or `None` if `self` is irreducible.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(GF(2), 5)
sage: SM = SGA.specht_module([3, 2])
sage: U = SM.find_subrepresentation()
sage: [SM(b) for b in U.basis()]
[S[[1, 2, 3], [4, 5]] + S[[1, 2, 4], [3, 5]]
 + S[[1, 2, 5], [3, 4]] + S[[1, 3, 4], [2, 5]]]
sage: B = U.basis()[0].lift()
sage: all(g * B == B for g in SGA.gens())
True
sage: SGA.specht_module([4, 1]).find_subrepresentation() is None
True
sage: SGA.specht_module([5]).find_subrepresentation() is None
True
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(2)), Integer(5))
>>> SM = SGA.specht_module([Integer(3), Integer(2)])
>>> U = SM.find_subrepresentation()
>>> [SM(b) for b in U.basis()]
[S[[1, 2, 3], [4, 5]] + S[[1, 2, 4], [3, 5]]
 + S[[1, 2, 5], [3, 4]] + S[[1, 3, 4], [2, 5]]]
>>> B = U.basis()[Integer(0)].lift()
>>> all(g * B == B for g in SGA.gens())
True
>>> SGA.specht_module([Integer(4), Integer(1)]).find_subrepresentation() is␣
↪None
True
>>> SGA.specht_module([Integer(5)]).find_subrepresentation() is None
True
```

invariant_module(*S=None*, ***kwargs*)

Return the submodule of `self` invariant under the action of `S`.

For a semigroup S acting on a module M, the invariant submodule is given by

$$M^S = \{m \in M : s \cdot m = m \forall s \in S\}.$$

INPUT:

- `S` – a finitely-generated semigroup (default: the semigroup this is a representation of)
- `action` – a function (default: `operator.mul`)
- `side` – `'left'` or `'right'` (default: *`side()`*); which side of `self` the elements of `S` acts

Note

Two sided actions are considered as left actions for the invariant module.

OUTPUT: *`FiniteDimensionalInvariantModule`*

EXAMPLES:

```
sage: S3 = SymmetricGroup(3)
sage: M = S3.regular_representation()
sage: I = M.invariant_module()
sage: [I.lift(b) for b in I.basis()]
[() + (2,3) + (1,2) + (1,2,3) + (1,3,2) + (1,3)]
```

```
>>> from sage.all import *
>>> S3 = SymmetricGroup(Integer(3))
>>> M = S3.regular_representation()
>>> I = M.invariant_module()
>>> [I.lift(b) for b in I.basis()]
[() + (2,3) + (1,2) + (1,2,3) + (1,3,2) + (1,3)]
```

We build the D_4-invariant representation inside of the regular representation of S_4:

```
sage: D4 = groups.permutation.Dihedral(4)
sage: S4 = SymmetricGroup(4)
sage: R = S4.regular_representation()
sage: I = R.invariant_module(D4)
sage: [I.lift(b) for b in I.basis()]
[() + (2,4) + (1,2)(3,4) + (1,2,3,4) + (1,3) + (1,3)(2,4) + (1,4,3,2) + (1,
↪4)(2,3),
 (3,4) + (2,3,4) + (1,2) + (1,2,4) + (1,3,2) + (1,3,2,4) + (1,4,3) + (1,4,2,
↪3),
 (2,3) + (2,4,3) + (1,2,3) + (1,2,4,3) + (1,3,4,2) + (1,3,4) + (1,4,2) + (1,
↪4)]
```

```
>>> from sage.all import *
>>> D4 = groups.permutation.Dihedral(Integer(4))
>>> S4 = SymmetricGroup(Integer(4))
>>> R = S4.regular_representation()
>>> I = R.invariant_module(D4)
>>> [I.lift(b) for b in I.basis()]
[() + (2,4) + (1,2)(3,4) + (1,2,3,4) + (1,3) + (1,3)(2,4) + (1,4,3,2) + (1,
↪4)(2,3),
 (3,4) + (2,3,4) + (1,2) + (1,2,4) + (1,3,2) + (1,3,2,4) + (1,4,3) + (1,4,2,
↪3),
 (2,3) + (2,4,3) + (1,2,3) + (1,2,4,3) + (1,3,4,2) + (1,3,4) + (1,4,2) + (1,
↪4)]
```

is_irreducible()

Return if `self` is an irreducible module or not.

A representation M is *irreducible* (also known as simple) if the only subrepresentations of M are the trivial module $\{0\}$ and M itself.

EXAMPLES:

```
sage: DC3 = groups.permutation.DiCyclic(3)
sage: L = DC3.regular_representation(GF(3), side='left')
sage: L.is_irreducible()
False
sage: E3 = L.exterior_power(3)
```

(continues on next page)

(continued from previous page)

```
sage: E3.is_irreducible()
False
sage: E12 = L.exterior_power(12)
sage: E12.is_irreducible()
True

sage: SGA = SymmetricGroupAlgebra(GF(2), 5)
sage: SGA.specht_module([3, 2]).is_irreducible()
False
sage: SGA.specht_module([4, 1]).is_irreducible()
True
```

```
>>> from sage.all import *
>>> DC3 = groups.permutation.DiCyclic(Integer(3))
>>> L = DC3.regular_representation(GF(Integer(3)), side='left')
>>> L.is_irreducible()
False
>>> E3 = L.exterior_power(Integer(3))
>>> E3.is_irreducible()
False
>>> E12 = L.exterior_power(Integer(12))
>>> E12.is_irreducible()
True

>>> SGA = SymmetricGroupAlgebra(GF(Integer(2)), Integer(5))
>>> SGA.specht_module([Integer(3), Integer(2)]).is_irreducible()
False
>>> SGA.specht_module([Integer(4), Integer(1)]).is_irreducible()
True
```

quotient_representation(*subrepr*, *already_echelonized=False*, ***kwds*)

Construct a quotient representation of `self` by `subrepr`.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(GF(2), 5)
sage: SM = SGA.specht_module([3, 2])
sage: v = sum(list(SM.basis())[1:])
sage: Q = SM.quotient_representation([v]); Q
Quotient representation with basis {[[1, 3, 5], [2, 4]],
 [[1, 3, 4], [2, 5]], [[1, 2, 4], [3, 5]], [[1, 2, 3], [4, 5]]}
 of Specht module of [3, 2] over Finite Field of size 2
sage: Q.is_irreducible()
True
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(2)), Integer(5))
>>> SM = SGA.specht_module([Integer(3), Integer(2)])
>>> v = sum(list(SM.basis())[Integer(1):])
>>> Q = SM.quotient_representation([v]); Q
Quotient representation with basis {[[1, 3, 5], [2, 4]],
 [[1, 3, 4], [2, 5]], [[1, 2, 4], [3, 5]], [[1, 2, 3], [4, 5]]}
```

(continues on next page)

(continued from previous page)

```
 of Specht module of [3, 2] over Finite Field of size 2
>>> Q.is_irreducible()
True
```

representation_matrix(*g*, *side=None*, *sparse=False*)

Return the matrix representation of `g` acting on `self`.

EXAMPLES:

```
sage: S3 = SymmetricGroup(3)
sage: g = S3.an_element(); g
(2,3)
sage: L = S3.regular_representation(side='left')
sage: R = S3.regular_representation(side='right')
sage: R.representation_matrix(g)
[0 0 0 1 0 0]
[0 0 0 0 0 1]
[0 0 0 0 1 0]
[1 0 0 0 0 0]
[0 0 1 0 0 0]
[0 1 0 0 0 0]
sage: L.representation_matrix(g)
[0 0 0 1 0 0]
[0 0 0 0 1 0]
[0 0 0 0 0 1]
[1 0 0 0 0 0]
[0 1 0 0 0 0]
[0 0 1 0 0 0]
sage: A = S3.algebra(ZZ)
sage: R.representation_matrix(sum(A.basis()), side='right')
[1 1 1 1 1 1]
[1 1 1 1 1 1]
[1 1 1 1 1 1]
[1 1 1 1 1 1]
[1 1 1 1 1 1]
[1 1 1 1 1 1]
```

```
>>> from sage.all import *
>>> S3 = SymmetricGroup(Integer(3))
>>> g = S3.an_element(); g
(2,3)
>>> L = S3.regular_representation(side='left')
>>> R = S3.regular_representation(side='right')
>>> R.representation_matrix(g)
[0 0 0 1 0 0]
[0 0 0 0 0 1]
[0 0 0 0 1 0]
[1 0 0 0 0 0]
[0 0 1 0 0 0]
[0 1 0 0 0 0]
>>> L.representation_matrix(g)
[0 0 0 1 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 0 0 0 1 0]
[0 0 0 0 0 1]
[1 0 0 0 0 0]
[0 1 0 0 0 0]
[0 0 1 0 0 0]
>>> A = S3.algebra(ZZ)
>>> R.representation_matrix(sum(A.basis()), side='right')
[1 1 1 1 1 1]
[1 1 1 1 1 1]
[1 1 1 1 1 1]
[1 1 1 1 1 1]
[1 1 1 1 1 1]
[1 1 1 1 1 1]
```

We verify tensor products agree:

```
sage: T = tensor([L, R])
sage: for g in S3:
....:     gL = L.representation_matrix(g, side='left')
....:     gR = R.representation_matrix(g, side='left')
....:     gT = T.representation_matrix(g, side='left')
....:     assert gL.tensor_product(gR) == gT
```

```
>>> from sage.all import *
>>> T = tensor([L, R])
>>> for g in S3:
...     gL = L.representation_matrix(g, side='left')
...     gR = R.representation_matrix(g, side='left')
...     gT = T.representation_matrix(g, side='left')
...     assert gL.tensor_product(gR) == gT
```

Some examples with Specht modules:

```
sage: SM = Partition([3,1,1]).specht_module(QQ)
sage: SM.representation_matrix(Permutation([2,1,3,5,4]))
[-1  0  1  0  1  0]
[ 0  0  0 -1 -1 -1]
[ 0  0  0  0  1  0]
[ 0 -1 -1  0  0  1]
[ 0  0  1  0  0  0]
[ 0  0  0  0  0 -1]

sage: SGA = SymmetricGroupAlgebra(QQ, 5)
sage: SM = SGA.specht_module([(0,0), (0,1), (0,2), (1,0), (2,0)])
sage: SM.representation_matrix(Permutation([2,1,3,5,4]))
[-1  0  0  0  0  0]
[ 0  0  1  0  0  0]
[ 0  1  0  0  0  0]
[ 1  0 -1  0 -1  0]
[-1 -1  0 -1  0  0]
[ 0  1  1  0  0 -1]
sage: SM.representation_matrix(SGA([3,1,5,2,4]))
```

(continues on next page)

(continued from previous page)

```
[ 0  0  0  0  1  0]
[-1  0  0  0  0  0]
[ 0  0  0 -1  0  0]
[ 1  0 -1  0 -1  0]
[ 0  0  0  1  1  1]
[-1 -1  0 -1  0  0]

sage: SGA = SymmetricGroupAlgebra(QQ, 4)
sage: SM = SGA.specht_module([3, 1])
sage: all(SM.representation_matrix(g) * b.to_vector() == (g * b).to_vector()
....:     for b in SM.basis() for g in SGA.group())
True

sage: SGA = SymmetricGroupAlgebra(QQ, SymmetricGroup(4))
sage: SM = SGA.specht_module([3, 1])
sage: all(SM.representation_matrix(g) * b.to_vector() == (g * b).to_vector()
....:     for b in SM.basis() for g in SGA.group())
True
```

```
>>> from sage.all import *
>>> SM = Partition([Integer(3),Integer(1),Integer(1)]).specht_module(QQ)
>>> SM.representation_matrix(Permutation([Integer(2),Integer(1),Integer(3),
↪Integer(5),Integer(4)]))
[-1  0  1  0  1  0]
[ 0  0  0 -1 -1 -1]
[ 0  0  0  0  1  0]
[ 0 -1 -1  0  0  1]
[ 0  0  1  0  0  0]
[ 0  0  0  0  0 -1]

>>> SGA = SymmetricGroupAlgebra(QQ, Integer(5))
>>> SM = SGA.specht_module([(Integer(0),Integer(0)), (Integer(0),Integer(1)),␣
↪(Integer(0),Integer(2)), (Integer(1),Integer(0)), (Integer(2),Integer(0))])
>>> SM.representation_matrix(Permutation([Integer(2),Integer(1),Integer(3),
↪Integer(5),Integer(4)]))
[-1  0  0  0  0  0]
[ 0  0  1  0  0  0]
[ 0  1  0  0  0  0]
[ 1  0 -1  0 -1  0]
[-1 -1  0 -1  0  0]
[ 0  1  1  0  0 -1]
>>> SM.representation_matrix(SGA([Integer(3),Integer(1),Integer(5),Integer(2),
↪Integer(4)]))
[ 0  0  0  0  1  0]
[-1  0  0  0  0  0]
[ 0  0  0 -1  0  0]
[ 1  0 -1  0 -1  0]
[ 0  0  0  1  1  1]
[-1 -1  0 -1  0  0]

>>> SGA = SymmetricGroupAlgebra(QQ, Integer(4))
>>> SM = SGA.specht_module([Integer(3), Integer(1)])
```

(continues on next page)

(continued from previous page)

```
>>> all(SM.representation_matrix(g) * b.to_vector() == (g * b).to_vector()
...     for b in SM.basis() for g in SGA.group())
True

>>> SGA = SymmetricGroupAlgebra(QQ, SymmetricGroup(Integer(4)))
>>> SM = SGA.specht_module([Integer(3), Integer(1)])
>>> all(SM.representation_matrix(g) * b.to_vector() == (g * b).to_vector()
...     for b in SM.basis() for g in SGA.group())
True
```

schur_functor(*la*)

Return the *Schur functor* with shape `la` applied to `self`.

EXAMPLES:

```
sage: W = CoxeterGroup(['H', 3])
sage: R = W.reflection_representation()
sage: S111 = R.schur_functor([1,1,1])
sage: S111.dimension()
1
sage: S3 = R.schur_functor([3])
sage: S3.dimension()
10
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(['H', Integer(3)])
>>> R = W.reflection_representation()
>>> S111 = R.schur_functor([Integer(1),Integer(1),Integer(1)])
>>> S111.dimension()
1
>>> S3 = R.schur_functor([Integer(3)])
>>> S3.dimension()
10
```

semigroup()

Return the semigroup whose representation `self` is.

EXAMPLES:

```
sage: G = SymmetricGroup(4)
sage: M = CombinatorialFreeModule(QQ, ['v'])
sage: on_basis = lambda g,m: M.term(m, g.sign())
sage: R = G.representation(M, on_basis)
sage: R.semigroup()
Symmetric group of order 4! as a permutation group
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(4))
>>> M = CombinatorialFreeModule(QQ, ['v'])
>>> on_basis = lambda g,m: M.term(m, g.sign())
>>> R = G.representation(M, on_basis)
>>> R.semigroup()
Symmetric group of order 4! as a permutation group
```

semigroup_algebra()

Return the semigroup algebra whose representation `self` is.

EXAMPLES:

```
sage: G = SymmetricGroup(4)
sage: M = CombinatorialFreeModule(QQ, ['v'])
sage: on_basis = lambda g,m: M.term(m, g.sign())
sage: R = G.representation(M, on_basis)
sage: R.semigroup_algebra()
Symmetric group algebra of order 4 over Rational Field
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(4))
>>> M = CombinatorialFreeModule(QQ, ['v'])
>>> on_basis = lambda g,m: M.term(m, g.sign())
>>> R = G.representation(M, on_basis)
>>> R.semigroup_algebra()
Symmetric group algebra of order 4 over Rational Field
```

side()

Return whether `self` is a left, right, or two-sided representation.

OUTPUT: the string `'left'`, `'right'`, or `'twosided'`

EXAMPLES:

```
sage: G = groups.permutation.Dihedral(4)
sage: R = G.regular_representation()
sage: R.side()
'left'
sage: S = G.regular_representation(side='right')
sage: S.side()
'right'
sage: R = G.sign_representation()
sage: R.side()
'twosided'
sage: R = G.trivial_representation()
sage: R.side()
'twosided'
```

```
>>> from sage.all import *
>>> G = groups.permutation.Dihedral(Integer(4))
>>> R = G.regular_representation()
>>> R.side()
'left'
>>> S = G.regular_representation(side='right')
>>> S.side()
'right'
>>> R = G.sign_representation()
>>> R.side()
'twosided'
>>> R = G.trivial_representation()
>>> R.side()
'twosided'
```

subrepresentation(*gens*, *check*, *already_echelonized=True*, *is_closed=False*, **args*, ***opts*)

Construct a subrepresentation of `self` generated by `gens`.

INPUT:

- `gens` – the generators of the submodule
- `check` – ignored
- `already_echelonized` – (default: `False`) whether the elements of `gens` are already in (not necessarily reduced) echelon form
- `is_closed` – (keyword only; default: `False`) whether `gens` already spans the subspace closed under the semigroup action

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(GF(2), 5)
sage: SM = SGA.specht_module([3, 2])
sage: B = next(iter(SM.basis()))
sage: U = SM.subrepresentation([B])
sage: U.dimension()
5
```

```
>>> from sage.all import *
>>> SGA = SymmetricGroupAlgebra(GF(Integer(2)), Integer(5))
>>> SM = SGA.specht_module([Integer(3), Integer(2)])
>>> B = next(iter(SM.basis()))
>>> U = SM.subrepresentation([B])
>>> U.dimension()
5
```

symmetric_power(*degree=None*)

Return the symmetric power of `self` in degree `degree`.

EXAMPLES:

```
sage: W = CoxeterGroup(['H', 3])
sage: R = W.reflection_representation()
sage: S3R = R.symmetric_power(3)
sage: S3R
Symmetric power representation of Reflection representation of
Finite Coxeter group over ... with Coxeter matrix:
[1 3 2]
[3 1 5]
[2 5 1] in degree 3
```

```
>>> from sage.all import *
>>> W = CoxeterGroup(['H', Integer(3)])
>>> R = W.reflection_representation()
>>> S3R = R.symmetric_power(Integer(3))
>>> S3R
Symmetric power representation of Reflection representation of
Finite Coxeter group over ... with Coxeter matrix:
[1 3 2]
[3 1 5]
[2 5 1] in degree 3
```

twisted_invariant_module(*chi*, *G=None*, ***kwargs*)

Create the isotypic component of the action of `G` on `self` with irreducible character given by `chi`.

See also

- *FiniteDimensionalTwistedInvariantModule*

INPUT:

- `chi` – list/tuple of character values or an instance of `ClassFunction_gap`
- `G` – a finitely-generated semigroup (default: the semigroup this is a representation of)

This also accepts the first argument to be the group.

OUTPUT: *FiniteDimensionalTwistedInvariantModule*

EXAMPLES:

```
sage: G = SymmetricGroup(3)
sage: R = G.regular_representation(QQ)
sage: T = R.twisted_invariant_module([2,0,-1])
sage: T.basis()
Finite family {0: B[0], 1: B[1], 2: B[2], 3: B[3]}
sage: [T.lift(b) for b in T.basis()]
[() - (1,2,3), -(1,2,3) + (1,3,2), (2,3) - (1,2), -(1,2) + (1,3)]
```

```
>>> from sage.all import *
>>> G = SymmetricGroup(Integer(3))
>>> R = G.regular_representation(QQ)
>>> T = R.twisted_invariant_module([Integer(2),Integer(0),-Integer(1)])
>>> T.basis()
Finite family {0: B[0], 1: B[1], 2: B[2], 3: B[3]}
>>> [T.lift(b) for b in T.basis()]
[() - (1,2,3), -(1,2,3) + (1,3,2), (2,3) - (1,2), -(1,2) + (1,3)]
```

We check the different inputs work:

```
sage: R.twisted_invariant_module([2,0,-1], G) is T
True
sage: R.twisted_invariant_module(G, [2,0,-1]) is T
True
```

```
>>> from sage.all import *
>>> R.twisted_invariant_module([Integer(2),Integer(0),-Integer(1)], G) is T
True
>>> R.twisted_invariant_module(G, [Integer(2),Integer(0),-Integer(1)]) is T
True
```

class sage.modules.with_basis.representation.**SchurFunctorRepresentation**(V, *shape*)

Bases: *Subrepresentation*

The representation constructed by the Schur functor.

Let G be a semigroup and let V be a representation of G. The *Schur functor* for a partition λ of size k is the functor $\mathbb{S}_\lambda$ that sends V to the G-subrepresentation of $V^{\otimes k}$ spanned by $(v_1 \otimes \cdots \otimes v_k)c_\lambda$, where c_λ is the `Young`

`symmetrizer` corresponding to λ. When $G = GL_n(F)$, the Schur functor image $\mathbb{S}_\lambda F^n$ is the (irreducible when F has characteristic 0) highest representation of shape λ.

EXAMPLES:

```
sage: G = groups.permutation.Dihedral(3)
sage: V = G.regular_representation()
sage: S21V = V.schur_functor([2, 1])
sage: S21V.dimension()
70
sage: SemistandardTableaux([2,1], max_entry=V.dimension()).cardinality()
70
sage: chi = G.character([S21V.representation_matrix(g).trace()
....:                    for g in G.conjugacy_classes_representatives()])
sage: chi.values()
[70, 0, -2]
sage: G.character_table()
[ 1 -1  1]
[ 2  0 -1]
[ 1  1  1]
sage: for m, phi in chi.decompose():
....:     print(m, phi.values())
11 [1, -1, 1]
11 [1, 1, 1]
24 [2, 0, -1]

sage: # long time
sage: S211V = V.schur_functor([2, 1, 1])
sage: S211V.dimension()
105
sage: SemistandardTableaux([2, 1, 1], max_entry=V.dimension()).cardinality()
105
sage: chi = G.character([S211V.representation_matrix(g).trace()
....:                    for g in G.conjugacy_classes_representatives()])
sage: chi.values()
[105, -3, 0]
sage: for m, phi in chi.decompose():
....:     print(m, phi.values())
19 [1, -1, 1]
16 [1, 1, 1]
35 [2, 0, -1]
```

```
>>> from sage.all import *
>>> G = groups.permutation.Dihedral(Integer(3))
>>> V = G.regular_representation()
>>> S21V = V.schur_functor([Integer(2), Integer(1)])
>>> S21V.dimension()
70
>>> SemistandardTableaux([Integer(2),Integer(1)], max_entry=V.dimension()).
↪cardinality()
70
>>> chi = G.character([S21V.representation_matrix(g).trace()
...                    for g in G.conjugacy_classes_representatives()])
>>> chi.values()
```

(continues on next page)

(continued from previous page)

```
[70, 0, -2]
>>> G.character_table()
[ 1 -1  1]
[ 2  0 -1]
[ 1  1  1]
>>> for m, phi in chi.decompose():
...     print(m, phi.values())
11 [1, -1, 1]
11 [1, 1, 1]
24 [2, 0, -1]

>>> # long time
>>> S211V = V.schur_functor([Integer(2), Integer(1), Integer(1)])
>>> S211V.dimension()
105
>>> SemistandardTableaux([Integer(2), Integer(1), Integer(1)], max_entry=V.
↪dimension()).cardinality()
105
>>> chi = G.character([S211V.representation_matrix(g).trace()
...                    for g in G.conjugacy_classes_representatives()])
>>> chi.values()
[105, -3, 0]
>>> for m, phi in chi.decompose():
...     print(m, phi.values())
19 [1, -1, 1]
16 [1, 1, 1]
35 [2, 0, -1]
```

An example with the cyclic group:

```
sage: C3 = groups.permutation.Cyclic(3)
sage: V3 = C3.regular_representation()
sage: S31V3 = V3.schur_functor([3, 1])
sage: S31V3.dimension()
15
sage: chi = C3.character([S31V3.representation_matrix(g).trace()
....:                     for g in C3.conjugacy_classes_representatives()])
sage: chi.values()
[15, 0, 0]
sage: C3.character_table()
[         1          1          1]
[         1       zeta3 -zeta3 - 1]
[         1 -zeta3 - 1       zeta3]
sage: for m, phi in chi.decompose():
....:     print(m, phi.values())
5 [1, 1, 1]
5 [1, -zeta3 - 1, zeta3]
5 [1, zeta3, -zeta3 - 1]
```

```
>>> from sage.all import *
>>> C3 = groups.permutation.Cyclic(Integer(3))
>>> V3 = C3.regular_representation()
```

(continues on next page)

(continued from previous page)

```
>>> S31V3 = V3.schur_functor([Integer(3), Integer(1)])
>>> S31V3.dimension()
15
>>> chi = C3.character([S31V3.representation_matrix(g).trace()
...                     for g in C3.conjugacy_classes_representatives()])
>>> chi.values()
[15, 0, 0]
>>> C3.character_table()
[         1          1          1]
[         1      zeta3 -zeta3 - 1]
[         1 -zeta3 - 1      zeta3]
>>> for m, phi in chi.decompose():
...     print(m, phi.values())
5 [1, 1, 1]
5 [1, -zeta3 - 1, zeta3]
5 [1, zeta3, -zeta3 - 1]
```

An example of $GL_3(\mathbf{F}_2)$:

```
sage: G = groups.matrix.GL(3, 2)
sage: from sage.modules.with_basis.representation import Representation
sage: N = G.natural_representation()
sage: S21N = N.schur_functor([2, 1])
sage: S21N.dimension()
8
```

```
>>> from sage.all import *
>>> G = groups.matrix.GL(Integer(3), Integer(2))
>>> from sage.modules.with_basis.representation import Representation
>>> N = G.natural_representation()
>>> S21N = N.schur_functor([Integer(2), Integer(1)])
>>> S21N.dimension()
8
```

An example with the Weyl/Coxeter group of type C_3:

```
sage: G = WeylGroup(['C', 3], prefix='s')
sage: R = G.reflection_representation()
sage: S = R.schur_functor([3, 2, 1])
sage: g = G.an_element(); g
s1*s2*s3
sage: v = S.an_element(); v
2*S[0] + 2*S[1] + 3*S[2]
sage: v * g
-(2*a+1)*S[0] - (a+2)*S[1] - (2*a-2)*S[2] + (2*a-2)*S[3]
 - (-2*a+4)*S[4] + (-2*a+4)*S[5] + 2*S[6] + 2*a*S[7]
sage: g * v
3*S[0] + (-2*a+5)*S[2] + 3*a*S[4] - (5*a-2)*S[6] - 6*S[7]
```

```
>>> from sage.all import *
>>> G = WeylGroup(['C', Integer(3)], prefix='s')
>>> R = G.reflection_representation()
```

(continues on next page)

(continued from previous page)

```
>>> S = R.schur_functor([Integer(3), Integer(2), Integer(1)])
>>> g = G.an_element(); g
s1*s2*s3
>>> v = S.an_element(); v
2*S[0] + 2*S[1] + 3*S[2]
>>> v * g
-(2*a+1)*S[0] - (a+2)*S[1] - (2*a-2)*S[2] + (2*a-2)*S[3]
 - (-2*a+4)*S[4] + (-2*a+4)*S[5] + 2*S[6] + 2*a*S[7]
>>> g * v
3*S[0] + (-2*a+5)*S[2] + 3*a*S[4] - (5*a-2)*S[6] - 6*S[7]
```

Element

alias of *Element*

class sage.modules.with_basis.representation.**SignRepresentationCoxeterGroup**(*group*, *base_ring*, *sign_function=None*)

Bases: *SignRepresentation_abstract*

The sign representation for a Coxeter group.

EXAMPLES:

```
sage: G = WeylGroup(["A", 1, 1])
sage: V = G.sign_representation()
sage: TestSuite(V).run()

sage: # optional - gap3
sage: W = CoxeterGroup(['B', 3], implementation="coxeter3")
sage: S = W.sign_representation()
sage: TestSuite(S).run()
```

```
>>> from sage.all import *
>>> G = WeylGroup(["A", Integer(1), Integer(1)])
>>> V = G.sign_representation()
>>> TestSuite(V).run()

>>> # optional - gap3
>>> W = CoxeterGroup(['B', Integer(3)], implementation="coxeter3")
>>> S = W.sign_representation()
>>> TestSuite(S).run()
```

class sage.modules.with_basis.representation.**SignRepresentationMatrixGroup**(*group*, *base_ring*, *sign_function=None*)

Bases: *SignRepresentation_abstract*

The sign representation for a matrix group.

EXAMPLES:

```
sage: G = groups.permutation.PGL(2, 3)
sage: V = G.sign_representation()
sage: TestSuite(V).run()
```

```
>>> from sage.all import *
>>> G = groups.permutation.PGL(Integer(2), Integer(3))
>>> V = G.sign_representation()
>>> TestSuite(V).run()
```

class sage.modules.with_basis.representation.**SignRepresentationPermgroup**(*group*, *base_ring*, *sign_function=None*)

Bases: *SignRepresentation_abstract*

The sign representation for a permutation group.

EXAMPLES:

```
sage: G = groups.permutation.PGL(2, 3)
sage: V = G.sign_representation()
sage: TestSuite(V).run()
```

```
>>> from sage.all import *
>>> G = groups.permutation.PGL(Integer(2), Integer(3))
>>> V = G.sign_representation()
>>> TestSuite(V).run()
```

class sage.modules.with_basis.representation.**SignRepresentation_abstract**(*group*, *base_ring*, *sign_function=None*)

Bases: *Representation_abstract*, CombinatorialFreeModule

Generic implementation of a sign representation.

The sign representation of a semigroup S over a commutative ring R is the 1-dimensional R-module on which every element of S acts by 1 if order of element is even (including 0) or -1 if order of element if odd.

This is simultaneously a left and right representation.

INPUT:

- `permgroup` – a permgroup
- `base_ring` – the base ring for the representation
- `sign_function` – a function which returns 1 or -1 depending on the elements sign

REFERENCES:

- Wikipedia article Representation_theory_of_the_symmetric_group

class sage.modules.with_basis.representation.**Subrepresentation**(*basis*, *support_order*, *ambient*, **args*, ***opts*)

Bases: *Representation_abstract*, *SubmoduleWithBasis*

A subrepresentation.

Let R be a representation of an algebraic object X. A subrepresentation is a submodule of R that is closed under the action of X.

class Element

Bases: *IndexedFreeModuleElement*

class sage.modules.with_basis.representation.**TrivialRepresentation**(*semigroup*, *base_ring*)

Bases: *Representation_abstract*, CombinatorialFreeModule

The trivial representation of a semigroup.

The trivial representation of a semigroup S over a commutative ring R is the 1-dimensional R-module on which every element of S acts by the identity.

This is simultaneously a left and right representation.

INPUT:

- semigroup – a semigroup
- base_ring – the base ring for the representation

REFERENCES:

- Wikipedia article Trivial_representation

class Element

Bases: *Element*

CHAPTER

FOUR

FINITELY GENERATED MODULES OVER A PID

4.1 Finitely generated modules over a PID

You can use Sage to compute with finitely generated modules (FGM's) over a principal ideal domain R presented as a quotient V/W, where V and W are free.

Note

Currently this is only enabled over `R=ZZ`, since it has not been tested and debugged over more general PIDs. All algorithms make sense whenever there is a Hermite form implementation. In theory the obstruction to extending the implementation is only that one has to decide how elements print.

We represent $M = V/W$ as a pair (V, W) with W contained in V, and we internally represent elements of M non-canonically as elements x of V. We also fix independent generators `g[i]` for M in V, and when we print out elements of V we print their coordinates with respect to the `g[i]`; over $\mathbf{Z}$ this is canonical, since each coefficient is reduced modulo the additive order of `g[i]`. To obtain the vector in V corresponding to x in M, use `x.lift()`.

Morphisms between finitely generated R-modules are well supported. You create a homomorphism by simply giving the images of generators of M_0 in M_1. Given a morphism $\phi : M_0 \to M_1$, you can compute the image of ϕ, the kernel of ϕ, and using `y = phi.lift(x)` you can lift an element x in M_1 to an element y in M_0, if such a y exists.

TECHNICAL NOTE: For efficiency, we introduce a notion of optimized representation for quotient modules. The optimized representation of $M = V/W$ is the quotient V'/W' where V' has as basis lifts of the generators `g[i]` for M. We internally store a morphism from $M_0 = V_0/W_0$ to $M_1 = V_1/W_1$ by giving a morphism from the optimized representation V_0' of M_0 to V_1 that sends W_0 into W_1.

The following TUTORIAL illustrates several of the above points.

First we create free modules V_0 and W_0 and the quotient module M_0. Notice that everything works fine even though V_0 and W_0 are not contained inside $\mathbf{Z}^n$, which is extremely convenient.

```
sage: V0 = span([[1/2,0,0], [3/2,2,1], [0,0,1]], ZZ)
sage: W0 = V0.span([V0.0 + 2*V0.1, 9*V0.0 + 2*V0.1, 4*V0.2])
sage: M0 = V0/W0; M0
Finitely generated module V/W over Integer Ring with invariants (4, 16)
```

```
>>> from sage.all import *
>>> V0 = span([[Integer(1)/Integer(2),Integer(0),Integer(0)], [Integer(3)/Integer(2),
...Integer(2),Integer(1)], [Integer(0),Integer(0),Integer(1)]], ZZ)
>>> W0 = V0.span([V0.gen(0) + Integer(2)*V0.gen(1), Integer(9)*V0.gen(0) +␣
...Integer(2)*V0.gen(1), Integer(4)*V0.gen(2)])
```

(continues on next page)

(continued from previous page)

```
>>> M0 = V0/W0; M0
Finitely generated module V/W over Integer Ring with invariants (4, 16)
```

The invariants are computed using the Smith normal form algorithm, and determine the structure of this finitely generated module.

You can get the V and W used in constructing the quotient module using the methods `V()` and `W()`:

```
sage: M0.V()
Free module of degree 3 and rank 3 over Integer Ring
Echelon basis matrix:
[1/2   0   0]
[  0   2   0]
[  0   0   1]
sage: M0.W()
Free module of degree 3 and rank 3 over Integer Ring
Echelon basis matrix:
[1/2   4   0]
[  0  32   0]
[  0   0   4]
```

```
>>> from sage.all import *
>>> M0.V()
Free module of degree 3 and rank 3 over Integer Ring
Echelon basis matrix:
[1/2   0   0]
[  0   2   0]
[  0   0   1]
>>> M0.W()
Free module of degree 3 and rank 3 over Integer Ring
Echelon basis matrix:
[1/2   4   0]
[  0  32   0]
[  0   0   4]
```

We note that the optimized representation of M_0, mentioned above in the technical note, has a V that need not be equal to V_0, in general.

```
sage: M0.optimized()[0].V()
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[ 0  8  1]
[ 0 -2  0]
```

```
>>> from sage.all import *
>>> M0.optimized()[Integer(0)].V()
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[ 0  8  1]
[ 0 -2  0]
```

Create elements of M_0 either by coercing in elements of V_0, getting generators, or coercing in a list or tuple or coercing in 0. Finally, one can express an element as a linear combination of the Smith form generators

```
sage: M0(V0.0)
(0, 2)
sage: M0(V0.0 + W0.0)  # no difference modulo W0
(0, 2)
sage: M0.linear_combination_of_smith_form_gens([3,20])
(3, 4)
sage: 3*M0.0 + 20*M0.1
(3, 4)
```

```
>>> from sage.all import *
>>> M0(V0.gen(0))
(0, 2)
>>> M0(V0.gen(0) + W0.gen(0))  # no difference modulo W0
(0, 2)
>>> M0.linear_combination_of_smith_form_gens([Integer(3),Integer(20)])
(3, 4)
>>> Integer(3)*M0.gen(0) + Integer(20)*M0.gen(1)
(3, 4)
```

We make an element of M_0 by taking a difference of two generators, and lift it. We also illustrate making an element from a list, which coerces to V_0, then take the equivalence class modulo W_0.

```
sage: x = M0.0 - M0.1; x
(1, 15)
sage: x.lift()
(0, 10, 1)
sage: M0(vector([1/2,0,0]))
(0, 2)
sage: x.additive_order()
16
```

```
>>> from sage.all import *
>>> x = M0.gen(0) - M0.gen(1); x
(1, 15)
>>> x.lift()
(0, 10, 1)
>>> M0(vector([Integer(1)/Integer(2),Integer(0),Integer(0)]))
(0, 2)
>>> x.additive_order()
16
```

Similarly, we construct V_1 and W_1, and the quotient M_1, in a completely different 2-dimensional ambient space.

```
sage: V1 = span([[1/2,0], [3/2,2]], ZZ); W1 = V1.span([2*V1.0, 3*V1.1])
sage: M1 = V1/W1; M1
Finitely generated module V/W over Integer Ring with invariants (6)
```

```
>>> from sage.all import *
>>> V1 = span([[Integer(1)/Integer(2),Integer(0)], [Integer(3)/Integer(2),
↪Integer(2)]], ZZ); W1 = V1.span([Integer(2)*V1.gen(0), Integer(3)*V1.gen(1)])
>>> M1 = V1/W1; M1
Finitely generated module V/W over Integer Ring with invariants (6)
```

We create the homomorphism from M_0 to M_1 that sends both generators of M_0 to 3 times the generator of M_1. This is well-defined since 3 times the generator has order 2.

```
sage: f = M0.hom([3*M1.0, 3*M1.0]); f
Morphism from module over Integer Ring with invariants (4, 16)
 to module with invariants (6,) that sends the generators to [(3), (3)]
```

```
>>> from sage.all import *
>>> f = M0.hom([Integer(3)*M1.gen(0), Integer(3)*M1.gen(0)]); f
Morphism from module over Integer Ring with invariants (4, 16)
 to module with invariants (6,) that sends the generators to [(3), (3)]
```

We evaluate the homomorphism on our element x of the domain, and on the first generator of the domain. We also evaluate at an element of V_0, which is coerced into M_0.

```
sage: f(x)
(0)
sage: f(M0.0)
(3)
sage: f(V0.1)
(3)
```

```
>>> from sage.all import *
>>> f(x)
(0)
>>> f(M0.gen(0))
(3)
>>> f(V0.gen(1))
(3)
```

Here we illustrate lifting an element of the image of f, i.e., finding an element of M_0 that maps to a given element of M_1:

```
sage: y = f.lift(3*M1.0)
sage: y # random
(0, 13)
sage: f(y)
(3)
```

```
>>> from sage.all import *
>>> y = f.lift(Integer(3)*M1.gen(0))
>>> y # random
(0, 13)
>>> f(y)
(3)
```

We compute the kernel of f, i.e., the submodule of elements of M_0 that map to 0. Note that the kernel is not explicitly represented as a submodule, but as another quotient V/W where V is contained in V_0. You can explicitly coerce elements of the kernel into M_0 though.

```
sage: K = f.kernel(); K
Finitely generated module V/W over Integer Ring with invariants (2, 16)
```

(continues on next page)

(continued from previous page)

```
sage: M0(K.0)
(2, 8)
sage: M0(K.1)
(1, 5)
sage: f(M0(K.0))
(0)
sage: f(M0(K.1))
(0)
```

```
>>> from sage.all import *
>>> K = f.kernel(); K
Finitely generated module V/W over Integer Ring with invariants (2, 16)

>>> M0(K.gen(0))
(2, 8)
>>> M0(K.gen(1))
(1, 5)
>>> f(M0(K.gen(0)))
(0)
>>> f(M0(K.gen(1)))
(0)
```

We compute the image of f.

```
sage: f.image()
Finitely generated module V/W over Integer Ring with invariants (2)
```

```
>>> from sage.all import *
>>> f.image()
Finitely generated module V/W over Integer Ring with invariants (2)
```

Notice how the elements of the image are written as (0) and (1), despite the image being naturally a submodule of M_1, which has elements (0), (1), (2), (3), (4), (5). However, below we coerce the element (1) of the image into the codomain, and get (3):

```
sage: list(f.image())
[(0), (1)]
sage: list(M1)
[(0), (1), (2), (3), (4), (5)]
sage: x = f.image().0; x
(1)
sage: M1(x)
(3)
```

```
>>> from sage.all import *
>>> list(f.image())
[(0), (1)]
>>> list(M1)
[(0), (1), (2), (3), (4), (5)]
>>> x = f.image().gen(0); x
(1)
```

(continues on next page)

(continued from previous page)

```
>>> M1(x)
(3)
```

AUTHOR:

- William Stein, 2009

sage.modules.fg_pid.fgp_module.**FGP_Module**(*V*, *W*, *check=True*)

INPUT:

- `V` – a free R-module
- `W` – a free R-submodule of V
- `check` – boolean (default: `True`); if `True`, more checks on correctness are performed. In particular, we check the data types of `V` and `W`, and that W is a submodule of V with the same base ring.

OUTPUT: the quotient V/W as a finitely generated R-module

EXAMPLES:

```
sage: V = span([[1/2,1,1], [3/2,2,1], [0,0,1]], ZZ)
sage: W = V.span([2*V.0 + 4*V.1, 9*V.0 + 12*V.1, 4*V.2])
sage: import sage.modules.fg_pid.fgp_module
sage: Q = sage.modules.fg_pid.fgp_module.FGP_Module(V, W)
sage: type(Q)
<class 'sage.modules.fg_pid.fgp_module.FGP_Module_class_with_category'>
sage: Q is sage.modules.fg_pid.fgp_module.FGP_Module(V, W, check=False)
True
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)], [Integer(3)/
↪Integer(2),Integer(2),Integer(1)], [Integer(0),Integer(0),Integer(1)]], ZZ)
>>> W = V.span([Integer(2)*V.gen(0) + Integer(4)*V.gen(1), Integer(9)*V.gen(0) +␣
↪Integer(12)*V.gen(1), Integer(4)*V.gen(2)])
>>> import sage.modules.fg_pid.fgp_module
>>> Q = sage.modules.fg_pid.fgp_module.FGP_Module(V, W)
>>> type(Q)
<class 'sage.modules.fg_pid.fgp_module.FGP_Module_class_with_category'>
>>> Q is sage.modules.fg_pid.fgp_module.FGP_Module(V, W, check=False)
True
```

class sage.modules.fg_pid.fgp_module.**FGP_Module_class**(*V*, *W*, *check=True*)

Bases: *Module*

A finitely generated module over a PID presented as a quotient V/W.

INPUT:

- `V` – an R-module
- `W` – an R-submodule of V
- `check` – boolean (default: `True`)

EXAMPLES:

```
sage: A = (ZZ^1)/span([[100]], ZZ); A
Finitely generated module V/W over Integer Ring with invariants (100)
sage: A.V()
Ambient free module of rank 1 over the principal ideal domain Integer Ring
sage: A.W()
Free module of degree 1 and rank 1 over Integer Ring
Echelon basis matrix:
[100]

sage: V = span([[1/2,1,1], [3/2,2,1], [0,0,1]], ZZ)
sage: W = V.span([2*V.0 + 4*V.1, 9*V.0 + 12*V.1, 4*V.2])
sage: Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 12)
sage: type(Q)
<class 'sage.modules.fg_pid.fgp_module.FGP_Module_class_with_category'>
```

```
>>> from sage.all import *
>>> A = (ZZ**Integer(1))/span([[Integer(100)]], ZZ); A
Finitely generated module V/W over Integer Ring with invariants (100)
>>> A.V()
Ambient free module of rank 1 over the principal ideal domain Integer Ring
>>> A.W()
Free module of degree 1 and rank 1 over Integer Ring
Echelon basis matrix:
[100]

>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)], [Integer(3)/
↪Integer(2),Integer(2),Integer(1)], [Integer(0),Integer(0),Integer(1)]], ZZ)
>>> W = V.span([Integer(2)*V.gen(0) + Integer(4)*V.gen(1), Integer(9)*V.gen(0) +␣
↪Integer(12)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 12)
>>> type(Q)
<class 'sage.modules.fg_pid.fgp_module.FGP_Module_class_with_category'>
```

Element

alias of *FGP_Element*

V()

If this module was constructed as a quotient V/W, return V.

EXAMPLES:

```
sage: V = span([[1/2,1,1], [3/2,2,1], [0,0,1]], ZZ)
sage: W = V.span([2*V.0 + 4*V.1, 9*V.0 + 12*V.1, 4*V.2])
sage: Q = V/W
sage: Q.V()
Free module of degree 3 and rank 3 over Integer Ring
Echelon basis matrix:
[1/2   0   0]
[  0   1   0]
[  0   0   1]
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)], [Integer(3)/
↪Integer(2),Integer(2),Integer(1)], [Integer(0),Integer(0),Integer(1)]], ZZ)
>>> W = V.span([Integer(2)*V.gen(0) + Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0) + Integer(12)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q = V/W
>>> Q.V()
Free module of degree 3 and rank 3 over Integer Ring
Echelon basis matrix:
[1/2   0   0]
[  0   1   0]
[  0   0   1]
```

W()

If this module was constructed as a quotient V/W, return W.

EXAMPLES:

```
sage: V = span([[1/2,1,1], [3/2,2,1], [0,0,1]], ZZ)
sage: W = V.span([2*V.0 + 4*V.1, 9*V.0 + 12*V.1, 4*V.2])
sage: Q = V/W
sage: Q.W()
Free module of degree 3 and rank 3 over Integer Ring
Echelon basis matrix:
[1/2   8   0]
[  0  12   0]
[  0   0   4]
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)], [Integer(3)/
↪Integer(2),Integer(2),Integer(1)], [Integer(0),Integer(0),Integer(1)]], ZZ)
>>> W = V.span([Integer(2)*V.gen(0) + Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0) + Integer(12)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q = V/W
>>> Q.W()
Free module of degree 3 and rank 3 over Integer Ring
Echelon basis matrix:
[1/2   8   0]
[  0  12   0]
[  0   0   4]
```

annihilator()

Return the ideal of the base ring that annihilates `self`. This is precisely the ideal generated by the LCM of the invariants of `self` if `self` is finite, and is 0 otherwise.

EXAMPLES:

```
sage: V = span([[1/2,0,0], [3/2,2,1], [0,0,1]], ZZ)
sage: W = V.span([V.0 + 2*V.1, 9*V.0 + 2*V.1, 4*V.2])
sage: Q = V/W; Q.annihilator()
Principal ideal (16) of Integer Ring
sage: Q.annihilator().gen()
16
```

(continues on next page)

(continued from previous page)

```
sage: Q = V / V.span([V.0]); Q
Finitely generated module V/W over Integer Ring with invariants (0, 0)
sage: Q.annihilator()
Principal ideal (0) of Integer Ring
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(0),Integer(0)], [Integer(3)/
↪Integer(2),Integer(2),Integer(1)], [Integer(0),Integer(0),Integer(1)]], ZZ)
>>> W = V.span([V.gen(0) + Integer(2)*V.gen(1), Integer(9)*V.gen(0) +␣
↪Integer(2)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q = V/W; Q.annihilator()
Principal ideal (16) of Integer Ring
>>> Q.annihilator().gen()
16

>>> Q = V / V.span([V.gen(0)]); Q
Finitely generated module V/W over Integer Ring with invariants (0, 0)
>>> Q.annihilator()
Principal ideal (0) of Integer Ring
```

We check that Issue #22720 is resolved:

```
sage: H = AdditiveAbelianGroup([])
sage: H.annihilator()
Principal ideal (1) of Integer Ring
```

```
>>> from sage.all import *
>>> H = AdditiveAbelianGroup([])
>>> H.annihilator()
Principal ideal (1) of Integer Ring
```

base_ring()

Return the base ring of `self`.

EXAMPLES:

```
sage: V = span([[1/2,1,1], [3/2,2,1], [0,0,1]], ZZ)
sage: W = V.span([2*V.0 + 4*V.1, 9*V.0 + 12*V.1, 4*V.2])
sage: Q = V/W
sage: Q.base_ring()
Integer Ring
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)], [Integer(3)/
↪Integer(2),Integer(2),Integer(1)], [Integer(0),Integer(0),Integer(1)]], ZZ)
>>> W = V.span([Integer(2)*V.gen(0) + Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0) + Integer(12)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q = V/W
>>> Q.base_ring()
Integer Ring
```

cardinality()

Return the cardinality of this module as a set.

EXAMPLES:

```
sage: V = ZZ^2; W = V.span([[1,2], [3,4]]); A = V/W; A
Finitely generated module V/W over Integer Ring with invariants (2)
sage: A.cardinality()
2
sage: V = ZZ^2; W = V.span([[1,2]]); A = V/W; A
Finitely generated module V/W over Integer Ring with invariants (0)
sage: A.cardinality()
+Infinity
sage: V = QQ^2; W = V.span([[1,2]]); A = V/W; A
Vector space quotient V/W of dimension 1 over Rational Field where
  V: Vector space of dimension 2 over Rational Field
  W: Vector space of degree 2 and dimension 1 over Rational Field
     Basis matrix:
     [1 2]
sage: A.cardinality()
+Infinity
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(2); W = V.span([[Integer(1),Integer(2)], [Integer(3),
↪Integer(4)]]); A = V/W; A
Finitely generated module V/W over Integer Ring with invariants (2)
>>> A.cardinality()
2
>>> V = ZZ**Integer(2); W = V.span([[Integer(1),Integer(2)]]); A = V/W; A
Finitely generated module V/W over Integer Ring with invariants (0)
>>> A.cardinality()
+Infinity
>>> V = QQ**Integer(2); W = V.span([[Integer(1),Integer(2)]]); A = V/W; A
Vector space quotient V/W of dimension 1 over Rational Field where
  V: Vector space of dimension 2 over Rational Field
  W: Vector space of degree 2 and dimension 1 over Rational Field
     Basis matrix:
     [1 2]
>>> A.cardinality()
+Infinity
```

construction()

The construction functor and ambient module for `self`.

EXAMPLES:

```
sage: W = ZZ^2
sage: A1 = W.submodule([[1,0]])
sage: B1 = W.submodule([[2,0]])
sage: T1 = A1 / B1
sage: T1.construction()
(QuotientModuleFunctor,
 Free module of degree 2 and rank 1 over Integer Ring
 Echelon basis matrix:
```

(continues on next page)

(continued from previous page)

```
 [1 0])
```

```
>>> from sage.all import *
>>> W = ZZ**Integer(2)
>>> A1 = W.submodule([[Integer(1),Integer(0)]])
>>> B1 = W.submodule([[Integer(2),Integer(0)]])
>>> T1 = A1 / B1
>>> T1.construction()
(QuotientModuleFunctor,
 Free module of degree 2 and rank 1 over Integer Ring
 Echelon basis matrix:
 [1 0])
```

coordinate_vector(*x*, *reduce=False*)

Return coordinates of `x` with respect to the optimized representation of `self`.

INPUT:

- `x` – element of `self`
- `reduce` – (default: `False`) if `True`, reduce coefficients modulo invariants; this is ignored if the base ring is not `ZZ`

OUTPUT:

The coordinates as a vector. That is, the same type as `self.V()`, but in general with fewer entries.

EXAMPLES:

```
sage: V = span([[1/4,0,0], [3/4,4,2], [0,0,2]], ZZ)
sage: W = V.span([4*V.0 + 12*V.1])
sage: Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 0, 0)
sage: Q.coordinate_vector(-Q.0)
(-1, 0, 0)
sage: Q.coordinate_vector(-Q.0, reduce=True)
(3, 0, 0)
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(4),Integer(0),Integer(0)], [Integer(3)/
↪Integer(4),Integer(4),Integer(2)], [Integer(0),Integer(0),Integer(2)]], ZZ)
>>> W = V.span([Integer(4)*V.gen(0) + Integer(12)*V.gen(1)])
>>> Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 0, 0)
>>> Q.coordinate_vector(-Q.gen(0))
(-1, 0, 0)
>>> Q.coordinate_vector(-Q.gen(0), reduce=True)
(3, 0, 0)
```

If x is not in self, it is coerced in:

```
sage: Q.coordinate_vector(V.0)
(1, -3, 0)
sage: Q.coordinate_vector(Q(V.0))
(1, -3, 0)
```

```
>>> from sage.all import *
>>> Q.coordinate_vector(V.gen(0))
(1, -3, 0)
>>> Q.coordinate_vector(Q(V.gen(0)))
(1, -3, 0)
```

cover()

If this module was constructed as V/W, return the cover module V.

This is the same as `self.V()`.

EXAMPLES:

```
sage: V = span([[1/2,1,1], [3/2,2,1], [0,0,1]], ZZ)
sage: W = V.span([2*V.0 + 4*V.1, 9*V.0 + 12*V.1, 4*V.2])
sage: Q = V/W
sage: Q.V()
Free module of degree 3 and rank 3 over Integer Ring
Echelon basis matrix:
[1/2   0   0]
[  0   1   0]
[  0   0   1]
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)], [Integer(3)/
↪Integer(2),Integer(2),Integer(1)], [Integer(0),Integer(0),Integer(1)]], ZZ)
>>> W = V.span([Integer(2)*V.gen(0) + Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0) + Integer(12)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q = V/W
>>> Q.V()
Free module of degree 3 and rank 3 over Integer Ring
Echelon basis matrix:
[1/2   0   0]
[  0   1   0]
[  0   0   1]
```

gen(i)

Return the `i`-th generator of `self`.

INPUT:

- `i` – integer

EXAMPLES:

```
sage: V = span([[1/2,1,1], [3/2,2,1], [0,0,1]], ZZ)
sage: W = V.span([2*V.0 + 4*V.1, 9*V.0 + 12*V.1, 4*V.2])
sage: Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 12)
sage: Q.gen(0)
(1, 0)
sage: Q.gen(1)
(0, 1)
sage: Q.gen(2)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: Generator 2 not defined
sage: Q.gen(-1)
Traceback (most recent call last):
...
ValueError: Generator -1 not defined
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)], [Integer(3)/
↪Integer(2),Integer(2),Integer(1)], [Integer(0),Integer(0),Integer(1)]], ZZ)
>>> W = V.span([Integer(2)*V.gen(0) + Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0) + Integer(12)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 12)
>>> Q.gen(Integer(0))
(1, 0)
>>> Q.gen(Integer(1))
(0, 1)
>>> Q.gen(Integer(2))
Traceback (most recent call last):
...
ValueError: Generator 2 not defined
>>> Q.gen(-Integer(1))
Traceback (most recent call last):
...
ValueError: Generator -1 not defined
```

gens()

Return tuple of elements $g_0, ..., g_n$ of `self` such that the module generated by the g_i is isomorphic to the direct sum of R/e_iR, where e_i are the invariants of `self` and R is the base ring.

Note that these are not generally uniquely determined, and depending on how Smith normal form is implemented for the base ring, they may not even be deterministic.

This can safely be overridden in all derived classes.

EXAMPLES:

```
sage: V = span([[1/2,1,1], [3/2,2,1], [0,0,1]], ZZ)
sage: W = V.span([2*V.0 + 4*V.1, 9*V.0 + 12*V.1, 4*V.2])
sage: Q = V/W
sage: Q.gens()
((1, 0), (0, 1))
sage: Q.0
(1, 0)
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)], [Integer(3)/
↪Integer(2),Integer(2),Integer(1)], [Integer(0),Integer(0),Integer(1)]], ZZ)
>>> W = V.span([Integer(2)*V.gen(0) + Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0) + Integer(12)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q = V/W
>>> Q.gens()
```

(continues on next page)

(continued from previous page)

```
((1, 0), (0, 1))
>>> Q.gen(0)
(1, 0)
```

gens_to_smith()

Return the transformation matrix from the user to Smith form generators.

To go in the other direction, use *smith_to_gens()*.

OUTPUT: a matrix over the base ring

EXAMPLES:

```
sage: L2 = IntegralLattice(3 * matrix([[-2,0,0], [0,1,0], [0,0,-4]]))
sage: D = L2.discriminant_group().normal_form(); D                              #␣
↪needs sage.libs.pari sage.rings.padics
Finite quadratic module over Integer Ring with invariants (3, 6, 12)
Gram matrix of the quadratic form with values in Q/Z:
[1/2   0   0   0   0]
[  0 1/4   0   0   0]
[  0   0 1/3   0   0]
[  0   0   0 1/3   0]
[  0   0   0   0 2/3]
sage: D.gens_to_smith()                                                        #␣
↪needs sage.libs.pari sage.rings.padics
[0 3 0]
[0 0 3]
[0 4 0]
[1 2 0]
[0 0 4]
sage: T = D.gens_to_smith() * D.smith_to_gens(); T                             #␣
↪needs sage.libs.pari sage.rings.padics
[ 3  0  3  0  0]
[ 0 33  0  0  3]
[ 4  0  4  0  0]
[ 2  0  3  1  0]
[ 0 44  0  0  4]
```

```
>>> from sage.all import *
>>> L2 = IntegralLattice(Integer(3) * matrix([[-Integer(2),Integer(0),
↪Integer(0)], [Integer(0),Integer(1),Integer(0)], [Integer(0),Integer(0),-
↪Integer(4)]]))
>>> D = L2.discriminant_group().normal_form(); D                               #␣
↪needs sage.libs.pari sage.rings.padics
Finite quadratic module over Integer Ring with invariants (3, 6, 12)
Gram matrix of the quadratic form with values in Q/Z:
[1/2   0   0   0   0]
[  0 1/4   0   0   0]
[  0   0 1/3   0   0]
[  0   0   0 1/3   0]
[  0   0   0   0 2/3]
>>> D.gens_to_smith()                                                          #␣
↪needs sage.libs.pari sage.rings.padics
```

(continues on next page)

(continued from previous page)

```
[0 3 0]
[0 0 3]
[0 4 0]
[1 2 0]
[0 0 4]
>>> T = D.gens_to_smith() * D.smith_to_gens(); T                                #␣
↪needs sage.libs.pari sage.rings.padics
[ 3  0  3  0  0]
[ 0 33  0  0  3]
[ 4  0  4  0  0]
[ 2  0  3  1  0]
[ 0 44  0  0  4]
```

The matrix T now satisfies a certain congruence:

```
sage: for i in range(T.nrows()):                                                #␣
↪needs sage.libs.pari sage.rings.padics
....:     T[:,i] = T[:,i] % D.gens()[i].order()
sage: T                                                                         #␣
↪needs sage.libs.pari sage.rings.padics
[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 *
>>> for i in range(T.nrows()):                                                  #␣
↪needs sage.libs.pari sage.rings.padics
...     T[:,i] = T[:,i] % D.gens()[i].order()
>>> T                                                                           #␣
↪needs sage.libs.pari sage.rings.padics
[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]
```

gens_vector(*x*, *reduce=False*)

Return coordinates of `x` with respect to the generators.

INPUT:

- `x` – element of `self`
- `reduce` – (default: `False`) if `True`, reduce coefficients modulo invariants; this is ignored if the base ring is not **Z**

EXAMPLES:

We create a derived class and overwrite *gens()*:

```
sage: from sage.modules.fg_pid.fgp_module import FGP_Module_class
sage: W = ZZ^3
```

(continues on next page)

(continued from previous page)

```
sage: V = W.span(matrix.diagonal([1/6, 1/3, 1/12]))
sage: class FGP_with_gens(FGP_Module_class):
....:     def __init__(self, V, W, gens):
....:         FGP_Module_class.__init__(self, V, W)
....:         self._gens = tuple([self(g) for g in gens])
....:     def gens(self):
....:         return self._gens
sage: gens = [(1/2, 0, 0), (0, 0, 1/4), (1/3, 0, 0), (0, 1/3, 0), (0, 0, 2/3)]
sage: gens = [V(g) for g in gens]
sage: D = FGP_with_gens(V, W, gens)
sage: D.gens()
((0, 3, 0), (0, 0, 3), (0, 4, 0), (1, 2, 0), (0, 0, 8))
```

```
>>> from sage.all import *
>>> from sage.modules.fg_pid.fgp_module import FGP_Module_class
>>> W = ZZ**Integer(3)
>>> V = W.span(matrix.diagonal([Integer(1)/Integer(6), Integer(1)/Integer(3),␣
↪Integer(1)/Integer(12)]))
>>> class FGP_with_gens(FGP_Module_class):
...     def __init__(self, V, W, gens):
...         FGP_Module_class.__init__(self, V, W)
...         self._gens = tuple([self(g) for g in gens])
...     def gens(self):
...         return self._gens
>>> gens = [(Integer(1)/Integer(2), Integer(0), Integer(0)), (Integer(0),␣
↪Integer(0), Integer(1)/Integer(4)), (Integer(1)/Integer(3), Integer(0),␣
↪Integer(0)), (Integer(0), Integer(1)/Integer(3), Integer(0)), (Integer(0),␣
↪Integer(0), Integer(2)/Integer(3))]
>>> gens = [V(g) for g in gens]
>>> D = FGP_with_gens(V, W, gens)
>>> D.gens()
((0, 3, 0), (0, 0, 3), (0, 4, 0), (1, 2, 0), (0, 0, 8))
```

We create some element of `D`:

```
sage: x = D.linear_combination_of_smith_form_gens((1,2,3)); x
(1, 2, 3)
```

```
>>> from sage.all import *
>>> x = D.linear_combination_of_smith_form_gens((Integer(1),Integer(2),
↪Integer(3))); x
(1, 2, 3)
```

In our generators:

```
sage: v = D.gens_vector(x); v                                                #␣
↪needs sage.libs.pari
(2, 9, 3, 1, 33)
```

```
>>> from sage.all import *
>>> v = D.gens_vector(x); v                                                  #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.libs.pari
(2, 9, 3, 1, 33)
```

The output can be further reduced:

```
sage: D.gens_vector(x, reduce=True)                                          #␣
↪needs sage.libs.pari
(0, 1, 0, 1, 0)
```

```
>>> from sage.all import *
>>> D.gens_vector(x, reduce=True)                                            #␣
↪needs sage.libs.pari
(0, 1, 0, 1, 0)
```

Let us check:

```
sage: x == sum(v[i]*D.gen(i) for i in range(len(D.gens())))                  #␣
↪needs sage.libs.pari
True
```

```
>>> from sage.all import *
>>> x == sum(v[i]*D.gen(i) for i in range(len(D.gens())))                    #␣
↪needs sage.libs.pari
True
```

has_canonical_map_to(*A*)

Return `True` if `self` has a canonical map to `A`, relative to the given presentation of `A`.

This means that `A` is a finitely generated quotient module, `self.V()` is a submodule of `A.V()` and `self.W()` is a submodule of `A.W()`, i.e., that there is a natural map induced by inclusion of the V's. Note that we do *not* require that this natural map be injective; for this use *is_submodule()*.

EXAMPLES:

```
sage: V = span([[1/2,1,1], [3/2,2,1], [0,0,1]], ZZ)
sage: W = V.span([2*V.0 + 4*V.1, 9*V.0 + 12*V.1, 4*V.2])
sage: Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 12)
sage: A = Q.submodule((Q.0, Q.0 + 3*Q.1)); A
Finitely generated module V/W over Integer Ring with invariants (4, 4)
sage: A.has_canonical_map_to(Q)
True
sage: Q.has_canonical_map_to(A)
False
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)], [Integer(3)/
↪Integer(2),Integer(2),Integer(1)], [Integer(0),Integer(0),Integer(1)]], ZZ)
>>> W = V.span([Integer(2)*V.gen(0) + Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0) + Integer(12)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 12)
```

(continues on next page)

(continued from previous page)

```
>>> A = Q.submodule((Q.gen(0), Q.gen(0) + Integer(3)*Q.gen(1))); A
Finitely generated module V/W over Integer Ring with invariants (4, 4)
>>> A.has_canonical_map_to(Q)
True
>>> Q.has_canonical_map_to(A)
False
```

hom(*im_gens*, *codomain=None*, *check=True*)

Homomorphism defined by giving the images of `self.gens()` in some fixed finitely generated R-module.

Note

We do not assume that the generators given by `self.gens()` are the same as the Smith form generators, since this may not be true for a general derived class.

INPUT:

- `im_gens` – list of the images of `self.gens()` in some R-module

EXAMPLES:

```
sage: V = span([[1/2,1,1], [3/2,2,1], [0,0,1]], ZZ)
sage: W = V.span([2*V.0 + 4*V.1, 9*V.0 + 12*V.1, 4*V.2])
sage: Q = V/W
sage: phi = Q.hom([3*Q.1, Q.0])
sage: phi
Morphism from module over Integer Ring with invariants (4, 12)
           to module with invariants (4, 12)
  that sends the generators to [(0, 3), (1, 0)]
sage: phi(Q.0)
(0, 3)
sage: phi(Q.1)
(1, 0)
sage: Q.0 == phi(Q.1)
True
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)], [Integer(3)/
↪Integer(2),Integer(2),Integer(1)], [Integer(0),Integer(0),Integer(1)]], ZZ)
>>> W = V.span([Integer(2)*V.gen(0) + Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0) + Integer(12)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q = V/W
>>> phi = Q.hom([Integer(3)*Q.gen(1), Q.gen(0)])
>>> phi
Morphism from module over Integer Ring with invariants (4, 12)
           to module with invariants (4, 12)
  that sends the generators to [(0, 3), (1, 0)]
>>> phi(Q.gen(0))
(0, 3)
>>> phi(Q.gen(1))
(1, 0)
```

(continues on next page)

(continued from previous page)

```
>>> Q.gen(0) == phi(Q.gen(1))
True
```

This example illustrates creating a morphism to a free module. The free module is turned into an FGP module (i.e., quotient V/W with $W = 0$), and the morphism is constructed:

```
sage: V = span([[1/2,0,0], [3/2,2,1], [0,0,1]], ZZ)
sage: W = V.span([2*V.0 + 4*V.1])
sage: Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (2, 0, 0)
sage: phi = Q.hom([0, V.0, V.1]); phi
Morphism from module over Integer Ring with invariants (2, 0, 0)
           to module with invariants (0, 0, 0)
  that sends the generators to [(0, 0, 0), (1, 0, 0), (0, 1, 0)]
sage: phi.domain()
Finitely generated module V/W over Integer Ring with invariants (2, 0, 0)
sage: phi.codomain()
Finitely generated module V/W over Integer Ring with invariants (0, 0, 0)
sage: phi(Q.0)
(0, 0, 0)
sage: phi(Q.1)
(1, 0, 0)
sage: phi(Q.2) == V.1
True
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(0),Integer(0)], [Integer(3)/
↪Integer(2),Integer(2),Integer(1)], [Integer(0),Integer(0),Integer(1)]], ZZ)
>>> W = V.span([Integer(2)*V.gen(0) + Integer(4)*V.gen(1)])
>>> Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (2, 0, 0)
>>> phi = Q.hom([Integer(0), V.gen(0), V.gen(1)]); phi
Morphism from module over Integer Ring with invariants (2, 0, 0)
           to module with invariants (0, 0, 0)
  that sends the generators to [(0, 0, 0), (1, 0, 0), (0, 1, 0)]
>>> phi.domain()
Finitely generated module V/W over Integer Ring with invariants (2, 0, 0)
>>> phi.codomain()
Finitely generated module V/W over Integer Ring with invariants (0, 0, 0)
>>> phi(Q.gen(0))
(0, 0, 0)
>>> phi(Q.gen(1))
(1, 0, 0)
>>> phi(Q.gen(2)) == V.gen(1)
True
```

Constructing two zero maps from the zero module:

```
sage: A = (ZZ^2)/(ZZ^2); A
Finitely generated module V/W over Integer Ring with invariants ()
sage: A.hom([])
Morphism from module over Integer Ring with invariants ()
```

(continues on next page)

(continued from previous page)

```
          to module with invariants ()
  that sends the generators to []
sage: A.hom([]).codomain() is A
True
sage: B = (ZZ^3)/(ZZ^3)
sage: phi = A.hom([], codomain=B); phi
Morphism from module over Integer Ring with invariants ()
          to module with invariants ()
  that sends the generators to []
sage: phi(A(0))
()
sage: phi(A(0)) == B(0)
True
```

```
>>> from sage.all import *
>>> A = (ZZ**Integer(2))/(ZZ**Integer(2)); A
Finitely generated module V/W over Integer Ring with invariants ()
>>> A.hom([])
Morphism from module over Integer Ring with invariants ()
          to module with invariants ()
  that sends the generators to []
>>> A.hom([]).codomain() is A
True
>>> B = (ZZ**Integer(3))/(ZZ**Integer(3))
>>> phi = A.hom([], codomain=B); phi
Morphism from module over Integer Ring with invariants ()
          to module with invariants ()
  that sends the generators to []
>>> phi(A(Integer(0)))
()
>>> phi(A(Integer(0))) == B(Integer(0))
True
```

A degenerate case:

```
sage: A = (ZZ^2)/(ZZ^2)
sage: phi = A.hom([]); phi
Morphism from module over Integer Ring with invariants ()
          to module with invariants ()
  that sends the generators to []
sage: phi(A(0))
()
```

```
>>> from sage.all import *
>>> A = (ZZ**Integer(2))/(ZZ**Integer(2))
>>> phi = A.hom([]); phi
Morphism from module over Integer Ring with invariants ()
          to module with invariants ()
  that sends the generators to []
>>> phi(A(Integer(0)))
()
```

The code checks that the morphism is valid. In the example below we try to send a generator of order 2 to an element of order 14:

```
sage: V = span([[1/14,3/14], [0,1/2]], ZZ); W = ZZ^2
sage: Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (2, 14)
sage: Q.linear_combination_of_smith_form_gens([1,11]).additive_order()
14
sage: f = Q.hom([Q.linear_combination_of_smith_form_gens([1,11]),
....:            Q.linear_combination_of_smith_form_gens([1,3])]); f
Traceback (most recent call last):
...
ValueError: phi must send optimized submodule of M.W() into N.W()
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(14),Integer(3)/Integer(14)], [Integer(0),
↪Integer(1)/Integer(2)]], ZZ); W = ZZ**Integer(2)
>>> Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (2, 14)
>>> Q.linear_combination_of_smith_form_gens([Integer(1),Integer(11)]).
↪additive_order()
14
>>> f = Q.hom([Q.linear_combination_of_smith_form_gens([Integer(1),
↪Integer(11)]),
...            Q.linear_combination_of_smith_form_gens([Integer(1),
↪Integer(3)])]); f
Traceback (most recent call last):
...
ValueError: phi must send optimized submodule of M.W() into N.W()
```

invariants(*include_ones=False*)

Return the diagonal entries of the Smith form of the relative matrix that defines `self` (see `_relative_matrix()`) padded with zeros, excluding 1s by default. Thus if `v` is the list of integers returned, then `self` is abstractly isomorphic to the product of cyclic groups $\mathbf{Z}/n\mathbf{Z}$ where n is in `v`.

INPUT:

- `include_ones` – boolean (default: `False`); if `True`, also include 1s in the output list

EXAMPLES:

```
sage: V = span([[1/2,1,1], [3/2,2,1], [0,0,1]], ZZ)
sage: W = V.span([2*V.0 + 4*V.1, 9*V.0 + 12*V.1, 4*V.2])
sage: Q = V/W
sage: Q.invariants()
(4, 12)
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)], [Integer(3)/
↪Integer(2),Integer(2),Integer(1)], [Integer(0),Integer(0),Integer(1)]], ZZ)
>>> W = V.span([Integer(2)*V.gen(0) + Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0) + Integer(12)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q = V/W
>>> Q.invariants()
(4, 12)
```

An example with 1 and 0 rows:

```
sage: V = ZZ^3; W = V.span([[1,2,0], [0,1,0], [0,2,0]]); Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (0)
sage: Q.invariants()
(0,)
sage: Q.invariants(include_ones=True)
(1, 1, 0)
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(3); W = V.span([[Integer(1),Integer(2),Integer(0)],␣
↪[Integer(0),Integer(1),Integer(0)], [Integer(0),Integer(2),Integer(0)]]); Q␣
↪= V/W; Q
Finitely generated module V/W over Integer Ring with invariants (0)
>>> Q.invariants()
(0,)
>>> Q.invariants(include_ones=True)
(1, 1, 0)
```

is_finite()

Return `True` if `self` is finite and `False` otherwise.

EXAMPLES:

```
sage: V = span([[1/2,0,0], [3/2,2,1], [0,0,1]], ZZ)
sage: W = V.span([V.0 + 2*V.1, 9*V.0 + 2*V.1, 4*V.2])
sage: Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 16)
sage: Q.is_finite()
True
sage: Q = V / V.zero_submodule(); Q
Finitely generated module V/W over Integer Ring with invariants (0, 0, 0)
sage: Q.is_finite()
False
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(0),Integer(0)], [Integer(3)/
↪Integer(2),Integer(2),Integer(1)], [Integer(0),Integer(0),Integer(1)]], ZZ)
>>> W = V.span([V.gen(0) + Integer(2)*V.gen(1), Integer(9)*V.gen(0) +␣
↪Integer(2)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 16)
>>> Q.is_finite()
True
>>> Q = V / V.zero_submodule(); Q
Finitely generated module V/W over Integer Ring with invariants (0, 0, 0)
>>> Q.is_finite()
False
```

is_submodule(*A*)

Return `True` if `self` is a submodule of `A`.

More precisely, this returns `True` if `self.V()` is a submodule of `A.V()`, with `self.W()` equal to `A.W()`.

Compare *has_canonical_map_to()*.

EXAMPLES:

```
sage: V = ZZ^2; W = V.span([[1,2]]); W2 = W.scale(2)
sage: A = V/W; B = W/W2
sage: B.is_submodule(A)
False
sage: A = V/W2; B = W/W2
sage: B.is_submodule(A)
True
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(2); W = V.span([[Integer(1),Integer(2)]]); W2 = W.
↪scale(Integer(2))
>>> A = V/W; B = W/W2
>>> B.is_submodule(A)
False
>>> A = V/W2; B = W/W2
>>> B.is_submodule(A)
True
```

This example illustrates that this command works in a subtle cases.:

```
sage: A = ZZ^1
sage: Q3 = A / A.span([[3]])
sage: Q6 = A / A.span([[6]])
sage: Q6.is_submodule(Q3)
False
sage: Q6.has_canonical_map_to(Q3)
True
sage: Q = A.span([[2]]) / A.span([[6]])
sage: Q.is_submodule(Q6)
True
```

```
>>> from sage.all import *
>>> A = ZZ**Integer(1)
>>> Q3 = A / A.span([[Integer(3)]])
>>> Q6 = A / A.span([[Integer(6)]])
>>> Q6.is_submodule(Q3)
False
>>> Q6.has_canonical_map_to(Q3)
True
>>> Q = A.span([[Integer(2)]]) / A.span([[Integer(6)]])
>>> Q.is_submodule(Q6)
True
```

linear_combination_of_smith_form_gens(*x*)

Compute a linear combination of the optimised generators of this module as returned by *smith_form_gens()*.

EXAMPLES:

```
sage: X = ZZ**2 / span([[3,0], [0,2]], ZZ)
sage: X.linear_combination_of_smith_form_gens([1])
(1)
```

```
>>> from sage.all import *
>>> X = ZZ**Integer(2) / span([[Integer(3),Integer(0)], [Integer(0),
↪Integer(2)]], ZZ)
>>> X.linear_combination_of_smith_form_gens([Integer(1)])
(1)
```

list()

Return a list of the elements of `self`.

EXAMPLES:

```
sage: V = ZZ^2; W = V.span([[1,2],[3,4]])
sage: list(V/W)
[(0), (1)]
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(2); W = V.span([[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]])
>>> list(V/W)
[(0), (1)]
```

ngens()

Return the number of generators of `self`.

(Note for developers: This is just the length of *gens()*, rather than of the minimal set of generators as returned by *smith_form_gens()*; these are the same in the *FGP_Module_class*, but not necessarily in derived classes.)

EXAMPLES:

```
sage: A = (ZZ**2) / span([[4,0], [0,3]], ZZ)
sage: A.ngens()
1
```

```
>>> from sage.all import *
>>> A = (ZZ**Integer(2)) / span([[Integer(4),Integer(0)], [Integer(0),
↪Integer(3)]], ZZ)
>>> A.ngens()
1
```

This works (but please do not do it in production code!)

```
sage: A.gens = lambda: [1,2,"Barcelona!"]
sage: A.ngens()
3
```

```
>>> from sage.all import *
>>> A.gens = lambda: [Integer(1),Integer(2),"Barcelona!"]
>>> A.ngens()
3
```

optimized()

Return a module isomorphic to this one, but with V replaced by a submodule of V such that the generators of `self` all lift trivially to generators of V. Replace W by the intersection of V and W. This has the advantage

that V has small dimension and any homomorphism from `self` trivially extends to a homomorphism from V.

OUTPUT:

- `Q` – an optimized quotient V_0/W_0 with V_0 a submodule of V such that $\phi : V_0/W_0 \to V/W$ is an isomorphism
- `Z` – matrix such that if x is in `self.V()` and `c` gives the coordinates of x in terms of the basis for `self.V()`, then `c*Z` is in V_0 and `c*Z` maps to x via ϕ above.

EXAMPLES:

```
sage: V = span([[1/2,1,1], [3/2,2,1], [0,0,1]], ZZ)
sage: W = V.span([2*V.0 + 4*V.1, 9*V.0 + 12*V.1, 4*V.2])
sage: Q = V/W
sage: O, X = Q.optimized(); O
Finitely generated module V/W over Integer Ring with invariants (4, 12)
sage: O.V()
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[ 0  3  1]
[ 0 -1  0]
sage: O.W()
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[ 0 12  0]
[ 0  0  4]
sage: X          # random
[0 4 0]
[0 1 0]
[0 0 1]
sage: OV = O.V()
sage: Q(OV([0,-8,0])) == V.0
True
sage: Q(OV([0,1,0])) == V.1
True
sage: Q(OV([0,0,1])) == V.2
True
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)], [Integer(3)/
↪Integer(2),Integer(2),Integer(1)], [Integer(0),Integer(0),Integer(1)]], ZZ)
>>> W = V.span([Integer(2)*V.gen(0) + Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0) + Integer(12)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q = V/W
>>> O, X = Q.optimized(); O
Finitely generated module V/W over Integer Ring with invariants (4, 12)
>>> O.V()
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[ 0  3  1]
[ 0 -1  0]
>>> O.W()
Free module of degree 3 and rank 2 over Integer Ring
```

(continues on next page)

(continued from previous page)

```
Echelon basis matrix:
[ 0 12  0]
[ 0  0  4]
>>> X          # random
[0 4 0]
[0 1 0]
[0 0 1]
>>> OV = O.V()
>>> Q(OV([Integer(0),-Integer(8),Integer(0)])) == V.gen(0)
True
>>> Q(OV([Integer(0),Integer(1),Integer(0)])) == V.gen(1)
True
>>> Q(OV([Integer(0),Integer(0),Integer(1)])) == V.gen(2)
True
```

quotient_map()

Given this quotient space $Q = V/W$, return the natural quotient map from V to Q.

EXAMPLES:

```
sage: A = (ZZ**2) / span([[4,0],[0,3]], ZZ)
sage: A.quotient_map()
Coercion map:
  From: Ambient free module of rank 2 over the principal ideal domain Integer␣
↪Ring
  To:   Finitely generated module V/W over Integer Ring with invariants (12)
```

```
>>> from sage.all import *
>>> A = (ZZ**Integer(2)) / span([[Integer(4),Integer(0)],[Integer(0),
↪Integer(3)]], ZZ)
>>> A.quotient_map()
Coercion map:
  From: Ambient free module of rank 2 over the principal ideal domain Integer␣
↪Ring
  To:   Finitely generated module V/W over Integer Ring with invariants (12)
```

random_element(**args*, ***kwds*)

Create a random element of `self` = V/W, by creating a random element of V and reducing it modulo W.

All arguments are passed on to the method *random_element()* of V.

EXAMPLES:

```
sage: V = span([[1/2,1,1], [3/2,2,1], [0,0,1]], ZZ)
sage: W = V.span([2*V.0 + 4*V.1, 9*V.0 + 12*V.1, 4*V.2])
sage: Q = V/W
sage: Q.random_element().parent() is Q
True
sage: Q.cardinality()
48
sage: S = set()
sage: while len(S) < 48:
....:     S.add(Q.random_element())
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)], [Integer(3)/
↪Integer(2),Integer(2),Integer(1)], [Integer(0),Integer(0),Integer(1)]], ZZ)
>>> W = V.span([Integer(2)*V.gen(0) + Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0) + Integer(12)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q = V/W
>>> Q.random_element().parent() is Q
True
>>> Q.cardinality()
48
>>> S = set()
>>> while len(S) < Integer(48):
...     S.add(Q.random_element())
```

relations()

If `self` was constructed as V/W, return the relations module W.

This is the same as `self.W()`.

EXAMPLES:

```
sage: V = span([[1/2,1,1], [3/2,2,1], [0,0,1]], ZZ)
sage: W = V.span([2*V.0 + 4*V.1, 9*V.0 + 12*V.1, 4*V.2])
sage: Q = V / W
sage: Q.relations()
Free module of degree 3 and rank 3 over Integer Ring
Echelon basis matrix:
[1/2   8   0]
[  0  12   0]
[  0   0   4]
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)], [Integer(3)/
↪Integer(2),Integer(2),Integer(1)], [Integer(0),Integer(0),Integer(1)]], ZZ)
>>> W = V.span([Integer(2)*V.gen(0) + Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0) + Integer(12)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q = V / W
>>> Q.relations()
Free module of degree 3 and rank 3 over Integer Ring
Echelon basis matrix:
[1/2   8   0]
[  0  12   0]
[  0   0   4]
```

smith_form_gen(i)

Return the `i`-th generator of `self`.

This is a separate method so we can freely override *gen()* in derived classes.

INPUT:

- `i` – integer

EXAMPLES:

```
sage: V = span([[1/2,1,1], [3/2,2,1], [0,0,1]], ZZ)
sage: W = V.span([2*V.0 + 4*V.1, 9*V.0 + 12*V.1, 4*V.2])
sage: Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 12)
sage: Q.smith_form_gen(0)
(1, 0)
sage: Q.smith_form_gen(1)
(0, 1)
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)], [Integer(3)/
↪Integer(2),Integer(2),Integer(1)], [Integer(0),Integer(0),Integer(1)]], ZZ)
>>> W = V.span([Integer(2)*V.gen(0) + Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0) + Integer(12)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 12)
>>> Q.smith_form_gen(Integer(0))
(1, 0)
>>> Q.smith_form_gen(Integer(1))
(0, 1)
```

smith_form_gens()

Return a set of generators for `self` which are in Smith normal form.

EXAMPLES:

```
sage: V = span([[1/2,1,1], [3/2,2,1], [0,0,1]], ZZ)
sage: W = V.span([2*V.0 + 4*V.1, 9*V.0 + 12*V.1, 4*V.2])
sage: Q = V/W
sage: Q.smith_form_gens()
((1, 0), (0, 1))
sage: [x.lift() for x in Q.smith_form_gens()]
[(0, 3, 1), (0, -1, 0)]
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)], [Integer(3)/
↪Integer(2),Integer(2),Integer(1)], [Integer(0),Integer(0),Integer(1)]], ZZ)
>>> W = V.span([Integer(2)*V.gen(0) + Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0) + Integer(12)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q = V/W
>>> Q.smith_form_gens()
((1, 0), (0, 1))
>>> [x.lift() for x in Q.smith_form_gens()]
[(0, 3, 1), (0, -1, 0)]
```

smith_to_gens()

Return the transformation matrix from Smith form to user generators.

To go in the other direction, use *gens_to_smith()*.

OUTPUT: a matrix over the base ring

EXAMPLES:

```
sage: L2 = IntegralLattice(3 * matrix([[-2,0,0], [0,1,0], [0,0,-4]]))
sage: D = L2.discriminant_group().normal_form(); D                              #␣
↪needs sage.libs.pari sage.rings.padics
Finite quadratic module over Integer Ring with invariants (3, 6, 12)
Gram matrix of the quadratic form with values in Q/Z:
[1/2   0   0   0   0]
[  0 1/4   0   0   0]
[  0   0 1/3   0   0]
[  0   0   0 1/3   0]
[  0   0   0   0 2/3]
sage: D.smith_to_gens()                                                        #␣
↪needs sage.libs.pari sage.rings.padics
[ 0  0  1  1  0]
[ 1  0  1  0  0]
[ 0 11  0  0  1]
sage: T = D.smith_to_gens() * D.gens_to_smith(); T                             #␣
↪needs sage.libs.pari sage.rings.padics
[ 1  6  0]
[ 0  7  0]
[ 0  0 37]
```

```
>>> from sage.all import *
>>> L2 = IntegralLattice(Integer(3) * matrix([[-Integer(2),Integer(0),
↪Integer(0)], [Integer(0),Integer(1),Integer(0)], [Integer(0),Integer(0),-
↪Integer(4)]]))
>>> D = L2.discriminant_group().normal_form(); D                               #␣
↪needs sage.libs.pari sage.rings.padics
Finite quadratic module over Integer Ring with invariants (3, 6, 12)
Gram matrix of the quadratic form with values in Q/Z:
[1/2   0   0   0   0]
[  0 1/4   0   0   0]
[  0   0 1/3   0   0]
[  0   0   0 1/3   0]
[  0   0   0   0 2/3]
>>> D.smith_to_gens()                                                          #␣
↪needs sage.libs.pari sage.rings.padics
[ 0  0  1  1  0]
[ 1  0  1  0  0]
[ 0 11  0  0  1]
>>> T = D.smith_to_gens() * D.gens_to_smith(); T                               #␣
↪needs sage.libs.pari sage.rings.padics
[ 1  6  0]
[ 0  7  0]
[ 0  0 37]
```

This matrix satisfies the congruence:

```
sage: for i in range(T.ncols()):                                               #␣
↪needs sage.libs.pari sage.rings.padics
....:     T[:, i] = T[:, i] % D.smith_form_gens()[i].order()
sage: T                                                                        #␣
↪needs sage.libs.pari sage.rings.padics
[1 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 1 0]
[0 0 1]
```

```
>>> from sage.all import *
>>> for i in range(T.ncols()):                                                #␣
↪needs sage.libs.pari sage.rings.padics
...     T[:, i] = T[:, i] % D.smith_form_gens()[i].order()
>>> T                                                                         #␣
↪needs sage.libs.pari sage.rings.padics
[1 0 0]
[0 1 0]
[0 0 1]
```

We create some element of our FGP module:

```
sage: x = D.linear_combination_of_smith_form_gens((1,2,3)); x                #␣
↪needs sage.libs.pari sage.rings.padics
(1, 2, 3)
```

```
>>> from sage.all import *
>>> x = D.linear_combination_of_smith_form_gens((Integer(1),Integer(2),
↪Integer(3))); x               # needs sage.libs.pari sage.rings.padics
(1, 2, 3)
```

and want to know some (it is not unique) linear combination of the user defined generators that is `x`:

```
sage: x.vector() * D.smith_to_gens()                                         #␣
↪needs sage.libs.pari sage.rings.padics
(2, 33, 3, 1, 3)
```

```
>>> from sage.all import *
>>> x.vector() * D.smith_to_gens()                                           #␣
↪needs sage.libs.pari sage.rings.padics
(2, 33, 3, 1, 3)
```

submodule(*x*)

Return the submodule defined by `x`.

INPUT:

- `x` – list, tuple, or FGP module

EXAMPLES:

```
sage: V = span([[1/2,1,1], [3/2,2,1], [0,0,1]], ZZ)
sage: W = V.span([2*V.0 + 4*V.1, 9*V.0 + 12*V.1, 4*V.2])
sage: Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 12)
sage: Q.gens()
((1, 0), (0, 1))
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)], [Integer(3)/
```

(continues on next page)

(continued from previous page)

```
↪Integer(2),Integer(2),Integer(1)], [Integer(0),Integer(0),Integer(1)]], ZZ)
>>> W = V.span([Integer(2)*V.gen(0) + Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0) + Integer(12)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 12)
>>> Q.gens()
((1, 0), (0, 1))
```

We create submodules generated by a list or tuple of elements:

```
sage: Q.submodule([Q.0])
Finitely generated module V/W over Integer Ring with invariants (4)
sage: Q.submodule([Q.1])
Finitely generated module V/W over Integer Ring with invariants (12)
sage: Q.submodule((Q.0, Q.0 + 3*Q.1))
Finitely generated module V/W over Integer Ring with invariants (4, 4)
```

```
>>> from sage.all import *
>>> Q.submodule([Q.gen(0)])
Finitely generated module V/W over Integer Ring with invariants (4)
>>> Q.submodule([Q.gen(1)])
Finitely generated module V/W over Integer Ring with invariants (12)
>>> Q.submodule((Q.gen(0), Q.gen(0) + Integer(3)*Q.gen(1)))
Finitely generated module V/W over Integer Ring with invariants (4, 4)
```

A submodule defined by a submodule:

```
sage: A = Q.submodule((Q.0, Q.0 + 3*Q.1)); A
Finitely generated module V/W over Integer Ring with invariants (4, 4)
sage: Q.submodule(A)
Finitely generated module V/W over Integer Ring with invariants (4, 4)
```

```
>>> from sage.all import *
>>> A = Q.submodule((Q.gen(0), Q.gen(0) + Integer(3)*Q.gen(1))); A
Finitely generated module V/W over Integer Ring with invariants (4, 4)
>>> Q.submodule(A)
Finitely generated module V/W over Integer Ring with invariants (4, 4)
```

Inclusion is checked:

```
sage: A.submodule(Q)
Traceback (most recent call last):
...
ValueError: x.V() must be contained in self's V.
```

```
>>> from sage.all import *
>>> A.submodule(Q)
Traceback (most recent call last):
...
ValueError: x.V() must be contained in self's V.
```

sage.modules.fg_pid.fgp_module.**is_FGP_Module**(*x*)

Return `True` if x is an FGP module, i.e., a finitely generated module over a PID represented as a quotient of finitely generated free modules over a PID.

EXAMPLES:

```
sage: V = span([[1/2,1,1],[3/2,2,1],[0,0,1]],ZZ)
sage: W = V.span([2*V.0 + 4*V.1, 9*V.0 + 12*V.1, 4*V.2]); Q = V/W
sage: sage.modules.fg_pid.fgp_module.is_FGP_Module(V)
doctest:warning...
DeprecationWarning: the function is_FGP_Module is deprecated;
use 'isinstance(..., FGP_Module_class)' instead
See https://github.com/sagemath/sage/issues/37924 for details.
False
sage: sage.modules.fg_pid.fgp_module.is_FGP_Module(Q)
True
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)],[Integer(3)/
↪Integer(2),Integer(2),Integer(1)],[Integer(0),Integer(0),Integer(1)]],ZZ)
>>> W = V.span([Integer(2)*V.gen(0) + Integer(4)*V.gen(1), Integer(9)*V.gen(0) +␣
↪Integer(12)*V.gen(1), Integer(4)*V.gen(2)]); Q = V/W
>>> sage.modules.fg_pid.fgp_module.is_FGP_Module(V)
doctest:warning...
DeprecationWarning: the function is_FGP_Module is deprecated;
use 'isinstance(..., FGP_Module_class)' instead
See https://github.com/sagemath/sage/issues/37924 for details.
False
>>> sage.modules.fg_pid.fgp_module.is_FGP_Module(Q)
True
```

`sage.modules.fg_pid.fgp_module.`**`random_fgp_module`**(*n*, *R=Integer Ring*, *finite=False*)

Return a random FGP module inside a rank n free module over R.

INPUT:

- `n` – nonnegative integer
- `R` – base ring (default: `ZZ`)
- `finite` – boolean (default: `True`); if `True`, make the random module finite

EXAMPLES:

```
sage: import sage.modules.fg_pid.fgp_module as fgp
sage: fgp.random_fgp_module(4)
Finitely generated module V/W over Integer Ring with invariants (...)
```

```
>>> from sage.all import *
>>> import sage.modules.fg_pid.fgp_module as fgp
>>> fgp.random_fgp_module(Integer(4))
Finitely generated module V/W over Integer Ring with invariants (...)
```

In most cases the cardinality is small or infinite:

```
sage: for g in (1, 2, 3, +Infinity):
....:     while fgp.random_fgp_module(4).cardinality() != 1:
....:         pass
```

```
>>> from sage.all import *
>>> for g in (Integer(1), Integer(2), Integer(3), +Infinity):
...     while fgp.random_fgp_module(Integer(4)).cardinality() != Integer(1):
...         pass
```

One can force a finite module:

```
sage: fgp.random_fgp_module(4, finite=True).is_finite()
True
```

```
>>> from sage.all import *
>>> fgp.random_fgp_module(Integer(4), finite=True).is_finite()
True
```

Larger finite modules appear:

```
sage: while fgp.random_fgp_module(4, finite=True).cardinality() < 100:
....:     pass
```

```
>>> from sage.all import *
>>> while fgp.random_fgp_module(Integer(4), finite=True).cardinality() <␣
↪Integer(100):
...     pass
```

sage.modules.fg_pid.fgp_module.**random_fgp_morphism_0**(**args*, ***kwds*)

Construct a random fgp module using *random_fgp_module()*, then construct a random morphism that sends each generator to a random multiple of itself.

Inputs are the same as to *random_fgp_module()*.

EXAMPLES:

```
sage: import sage.modules.fg_pid.fgp_module as fgp
sage: mor = fgp.random_fgp_morphism_0(4)
sage: mor.domain() == mor.codomain()
True
sage: isinstance(mor.domain(), fgp.FGP_Module_class)
True
```

```
>>> from sage.all import *
>>> import sage.modules.fg_pid.fgp_module as fgp
>>> mor = fgp.random_fgp_morphism_0(Integer(4))
>>> mor.domain() == mor.codomain()
True
>>> isinstance(mor.domain(), fgp.FGP_Module_class)
True
```

Each generator is sent to a random multiple of itself:

```
sage: gens = mor.domain().gens()
sage: im_gens = mor.im_gens()
sage: all(im_gens[i] == sum(im_gens[i])*gens[i] for i in range(len(gens)))
True
```

```
>>> from sage.all import *
>>> gens = mor.domain().gens()
>>> im_gens = mor.im_gens()
>>> all(im_gens[i] == sum(im_gens[i])*gens[i] for i in range(len(gens)))
True
```

4.2 Elements of finitely generated modules over a PID

AUTHOR:

- William Stein, 2009

class sage.modules.fg_pid.fgp_element.**FGP_Element**(*parent*, *x*, *check=True*)

Bases: ModuleElement

An element of a finitely generated module over a PID.

INPUT:

- parent – parent module M
- x – element of M.V()

EXAMPLES:

```
sage: V = span([[1/2,1,1],[3/2,2,1],[0,0,1]],ZZ); W = V.span([2*V.0+4*V.1, 9*V.
↪0+12*V.1, 4*V.2])
sage: Q = V/W
sage: x = Q(V.0-V.1); x #indirect doctest
(0, 9)
sage: isinstance(x, sage.modules.fg_pid.fgp_element.FGP_Element)
True
sage: type(x)
<class 'sage.modules.fg_pid.fgp_module.FGP_Module_class_with_category.element_
↪class'>
sage: x is Q(x)
True
sage: x.parent() is Q
True
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)],[Integer(3)/
↪Integer(2),Integer(2),Integer(1)],[Integer(0),Integer(0),Integer(1)]],ZZ); W =␣
↪V.span([Integer(2)*V.gen(0)+Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0)+Integer(12)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q = V/W
>>> x = Q(V.gen(0)-V.gen(1)); x #indirect doctest
(0, 9)
>>> isinstance(x, sage.modules.fg_pid.fgp_element.FGP_Element)
```

(continues on next page)

(continued from previous page)

```
True
>>> type(x)
<class 'sage.modules.fg_pid.fgp_module.FGP_Module_class_with_category.element_
↪class'>
>>> x is Q(x)
True
>>> x.parent() is Q
True
```

additive_order()

Return the additive order of this element.

EXAMPLES:

```
sage: V = span([[1/2,1,1],[3/2,2,1],[0,0,1]],ZZ); W = V.span([2*V.0+4*V.1,␣
↪9*V.0+12*V.1, 4*V.2])
sage: Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 12)
sage: Q.0.additive_order()
4
sage: Q.1.additive_order()
12
sage: (Q.0+Q.1).additive_order()
12
sage: V = span([[1/2,1,1],[3/2,2,1],[0,0,1]],ZZ); W = V.span([2*V.0+4*V.1,␣
↪9*V.0+12*V.1])
sage: Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (12, 0)
sage: Q.0.additive_order()
12
sage: type(Q.0.additive_order())
<class 'sage.rings.integer.Integer'>
sage: Q.1.additive_order()
+Infinity
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)],[Integer(3)/
↪Integer(2),Integer(2),Integer(1)],[Integer(0),Integer(0),Integer(1)]],ZZ);␣
↪W = V.span([Integer(2)*V.gen(0)+Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0)+Integer(12)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 12)
>>> Q.gen(0).additive_order()
4
>>> Q.gen(1).additive_order()
12
>>> (Q.gen(0)+Q.gen(1)).additive_order()
12
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)],[Integer(3)/
↪Integer(2),Integer(2),Integer(1)],[Integer(0),Integer(0),Integer(1)]],ZZ);␣
↪W = V.span([Integer(2)*V.gen(0)+Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0)+Integer(12)*V.gen(1)])
```

(continues on next page)

(continued from previous page)

```
>>> Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (12, 0)
>>> Q.gen(0).additive_order()
12
>>> type(Q.gen(0).additive_order())
<class 'sage.rings.integer.Integer'>
>>> Q.gen(1).additive_order()
+Infinity
```

lift()

Lift `self` to an element of V, where the parent of `self` is the quotient module V/W.

EXAMPLES:

```
sage: V = span([[1/2,0,0],[3/2,2,1],[0,0,1]],ZZ); W = V.span([2*V.0+4*V.1,␣
↪9*V.0+12*V.1, 4*V.2])
sage: Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 12)
sage: Q.0
(1, 0)
sage: Q.1
(0, 1)
sage: Q.0.lift()
(0, 6, 1)
sage: Q.1.lift()
(0, -2, 0)
sage: x = Q(V.0); x
(0, 8)
sage: x.lift()
(1/2, 0, 0)
sage: x == 8*Q.1
True
sage: x.lift().parent() == V
True
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(0),Integer(0)],[Integer(3)/
↪Integer(2),Integer(2),Integer(1)],[Integer(0),Integer(0),Integer(1)]],ZZ);␣
↪W = V.span([Integer(2)*V.gen(0)+Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0)+Integer(12)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 12)
>>> Q.gen(0)
(1, 0)
>>> Q.gen(1)
(0, 1)
>>> Q.gen(0).lift()
(0, 6, 1)
>>> Q.gen(1).lift()
(0, -2, 0)
>>> x = Q(V.gen(0)); x
(0, 8)
```

(continues on next page)

(continued from previous page)

```
>>> x.lift()
(1/2, 0, 0)
>>> x == Integer(8)*Q.gen(1)
True
>>> x.lift().parent() == V
True
```

A silly version of the integers modulo 100:

```
sage: A = (ZZ^1)/span([[100]], ZZ); A
Finitely generated module V/W over Integer Ring with invariants (100)
sage: x = A([5]); x
(5)
sage: v = x.lift(); v
(5)
sage: v.parent()
Ambient free module of rank 1 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> A = (ZZ**Integer(1))/span([[Integer(100)]], ZZ); A
Finitely generated module V/W over Integer Ring with invariants (100)
>>> x = A([Integer(5)]); x
(5)
>>> v = x.lift(); v
(5)
>>> v.parent()
Ambient free module of rank 1 over the principal ideal domain Integer Ring
```

vector()

EXAMPLES:

```
sage: V = span([[1/2,0,0],[3/2,2,1],[0,0,1]],ZZ); W = V.span([2*V.0+4*V.1,␣
↪9*V.0+12*V.1, 4*V.2])
sage: Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 12)
sage: x = Q.0 + 3*Q.1; x
(1, 3)
sage: x.vector()
(1, 3)
sage: tuple(x)
(1, 3)
sage: list(x)
[1, 3]
sage: x.vector().parent()
Ambient free module of rank 2 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(0),Integer(0)],[Integer(3)/
↪Integer(2),Integer(2),Integer(1)],[Integer(0),Integer(0),Integer(1)]],ZZ);␣
↪W = V.span([Integer(2)*V.gen(0)+Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0)+Integer(12)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q = V/W; Q
```

(continues on next page)

(continued from previous page)

```
Finitely generated module V/W over Integer Ring with invariants (4, 12)
>>> x = Q.gen(0) + Integer(3)*Q.gen(1); x
(1, 3)
>>> x.vector()
(1, 3)
>>> tuple(x)
(1, 3)
>>> list(x)
[1, 3]
>>> x.vector().parent()
Ambient free module of rank 2 over the principal ideal domain Integer Ring
```

4.3 Morphisms between finitely generated modules over a PID

AUTHOR:

- William Stein, 2009

sage.modules.fg_pid.fgp_morphism.**FGP_Homset**(*X*, *Y*)

EXAMPLES:

```
sage: V = span([[1/2,1,1],[3/2,2,1],[0,0,1]],ZZ); W = V.span([2*V.0+4*V.1, 9*V.
↪0+12*V.1, 4*V.2]); Q = V/W
sage: Q.Hom(Q)            # indirect doctest
Set of Morphisms from Finitely generated module V/W over Integer Ring with␣
↪invariants (4, 12) to Finitely generated module V/W over Integer Ring with␣
↪invariants (4, 12) in Category of modules over Integer Ring
sage: True # Q.Hom(Q) is Q.Hom(Q)
True
sage: type(Q.Hom(Q))
<class 'sage.modules.fg_pid.fgp_morphism.FGP_Homset_class_with_category'>
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)],[Integer(3)/
↪Integer(2),Integer(2),Integer(1)],[Integer(0),Integer(0),Integer(1)]],ZZ); W =␣
↪V.span([Integer(2)*V.gen(0)+Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0)+Integer(12)*V.gen(1), Integer(4)*V.gen(2)]); Q = V/W
>>> Q.Hom(Q)            # indirect doctest
Set of Morphisms from Finitely generated module V/W over Integer Ring with␣
↪invariants (4, 12) to Finitely generated module V/W over Integer Ring with␣
↪invariants (4, 12) in Category of modules over Integer Ring
>>> True # Q.Hom(Q) is Q.Hom(Q)
True
>>> type(Q.Hom(Q))
<class 'sage.modules.fg_pid.fgp_morphism.FGP_Homset_class_with_category'>
```

class sage.modules.fg_pid.fgp_morphism.**FGP_Homset_class**(*X*, *Y*, *category=None*)

Bases: Homset

Homsets of *FGP_Module*.

Element

alias of *FGP_Morphism*

class sage.modules.fg_pid.fgp_morphism.**FGP_Morphism**(*parent*, *phi*, *check=True*)

Bases: Morphism

A morphism between finitely generated modules over a PID.

EXAMPLES:

An endomorphism:

```
sage: V = span([[1/2,1,1],[3/2,2,1],[0,0,1]],ZZ); W = V.span([2*V.0+4*V.1, 9*V.
↪0+12*V.1, 4*V.2])
sage: Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 12)
sage: phi = Q.hom([Q.0+3*Q.1, -Q.1]); phi
Morphism from module over Integer Ring with invariants (4, 12) to module with␣
↪invariants (4, 12) that sends the generators to [(1, 3), (0, 11)]
sage: phi(Q.0) == Q.0 + 3*Q.1
True
sage: phi(Q.1) == -Q.1
True
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)],[Integer(3)/
↪Integer(2),Integer(2),Integer(1)],[Integer(0),Integer(0),Integer(1)]],ZZ); W =␣
↪V.span([Integer(2)*V.gen(0)+Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0)+Integer(12)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 12)
>>> phi = Q.hom([Q.gen(0)+Integer(3)*Q.gen(1), -Q.gen(1)]); phi
Morphism from module over Integer Ring with invariants (4, 12) to module with␣
↪invariants (4, 12) that sends the generators to [(1, 3), (0, 11)]
>>> phi(Q.gen(0)) == Q.gen(0) + Integer(3)*Q.gen(1)
True
>>> phi(Q.gen(1)) == -Q.gen(1)
True
```

A morphism between different modules V1/W1 —> V2/W2 in different ambient spaces:

```
sage: V1 = ZZ^2; W1 = V1.span([[1,2],[3,4]]); A1 = V1/W1
sage: V2 = span([[1/2,1,1],[3/2,2,1],[0,0,1]],ZZ); W2 = V2.span([2*V2.0+4*V2.1,␣
↪9*V2.0+12*V2.1, 4*V2.2]); A2=V2/W2
sage: A1
Finitely generated module V/W over Integer Ring with invariants (2)
sage: A2
Finitely generated module V/W over Integer Ring with invariants (4, 12)
sage: phi = A1.hom([2*A2.0])
sage: phi(A1.0)
(2, 0)
sage: 2*A2.0
(2, 0)
sage: phi(2*A1.0)
(0, 0)
```

```
>>> from sage.all import *
>>> V1 = ZZ**Integer(2); W1 = V1.span([[Integer(1),Integer(2)],[Integer(3),
↪Integer(4)]]); A1 = V1/W1
>>> V2 = span([[Integer(1)/Integer(2),Integer(1),Integer(1)],[Integer(3)/
↪Integer(2),Integer(2),Integer(1)],[Integer(0),Integer(0),Integer(1)]],ZZ); W2 =␣
↪V2.span([Integer(2)*V2.gen(0)+Integer(4)*V2.gen(1), Integer(9)*V2.
↪gen(0)+Integer(12)*V2.gen(1), Integer(4)*V2.gen(2)]); A2=V2/W2
>>> A1
Finitely generated module V/W over Integer Ring with invariants (2)
>>> A2
Finitely generated module V/W over Integer Ring with invariants (4, 12)
>>> phi = A1.hom([Integer(2)*A2.gen(0)])
>>> phi(A1.gen(0))
(2, 0)
>>> Integer(2)*A2.gen(0)
(2, 0)
>>> phi(Integer(2)*A1.gen(0))
(0, 0)
```

im_gens()

Return tuple of the images of the generators of the domain under this morphism.

EXAMPLES:

```
sage: V = span([[1/2,1,1],[3/2,2,1],[0,0,1]],ZZ); W = V.span([2*V.0+4*V.1,␣
↪9*V.0+12*V.1, 4*V.2]); Q = V/W
sage: phi = Q.hom([Q.0,Q.0 + 2*Q.1])
sage: phi.im_gens()
((1, 0), (1, 2))
sage: phi.im_gens() is phi.im_gens()
True
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)],[Integer(3)/
↪Integer(2),Integer(2),Integer(1)],[Integer(0),Integer(0),Integer(1)]],ZZ);␣
↪W = V.span([Integer(2)*V.gen(0)+Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0)+Integer(12)*V.gen(1), Integer(4)*V.gen(2)]); Q = V/W
>>> phi = Q.hom([Q.gen(0),Q.gen(0) + Integer(2)*Q.gen(1)])
>>> phi.im_gens()
((1, 0), (1, 2))
>>> phi.im_gens() is phi.im_gens()
True
```

image()

Compute the image of this homomorphism.

EXAMPLES:

```
sage: V = span([[1/2,1,1],[3/2,2,1],[0,0,1]],ZZ); W = V.span([2*V.0+4*V.1,␣
↪9*V.0+12*V.1, 4*V.2])
sage: Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 12)
sage: Q.hom([Q.0+3*Q.1, -Q.1]).image()
```

(continues on next page)

(continued from previous page)

```
Finitely generated module V/W over Integer Ring with invariants (4, 12)
sage: Q.hom([3*Q.1, Q.1]).image()
Finitely generated module V/W over Integer Ring with invariants (12)
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)],[Integer(3)/
↪Integer(2),Integer(2),Integer(1)],[Integer(0),Integer(0),Integer(1)]],ZZ);␣
↪W = V.span([Integer(2)*V.gen(0)+Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0)+Integer(12)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 12)
>>> Q.hom([Q.gen(0)+Integer(3)*Q.gen(1), -Q.gen(1)]).image()
Finitely generated module V/W over Integer Ring with invariants (4, 12)
>>> Q.hom([Integer(3)*Q.gen(1), Q.gen(1)]).image()
Finitely generated module V/W over Integer Ring with invariants (12)
```

inverse_image(*A*)

Given a submodule A of the codomain of this morphism, return the inverse image of A under this morphism.

EXAMPLES:

```
sage: V = span([[1/2,1,1],[3/2,2,1],[0,0,1]],ZZ); W = V.span([2*V.0+4*V.1,␣
↪9*V.0+12*V.1, 4*V.2]); Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 12)
sage: phi = Q.hom([0, Q.1])
sage: phi.inverse_image(Q.submodule([]))
Finitely generated module V/W over Integer Ring with invariants (4)
sage: phi.kernel()
Finitely generated module V/W over Integer Ring with invariants (4)
sage: phi.inverse_image(phi.codomain())
Finitely generated module V/W over Integer Ring with invariants (4, 12)

sage: phi.inverse_image(Q.submodule([Q.0]))
Finitely generated module V/W over Integer Ring with invariants (4)
sage: phi.inverse_image(Q.submodule([Q.1]))
Finitely generated module V/W over Integer Ring with invariants (4, 12)

sage: phi.inverse_image(ZZ^3)
Traceback (most recent call last):
...
TypeError: A must be a finitely generated quotient module
sage: phi.inverse_image(ZZ^3 / W.scale(2))
Traceback (most recent call last):
...
ValueError: A must be a submodule of the codomain
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)],[Integer(3)/
↪Integer(2),Integer(2),Integer(1)],[Integer(0),Integer(0),Integer(1)]],ZZ);␣
↪W = V.span([Integer(2)*V.gen(0)+Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0)+Integer(12)*V.gen(1), Integer(4)*V.gen(2)]); Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 12)
```

(continues on next page)

(continued from previous page)

```
>>> phi = Q.hom([Integer(0), Q.gen(1)])
>>> phi.inverse_image(Q.submodule([]))
Finitely generated module V/W over Integer Ring with invariants (4)
>>> phi.kernel()
Finitely generated module V/W over Integer Ring with invariants (4)
>>> phi.inverse_image(phi.codomain())
Finitely generated module V/W over Integer Ring with invariants (4, 12)

>>> phi.inverse_image(Q.submodule([Q.gen(0)]))
Finitely generated module V/W over Integer Ring with invariants (4)
>>> phi.inverse_image(Q.submodule([Q.gen(1)]))
Finitely generated module V/W over Integer Ring with invariants (4, 12)

>>> phi.inverse_image(ZZ**Integer(3))
Traceback (most recent call last):
...
TypeError: A must be a finitely generated quotient module
>>> phi.inverse_image(ZZ**Integer(3) / W.scale(Integer(2)))
Traceback (most recent call last):
...
ValueError: A must be a submodule of the codomain
```

kernel()

Compute the kernel of this homomorphism.

EXAMPLES:

```
sage: V = span([[1/2,1,1],[3/2,2,1],[0,0,1]],ZZ); W = V.span([2*V.0+4*V.1,␣
↪9*V.0+12*V.1, 4*V.2])
sage: Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 12)
sage: Q.hom([0, Q.1]).kernel()
Finitely generated module V/W over Integer Ring with invariants (4)
sage: A = Q.hom([Q.0, 0]).kernel(); A
Finitely generated module V/W over Integer Ring with invariants (12)
sage: Q.1 in A
True
sage: phi = Q.hom([Q.0-3*Q.1, Q.0+Q.1])
sage: A = phi.kernel(); A
Finitely generated module V/W over Integer Ring with invariants (4)
sage: phi(A)
Finitely generated module V/W over Integer Ring with invariants ()
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)],[Integer(3)/
↪Integer(2),Integer(2),Integer(1)],[Integer(0),Integer(0),Integer(1)]],ZZ);␣
↪W = V.span([Integer(2)*V.gen(0)+Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0)+Integer(12)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q = V/W; Q
Finitely generated module V/W over Integer Ring with invariants (4, 12)
>>> Q.hom([Integer(0), Q.gen(1)]).kernel()
Finitely generated module V/W over Integer Ring with invariants (4)
```

(continues on next page)

(continued from previous page)

```
>>> A = Q.hom([Q.gen(0), Integer(0)]).kernel(); A
Finitely generated module V/W over Integer Ring with invariants (12)
>>> Q.gen(1) in A
True
>>> phi = Q.hom([Q.gen(0)-Integer(3)*Q.gen(1), Q.gen(0)+Q.gen(1)])
>>> A = phi.kernel(); A
Finitely generated module V/W over Integer Ring with invariants (4)
>>> phi(A)
Finitely generated module V/W over Integer Ring with invariants ()
```

lift (*x*)

Given an element x in the codomain of self, if possible find an element y in the domain such that self(y) == x. Raise a ValueError if no such y exists.

INPUT:

- `x` – element of the codomain of self

EXAMPLES:

```
sage: V = span([[1/2,1,1],[3/2,2,1],[0,0,1]],ZZ); W = V.span([2*V.0+4*V.1,␣
↪9*V.0+12*V.1, 4*V.2])
sage: Q=V/W; phi = Q.hom([2*Q.0, Q.1])
sage: phi.lift(Q.1)
(0, 1)
sage: phi.lift(Q.0)
Traceback (most recent call last):
...
ValueError: no lift of element to domain
sage: phi.lift(2*Q.0)
(1, 0)
sage: phi.lift(2*Q.0+Q.1)
(1, 1)
sage: V = span([[5, -1/2]],ZZ); W = span([[20,-2]],ZZ); Q = V/W; phi=Q.
↪hom([2*Q.0])
sage: x = phi.image().0; phi(phi.lift(x)) == x
True
```

```
>>> from sage.all import *
>>> V = span([[Integer(1)/Integer(2),Integer(1),Integer(1)],[Integer(3)/
↪Integer(2),Integer(2),Integer(1)],[Integer(0),Integer(0),Integer(1)]],ZZ);␣
↪W = V.span([Integer(2)*V.gen(0)+Integer(4)*V.gen(1), Integer(9)*V.
↪gen(0)+Integer(12)*V.gen(1), Integer(4)*V.gen(2)])
>>> Q=V/W; phi = Q.hom([Integer(2)*Q.gen(0), Q.gen(1)])
>>> phi.lift(Q.gen(1))
(0, 1)
>>> phi.lift(Q.gen(0))
Traceback (most recent call last):
...
ValueError: no lift of element to domain
>>> phi.lift(Integer(2)*Q.gen(0))
(1, 0)
>>> phi.lift(Integer(2)*Q.gen(0)+Q.gen(1))
```

(continues on next page)

(continued from previous page)

```
(1, 1)
>>> V = span([[Integer(5), -Integer(1)/Integer(2)]],ZZ); W =␣
↪span([[Integer(20),-Integer(2)]],ZZ); Q = V/W; phi=Q.hom([Integer(2)*Q.
↪gen(0)])
>>> x = phi.image().gen(0); phi(phi.lift(x)) == x
True
```

CHAPTER

FIVE

FINITELY PRESENTED GRADED MODULES

5.1 Finitely generated free graded left modules over connected graded algebras

Let A be a connected graded algebra. Some methods here require in addition that A be an algebra over a field or a PID and that Sage has a description of a basis for A.

For example, let p be a prime number. The mod p Steenrod algebra A_p is a connected algebra over the finite field of p elements. Many of the modules presented here will be defined over A_p, or one of its sub-Hopf algebras. E.g.:

```
sage: A = SteenrodAlgebra(p=2)
```

```
>>> from sage.all import *
>>> A = SteenrodAlgebra(p=Integer(2))
```

However, the current implementation can use any connected graded algebra that has a graded basis where each graded part is finite dimensional. Another good family is the exterior algebras:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
```

```
>>> from sage.all import *
>>> E = ExteriorAlgebra(QQ, names=('x', 'y', 'z',)); (x, y, z,) = E._first_ngens(3)
```

A free module is defined by the graded algebra and an ordered tuple of degrees for the generators:

```
sage: M = A.free_graded_module(generator_degrees=(0,1))
sage: M
Free graded left module on 2 generators over
 mod 2 Steenrod algebra, milnor basis

sage: F.<a,b,c> = E.free_graded_module((0,3,6))
sage: F
Free graded left module on 3 generators over
 The exterior algebra of rank 3 over Rational Field
```

```
>>> from sage.all import *
>>> M = A.free_graded_module(generator_degrees=(Integer(0),Integer(1)))
>>> M
Free graded left module on 2 generators over
 mod 2 Steenrod algebra, milnor basis
```

(continues on next page)

(continued from previous page)

```
>>> F = E.free_graded_module((Integer(0),Integer(3),Integer(6)), names=('a', 'b', 'c',
↪)); (a, b, c,) = F._first_ngens(3)
>>> F
Free graded left module on 3 generators over
 The exterior algebra of rank 3 over Rational Field
```

The resulting free modules will have generators in the degrees as specified:

```
sage: M.generator_degrees()
(0, 1)
sage: F.generator_degrees()
(0, 3, 6)
```

```
>>> from sage.all import *
>>> M.generator_degrees()
(0, 1)
>>> F.generator_degrees()
(0, 3, 6)
```

The default names for the generators are `g[degree]` if they are in distinct degrees, `g[degree, i]` otherwise. They can be given other names, as was done when creating the module `F`:

```
sage: M.generators()
(g[0], g[1])
sage: F.generators()
(a, b, c)
```

```
>>> from sage.all import *
>>> M.generators()
(g[0], g[1])
>>> F.generators()
(a, b, c)
```

The connectivity of a module over a connected graded algebra is the minimum degree of all its module generators. Thus, if the module is non-trivial, the connectivity is an integer:

```
sage: M.connectivity()
0
```

```
>>> from sage.all import *
>>> M.connectivity()
0
```

5.1.1 Module elements

For an A-module with generators $\{g_i\}_{i=1}^N$, any homogeneous element of degree n has the form

$$x = \sum_{i=1}^{N} a_i \cdot g_i \,,$$

where $a_i \in A_{n-\deg(g_i)}$ for all i. The ordered set $\{a_i\}$ is referred to as the coefficients of x.

You can produce module elements from a given set of coefficients:

```
sage: coeffs = [Sq(5), Sq(1,1)]
sage: x = M(coeffs); x
Sq(5)*g[0] + Sq(1,1)*g[1]
```

```
>>> from sage.all import *
>>> coeffs = [Sq(Integer(5)), Sq(Integer(1),Integer(1))]
>>> x = M(coeffs); x
Sq(5)*g[0] + Sq(1,1)*g[1]
```

You can also use the module action:

```
sage: Sq(2) * x
(Sq(4,1)+Sq(7))*g[0] + Sq(3,1)*g[1]
```

```
>>> from sage.all import *
>>> Sq(Integer(2)) * x
(Sq(4,1)+Sq(7))*g[0] + Sq(3,1)*g[1]
```

Each nonzero element has a well-defined degree:

```
sage: x.degree()
5
```

```
>>> from sage.all import *
>>> x.degree()
5
```

However the zero element does not:

```
sage: zero = M.zero(); zero
0
sage: zero.degree()
Traceback (most recent call last):
...
ValueError: the zero element does not have a well-defined degree
```

```
>>> from sage.all import *
>>> zero = M.zero(); zero
0
>>> zero.degree()
Traceback (most recent call last):
...
ValueError: the zero element does not have a well-defined degree
```

Any two elements can be added as long as they are in the same degree:

```
sage: y = M.an_element(5); y
Sq(2,1)*g[0] + Sq(4)*g[1]
sage: x + y
(Sq(2,1)+Sq(5))*g[0] + (Sq(1,1)+Sq(4))*g[1]
```

```
>>> from sage.all import *
>>> y = M.an_element(Integer(5)); y
Sq(2,1)*g[0] + Sq(4)*g[1]
>>> x + y
(Sq(2,1)+Sq(5))*g[0] + (Sq(1,1)+Sq(4))*g[1]
```

or when at least one of them is zero:

```
sage: x + zero == x
True
```

```
>>> from sage.all import *
>>> x + zero == x
True
```

Finally, additive inverses exist:

```
sage: x - x
0
```

```
>>> from sage.all import *
>>> x - x
0
```

For every integer n, the set of module elements of degree n form a free module over the ground ring k. A basis for this free module can be computed:

```
sage: M.basis_elements(5)
(Sq(2,1)*g[0], Sq(5)*g[0], Sq(1,1)*g[1], Sq(4)*g[1])
```

```
>>> from sage.all import *
>>> M.basis_elements(Integer(5))
(Sq(2,1)*g[0], Sq(5)*g[0], Sq(1,1)*g[1], Sq(4)*g[1])
```

together with a corresponding free module presentation:

```
sage: M.vector_presentation(5)
Vector space of dimension 4 over Finite Field of size 2
```

```
>>> from sage.all import *
>>> M.vector_presentation(Integer(5))
Vector space of dimension 4 over Finite Field of size 2
```

Given any element, its coordinates with respect to this basis can be computed:

```
sage: v = x.vector_presentation(); v
(0, 1, 1, 0)
```

```
>>> from sage.all import *
>>> v = x.vector_presentation(); v
(0, 1, 1, 0)
```

Going the other way, any element can be constructed by specifying its coordinates:

```
sage: x_ = M.element_from_coordinates((0,1,1,0), 5)
sage: x_
Sq(5)*g[0] + Sq(1,1)*g[1]
sage: x_ == x
True
```

```
>>> from sage.all import *
>>> x_ = M.element_from_coordinates((Integer(0),Integer(1),Integer(1),Integer(0)),␣
↪Integer(5))
>>> x_
Sq(5)*g[0] + Sq(1,1)*g[1]
>>> x_ == x
True
```

5.1.2 Module homomorphisms

Homomorphisms of free graded A-modules $M \to N$ are linear maps of their underlying free k-module which commute with the A-module structure.

To create a homomorphism, first create the object modeling the set of all such homomorphisms using the free function `Hom`:

```
sage: M = A.free_graded_module((0,1))
sage: N.<c2> = A.free_graded_module((2,))
sage: homspace = Hom(M, N); homspace
Set of Morphisms from Free graded left module on 2 generators
  over mod 2 Steenrod algebra, milnor basis
 to Free graded left module on 1 generator
  over mod 2 Steenrod algebra, milnor basis
 in Category of finite dimensional graded modules with basis
  over mod 2 Steenrod algebra, milnor basis
```

```
>>> from sage.all import *
>>> M = A.free_graded_module((Integer(0),Integer(1)))
>>> N = A.free_graded_module((Integer(2),), names=('c2',)); (c2,) = N._first_ngens(1)
>>> homspace = Hom(M, N); homspace
Set of Morphisms from Free graded left module on 2 generators
  over mod 2 Steenrod algebra, milnor basis
 to Free graded left module on 1 generator
  over mod 2 Steenrod algebra, milnor basis
 in Category of finite dimensional graded modules with basis
  over mod 2 Steenrod algebra, milnor basis
```

Just as module elements, homomorphisms are created using the homspace object. The only argument is a list of module elements in the codomain, corresponding to the module generators of the domain:

```
sage: values = [Sq(2)*c2, Sq(2)*Sq(1)*c2]
sage: f = homspace(values)
```

```
>>> from sage.all import *
>>> values = [Sq(Integer(2))*c2, Sq(Integer(2))*Sq(Integer(1))*c2]
>>> f = homspace(values)
```

The resulting homomorphism is the one sending the i-th generator of the domain to the i-th codomain value given:

```
sage: f
Module morphism:
  From: Free graded left module on 2 generators over mod 2 Steenrod algebra, milnor␣
↪basis
  To:   Free graded left module on 1 generator over mod 2 Steenrod algebra, milnor␣
↪basis
  Defn: g[0] |--> Sq(2)*c2
        g[1] |--> (Sq(0,1)+Sq(3))*c2
```

```
>>> from sage.all import *
>>> f
Module morphism:
  From: Free graded left module on 2 generators over mod 2 Steenrod algebra, milnor␣
↪basis
  To:   Free graded left module on 1 generator over mod 2 Steenrod algebra, milnor␣
↪basis
  Defn: g[0] |--> Sq(2)*c2
        g[1] |--> (Sq(0,1)+Sq(3))*c2
```

Convenience methods exist for creating the trivial morphism:

```
sage: homspace.zero()
Module morphism:
  From: Free graded left module on 2 generators over mod 2 Steenrod algebra, milnor␣
↪basis
  To:   Free graded left module on 1 generator over mod 2 Steenrod algebra, milnor␣
↪basis
  Defn: g[0] |--> 0
        g[1] |--> 0
```

```
>>> from sage.all import *
>>> homspace.zero()
Module morphism:
  From: Free graded left module on 2 generators over mod 2 Steenrod algebra, milnor␣
↪basis
  To:   Free graded left module on 1 generator over mod 2 Steenrod algebra, milnor␣
↪basis
  Defn: g[0] |--> 0
        g[1] |--> 0
```

as well as the identity endomorphism:

```
sage: Hom(M, M).identity()
Module endomorphism of Free graded left module on 2 generators over mod 2 Steenrod␣
↪algebra, milnor basis
  Defn: g[0] |--> g[0]
        g[1] |--> g[1]
```

```
>>> from sage.all import *
>>> Hom(M, M).identity()
Module endomorphism of Free graded left module on 2 generators over mod 2 Steenrod␣
```

(continues on next page)

(continued from previous page)

```
↪algebra, milnor basis
  Defn: g[0] |--> g[0]
        g[1] |--> g[1]
```

Homomorphisms can be evaluated on elements of the domain module:

```
sage: v1 = f(Sq(7)*M.generator(0)); v1
Sq(3,2)*c2

sage: v2 = f(Sq(17)*M.generator(1)); v2
(Sq(11,3)+Sq(13,0,1)+Sq(17,1))*c2
```

```
>>> from sage.all import *
>>> v1 = f(Sq(Integer(7))*M.generator(Integer(0))); v1
Sq(3,2)*c2

>>> v2 = f(Sq(Integer(17))*M.generator(Integer(1))); v2
(Sq(11,3)+Sq(13,0,1)+Sq(17,1))*c2
```

and they respect the module action:

```
sage: v1 == Sq(7)*f(M.generator(0))
True

sage: v2 == Sq(17)*f(M.generator(1))
True
```

```
>>> from sage.all import *
>>> v1 == Sq(Integer(7))*f(M.generator(Integer(0)))
True

>>> v2 == Sq(Integer(17))*f(M.generator(Integer(1)))
True
```

Any non-trivial homomorphism has a well-defined degree:

```
sage: f.degree()
4
```

```
>>> from sage.all import *
>>> f.degree()
4
```

but just as module elements, the trivial homomorphism does not:

```
sage: zero_map = homspace.zero()
sage: zero_map.degree()
Traceback (most recent call last):
...
ValueError: the zero morphism does not have a well-defined degree
```

```
>>> from sage.all import *
>>> zero_map = homspace.zero()
>>> zero_map.degree()
Traceback (most recent call last):
...
ValueError: the zero morphism does not have a well-defined degree
```

Any two homomorphisms can be added as long as they are of the same degree:

```
sage: f2 = homspace([Sq(2)*c2, Sq(3)*c2])
sage: f + f2
Module morphism:
  From: Free graded left module on 2 generators over mod 2 Steenrod algebra, milnor␣
↪basis
  To:   Free graded left module on 1 generator over mod 2 Steenrod algebra, milnor␣
↪basis
  Defn: g[0] |--> 0
        g[1] |--> Sq(0,1)*c2
```

```
>>> from sage.all import *
>>> f2 = homspace([Sq(Integer(2))*c2, Sq(Integer(3))*c2])
>>> f + f2
Module morphism:
  From: Free graded left module on 2 generators over mod 2 Steenrod algebra, milnor␣
↪basis
  To:   Free graded left module on 1 generator over mod 2 Steenrod algebra, milnor␣
↪basis
  Defn: g[0] |--> 0
        g[1] |--> Sq(0,1)*c2
```

or when at least one of them is zero:

```
sage: f + zero_map == f
True
```

```
>>> from sage.all import *
>>> f + zero_map == f
True
```

Finally, additive inverses exist:

```
sage: f - f == 0
True
```

```
>>> from sage.all import *
>>> f - f == Integer(0)
True
```

The restriction of a homomorphism to the free module of n-dimensional module elements is a linear transformation:

```
sage: f_4 = f.vector_presentation(4); f_4
Vector space morphism represented by the matrix:
[0 1 0]
```

(continues on next page)

(continued from previous page)

```
[1 1 1]
[0 1 0]
[0 0 0]
Domain: Vector space of dimension 4 over Finite Field of size 2
Codomain: Vector space of dimension 3 over Finite Field of size 2
```

```
>>> from sage.all import *
>>> f_4 = f.vector_presentation(Integer(4)); f_4
Vector space morphism represented by the matrix:
[0 1 0]
[1 1 1]
[0 1 0]
[0 0 0]
Domain: Vector space of dimension 4 over Finite Field of size 2
Codomain: Vector space of dimension 3 over Finite Field of size 2
```

This is compatible with the vector presentations of its domain and codomain modules:

```
sage: f.domain() is M
True
sage: f.codomain() is N
True
sage: f_4.domain() is M.vector_presentation(4)
True
sage: f_4.codomain() is N.vector_presentation(4 + f.degree())
True
```

```
>>> from sage.all import *
>>> f.domain() is M
True
>>> f.codomain() is N
True
>>> f_4.domain() is M.vector_presentation(Integer(4))
True
>>> f_4.codomain() is N.vector_presentation(Integer(4) + f.degree())
True
```

AUTHORS:

- Robert R. Bruner, Michael J. Catanzaro (2012): Initial version.
- Sverre Lunoee–Nielsen and Koen van Woerden (2019-11-29): Updated the original software to Sage version 8.9.
- Sverre Lunoee–Nielsen (2020-07-01): Refactored the code and added new documentation and tests.

class sage.modules.fp_graded.free_module.**FreeGradedModule**(*algebra*, *generator_degrees*, *category*, *names=None*, ***kwds*)

Bases: CombinatorialFreeModule

Create a finitely generated free graded module over a connected graded algebra, with generators in specified degrees.

INPUT:

- `algebra` – the graded connected algebra over which the module is defined; this algebra must be equipped with a graded basis

- `generator_degrees` – tuple of integers defining the number of generators of the module, and their degrees
- `names` – (optional) the names of the generators. If `names` is a comma-separated string like `'a, b, c'`, then those will be the names. Otherwise, for example if `names` is `abc`, then the names will be `abc(d,i)`.

By default, if all generators are in distinct degrees, then the `names` of the generators will have the form `g_{d}` where `d` is the degree of the generator. If the degrees are not distinct, then the generators will be called `g_{d,i}` where `d` is the degree and `i` is its index in the list of generators in that degree.

EXAMPLES:

```
sage: from sage.modules.fp_graded.free_module import FreeGradedModule
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: M = FreeGradedModule(E, (-1,3))
sage: M
Free graded left module on 2 generators over
 The exterior algebra of rank 3 over Rational Field
sage: M.generator_degrees()
(-1, 3)
sage: a, b = M.generators()
sage: (x*y*b).degree()
5
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.free_module import FreeGradedModule
>>> E = ExteriorAlgebra(QQ, names=('x', 'y', 'z',)); (x, y, z,) = E._first_
↪ngens(3)
>>> M = FreeGradedModule(E, (-Integer(1),Integer(3)))
>>> M
Free graded left module on 2 generators over
 The exterior algebra of rank 3 over Rational Field
>>> M.generator_degrees()
(-1, 3)
>>> a, b = M.generators()
>>> (x*y*b).degree()
5
```

`names` of generators:

```
sage: M.generators()
(g[-1], g[3])
sage: FreeGradedModule(E, (0, 0, 2)).generators()
(g[0, 0], g[0, 1], g[2, 0])
sage: FreeGradedModule(E, (0, 0, 2), names='x, y, z').generators()
(x, y, z)
sage: FreeGradedModule(E, (0, 0, 2), names='xyz').generators()
(xyz[0, 0], xyz[0, 1], xyz[2, 0])
```

```
>>> from sage.all import *
>>> M.generators()
(g[-1], g[3])
>>> FreeGradedModule(E, (Integer(0), Integer(0), Integer(2))).generators()
(g[0, 0], g[0, 1], g[2, 0])
>>> FreeGradedModule(E, (Integer(0), Integer(0), Integer(2)), names='x, y, z').
↪generators()
```

(continues on next page)

(continued from previous page)

```
(x, y, z)
>>> FreeGradedModule(E, (Integer(0), Integer(0), Integer(2)), names='xyz').
↪generators()
(xyz[0, 0], xyz[0, 1], xyz[2, 0])
```

`names` can also be defined implicitly using Sage's `M.<...>` syntax:

```
sage: A = SteenrodAlgebra(2)
sage: M.<x,y,z> = FreeGradedModule(A, (-2,2,4))
sage: M
Free graded left module on 3 generators over
 mod 2 Steenrod algebra, milnor basis
sage: M.gens()
(x, y, z)
```

```
>>> from sage.all import *
>>> A = SteenrodAlgebra(Integer(2))
>>> M = FreeGradedModule(A, (-Integer(2),Integer(2),Integer(4)), names=('x', 'y',
↪'z',)); (x, y, z,) = M._first_ngens(3)
>>> M
Free graded left module on 3 generators over
 mod 2 Steenrod algebra, milnor basis
>>> M.gens()
(x, y, z)
```

Element

alias of *FreeGradedModuleElement*

an_element(*n=None*)

Return an element of `self`.

This function chooses deterministically an element of the module in the given degree.

INPUT:

- `n` – (optional) the degree of the element to construct

OUTPUT:

An element (of the given degree if specified).

EXAMPLES:

```
sage: from sage.modules.fp_graded.free_module import FreeGradedModule
sage: A = SteenrodAlgebra(2)
sage: M = FreeGradedModule(A, (0,2,4))
sage: M.an_element(172)
Sq(0,0,2,0,1,0,1)*g[0] + Sq(0,4,0,0,1,0,1)*g[2] + Sq(7,1,0,0,1,0,1)*g[4]
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.free_module import FreeGradedModule
>>> A = SteenrodAlgebra(Integer(2))
>>> M = FreeGradedModule(A, (Integer(0),Integer(2),Integer(4)))
>>> M.an_element(Integer(172))
Sq(0,0,2,0,1,0,1)*g[0] + Sq(0,4,0,0,1,0,1)*g[2] + Sq(7,1,0,0,1,0,1)*g[4]
```

Zero is the only element in the trivial module:

```
sage: FreeGradedModule(A, ()).an_element()
0
```

```
>>> from sage.all import *
>>> FreeGradedModule(A, ()).an_element()
0
```

basis_elements(*n*)

Return a basis for the free module of degree `n` module elements.

Note

Suppose `self` is a module over the graded algebra A with base ring R. This returns a basis as a free module over R, not a basis as a free module over A.

INPUT:

- `n` – integer

OUTPUT:

A sequence of homogeneous module elements of degree `n`, which is a basis for the free module of all degree `n` module elements.

See also

vector_presentation(), *element_from_coordinates()*

EXAMPLES:

```
sage: A = SteenrodAlgebra(2)
sage: M.<m0, m2, m4> = A.free_graded_module((0,2,4))
sage: M.basis_elements(8)
(Sq(1,0,1)*m0,
 Sq(2,2)*m0,
 Sq(5,1)*m0,
 Sq(8)*m0,
 Sq(0,2)*m2,
 Sq(3,1)*m2,
 Sq(6)*m2,
 Sq(1,1)*m4,
 Sq(4)*m4)
```

```
>>> from sage.all import *
>>> A = SteenrodAlgebra(Integer(2))
>>> M = A.free_graded_module((Integer(0),Integer(2),Integer(4)), names=('m0',
↪'m2', 'm4',)); (m0, m2, m4,) = M._first_ngens(3)
>>> M.basis_elements(Integer(8))
(Sq(1,0,1)*m0,
 Sq(2,2)*m0,
```

(continues on next page)

(continued from previous page)

```
 Sq(5,1)*m0,
 Sq(8)*m0,
 Sq(0,2)*m2,
 Sq(3,1)*m2,
 Sq(6)*m2,
 Sq(1,1)*m4,
 Sq(4)*m4)
```

change_ring(*algebra*)

Change the base ring of `self`.

INPUT:

- `algebra` – a connected graded algebra

OUTPUT:

The free graded module over `algebra` defined with the same number of generators of the same degrees as `self`.

EXAMPLES:

```
sage: from sage.modules.fp_graded.free_module import FreeGradedModule
sage: A = SteenrodAlgebra(2)
sage: A2 = SteenrodAlgebra(2, profile=(3,2,1))

sage: M = FreeGradedModule(A, [0,1])
sage: N = M.change_ring(A2); N
Free graded left module on 2 generators over sub-Hopf algebra of
 mod 2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.free_module import FreeGradedModule
>>> A = SteenrodAlgebra(Integer(2))
>>> A2 = SteenrodAlgebra(Integer(2), profile=(Integer(3),Integer(2),
↪Integer(1)))

>>> M = FreeGradedModule(A, [Integer(0),Integer(1)])
>>> N = M.change_ring(A2); N
Free graded left module on 2 generators over sub-Hopf algebra of
 mod 2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
```

Changing back yields the original module:

```
sage: N.change_ring(A) is M
True
```

```
>>> from sage.all import *
>>> N.change_ring(A) is M
True
```

connectivity()

The connectivity of `self`.

OUTPUT:

An integer equal to the minimal degree of all the generators, if this module is non-trivial. Otherwise, $+\infty$.

EXAMPLES:

```
sage: from sage.modules.fp_graded.free_module import FreeGradedModule
sage: A = SteenrodAlgebra(2)
sage: M = FreeGradedModule(A, (-2,2,4))
sage: M.connectivity()
-2
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.free_module import FreeGradedModule
>>> A = SteenrodAlgebra(Integer(2))
>>> M = FreeGradedModule(A, (-Integer(2),Integer(2),Integer(4)))
>>> M.connectivity()
-2
```

element_from_coordinates(*coordinates*, *n*)

The module element of degree `n` having the given coordinates with respect to the basis of module elements given by *basis_elements()*.

INPUT:

- `coordinates` – a sequence of elements of the ground ring
- `n` – integer

OUTPUT: a module element of degree `n`

See also

vector_presentation(), and *basis_elements()*.

EXAMPLES:

```
sage: A = SteenrodAlgebra(2)
sage: M = A.free_graded_module((0,1))
sage: x = M.element_from_coordinates((0,1,0,1), 5); x
Sq(5)*g[0] + Sq(4)*g[1]
sage: basis = M.basis_elements(5)
sage: y = 0*basis[0] + 1*basis[1] + 0*basis[2] + 1*basis[3]
sage: x == y
True

sage: M.element_from_coordinates((0,0,0,0), 5)
0
```

```
>>> from sage.all import *
>>> A = SteenrodAlgebra(Integer(2))
>>> M = A.free_graded_module((Integer(0),Integer(1)))
>>> x = M.element_from_coordinates((Integer(0),Integer(1),Integer(0),
↪Integer(1)), Integer(5)); x
Sq(5)*g[0] + Sq(4)*g[1]
>>> basis = M.basis_elements(Integer(5))
```

(continues on next page)

(continued from previous page)

```
>>> y = Integer(0)*basis[Integer(0)] + Integer(1)*basis[Integer(1)] +␣
↪Integer(0)*basis[Integer(2)] + Integer(1)*basis[Integer(3)]
>>> x == y
True

>>> M.element_from_coordinates((Integer(0),Integer(0),Integer(0),Integer(0)),␣
↪Integer(5))
0
```

gen(*index*)

Return the module generator with the given index.

EXAMPLES:

```
sage: from sage.modules.fp_graded.free_module import FreeGradedModule
sage: A = SteenrodAlgebra(2)
sage: M = FreeGradedModule(A, (0,2,4))
sage: M.generator(0)
g[0]
sage: M.generator(1)
g[2]
sage: M.generator(2)
g[4]
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.free_module import FreeGradedModule
>>> A = SteenrodAlgebra(Integer(2))
>>> M = FreeGradedModule(A, (Integer(0),Integer(2),Integer(4)))
>>> M.generator(Integer(0))
g[0]
>>> M.generator(Integer(1))
g[2]
>>> M.generator(Integer(2))
g[4]
```

generator(*index*)

Return the module generator with the given index.

EXAMPLES:

```
sage: from sage.modules.fp_graded.free_module import FreeGradedModule
sage: A = SteenrodAlgebra(2)
sage: M = FreeGradedModule(A, (0,2,4))
sage: M.generator(0)
g[0]
sage: M.generator(1)
g[2]
sage: M.generator(2)
g[4]
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.free_module import FreeGradedModule
```

(continues on next page)

(continued from previous page)

```
>>> A = SteenrodAlgebra(Integer(2))
>>> M = FreeGradedModule(A, (Integer(0),Integer(2),Integer(4)))
>>> M.generator(Integer(0))
g[0]
>>> M.generator(Integer(1))
g[2]
>>> M.generator(Integer(2))
g[4]
```

generator_degrees()

The degrees of the module generators.

OUTPUT:

A tuple containing the degrees of the generators for this module, in the order that the generators were given when `self` was constructed.

EXAMPLES:

```
sage: from sage.modules.fp_graded.free_module import FreeGradedModule
sage: A = SteenrodAlgebra(2)
sage: M = FreeGradedModule(A, (-2,2,4))
sage: M.generator_degrees()
(-2, 2, 4)
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.free_module import FreeGradedModule
>>> A = SteenrodAlgebra(Integer(2))
>>> M = FreeGradedModule(A, (-Integer(2),Integer(2),Integer(4)))
>>> M.generator_degrees()
(-2, 2, 4)
```

generators()

Return all the module generators.

EXAMPLES:

```
sage: from sage.modules.fp_graded.free_module import FreeGradedModule
sage: A = SteenrodAlgebra(2)
sage: M = FreeGradedModule(A, (-2,1))
sage: M.generators()
(g[-2], g[1])
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.free_module import FreeGradedModule
>>> A = SteenrodAlgebra(Integer(2))
>>> M = FreeGradedModule(A, (-Integer(2),Integer(1)))
>>> M.generators()
(g[-2], g[1])
```

has_relations()

Return `False` as this has no relations.

This is for compatibility with *FPModule*.

EXAMPLES:

```
sage: from sage.modules.fp_graded.free_module import FreeGradedModule
sage: A = SteenrodAlgebra(2)
sage: F = FreeGradedModule(A, (-2,2,4))
sage: F.has_relations()
False
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.free_module import FreeGradedModule
>>> A = SteenrodAlgebra(Integer(2))
>>> F = FreeGradedModule(A, (-Integer(2),Integer(2),Integer(4)))
>>> F.has_relations()
False
```

is_trivial()

Return `True` if this module is trivial and `False` otherwise.

EXAMPLES:

```
sage: from sage.modules.fp_graded.free_module import FreeGradedModule
sage: A = SteenrodAlgebra(2)
sage: FreeGradedModule(A, (-2,2,4)).is_trivial()
False
sage: FreeGradedModule(A, ()).is_trivial()
True
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.free_module import FreeGradedModule
>>> A = SteenrodAlgebra(Integer(2))
>>> FreeGradedModule(A, (-Integer(2),Integer(2),Integer(4))).is_trivial()
False
>>> FreeGradedModule(A, ()).is_trivial()
True
```

minimal_presentation(*top_dim=None*, *verbose=False*)

Return a minimal presentation of `self`.

OUTPUT: the identity morphism as `self` is free

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A2 = SteenrodAlgebra(2)

sage: M = A2.free_graded_module([0,1])
sage: M.minimal_presentation().is_identity()
True
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A2 = SteenrodAlgebra(Integer(2))

>>> M = A2.free_graded_module([Integer(0),Integer(1)])
```

(continues on next page)

(continued from previous page)

```
>>> M.minimal_presentation().is_identity()
True
```

relations()

Return the relations of `self`, which is `()`.

This is for compatibility with *FPModule*.

EXAMPLES:

```
sage: from sage.modules.fp_graded.free_module import FreeGradedModule
sage: A = SteenrodAlgebra(2)
sage: F = FreeGradedModule(A, (-2,2,4))
sage: F.relations()
()
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.free_module import FreeGradedModule
>>> A = SteenrodAlgebra(Integer(2))
>>> F = FreeGradedModule(A, (-Integer(2),Integer(2),Integer(4)))
>>> F.relations()
()
```

resolution(*k*, *top_dim=None*, *verbose=False*)

Return a free resolution of `self` of length `k`.

Since `self` is free, the initial map in the resolution will be the identity, and the rest of the maps will be zero.

INPUT:

- `k` – nonnegative integer
- `top_dim` – stop the computation at this degree. Ignored, for compatibility with *sage.modules.fp_graded.module.FPModule.resolution()*.
- `verbose` – boolean (default: `False`); controls whether log messages should be emitted

OUTPUT:

A list of homomorphisms $[1_M, 0, 0, \ldots, 0]$ consisting of the identity map on this module followed by zero maps. Other than this module, the other modules in the resolution will be zero.

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: M = E.free_graded_module((1,2))
sage: M.resolution(0)
[Module endomorphism of Free graded left module on 2 generators over The␣
↪exterior algebra of rank 3 over Rational Field
  Defn: g[1] |--> g[1]
        g[2] |--> g[2]]
sage: M.resolution(1)
[Module endomorphism of Free graded left module on 2 generators over The␣
↪exterior algebra of rank 3 over Rational Field
  Defn: g[1] |--> g[1]
        g[2] |--> g[2],
 Module morphism:
```

(continues on next page)

(continued from previous page)

```
   From: Free graded left module on 0 generators over The exterior algebra of␣
↪rank 3 over Rational Field
   To:   Free graded left module on 2 generators over The exterior algebra of␣
↪rank 3 over Rational Field]
sage: M.resolution(4)
[Module endomorphism of Free graded left module on 2 generators over The␣
↪exterior algebra of rank 3 over Rational Field
   Defn: g[1] |--> g[1]
         g[2] |--> g[2],
 Module morphism:
   From: Free graded left module on 0 generators over The exterior algebra of␣
↪rank 3 over Rational Field
   To:   Free graded left module on 2 generators over The exterior algebra of␣
↪rank 3 over Rational Field,
 Module endomorphism of Free graded left module on 0 generators over The␣
↪exterior algebra of rank 3 over Rational Field,
 Module endomorphism of Free graded left module on 0 generators over The␣
↪exterior algebra of rank 3 over Rational Field,
 Module endomorphism of Free graded left module on 0 generators over 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)
>>> M = E.free_graded_module((Integer(1),Integer(2)))
>>> M.resolution(Integer(0))
[Module endomorphism of Free graded left module on 2 generators over The␣
↪exterior algebra of rank 3 over Rational Field
   Defn: g[1] |--> g[1]
         g[2] |--> g[2]]
>>> M.resolution(Integer(1))
[Module endomorphism of Free graded left module on 2 generators over The␣
↪exterior algebra of rank 3 over Rational Field
   Defn: g[1] |--> g[1]
         g[2] |--> g[2],
 Module morphism:
   From: Free graded left module on 0 generators over The exterior algebra of␣
↪rank 3 over Rational Field
   To:   Free graded left module on 2 generators over The exterior algebra of␣
↪rank 3 over Rational Field]
>>> M.resolution(Integer(4))
[Module endomorphism of Free graded left module on 2 generators over The␣
↪exterior algebra of rank 3 over Rational Field
   Defn: g[1] |--> g[1]
         g[2] |--> g[2],
 Module morphism:
   From: Free graded left module on 0 generators over The exterior algebra of␣
↪rank 3 over Rational Field
   To:   Free graded left module on 2 generators over The exterior algebra of␣
↪rank 3 over Rational Field,
 Module endomorphism of Free graded left module on 0 generators over The␣
↪exterior algebra of rank 3 over Rational Field,
```

(continues on next page)

(continued from previous page)

```
 Module endomorphism of Free graded left module on 0 generators over The␣
↪exterior algebra of rank 3 over Rational Field,
 Module endomorphism of Free graded left module on 0 generators over The␣
↪exterior algebra of rank 3 over Rational Field]
```

suspension(*t*)

Suspend `self` by the given degree `t`.

INPUT:

- `t` – integer

OUTPUT:

A module which is isomorphic to this module by a shift of degrees by the integer `t`.

EXAMPLES:

```
sage: from sage.modules.fp_graded.free_module import FreeGradedModule
sage: A = SteenrodAlgebra(2)
sage: M = FreeGradedModule(A, (0,2,4))
sage: M.suspension(4).generator_degrees()
(4, 6, 8)
sage: M.suspension(-4).generator_degrees()
(-4, -2, 0)
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.free_module import FreeGradedModule
>>> A = SteenrodAlgebra(Integer(2))
>>> M = FreeGradedModule(A, (Integer(0),Integer(2),Integer(4)))
>>> M.suspension(Integer(4)).generator_degrees()
(4, 6, 8)
>>> M.suspension(-Integer(4)).generator_degrees()
(-4, -2, 0)
```

vector_presentation(*n*)

Return a free module over the ground ring of the module algebra isomorphic to the degree `n` elements of `self`.

Let $\|$ be the ground ring of the algebra over this module is defined, and let M_n be the free module of module elements of degree `n`.

The return value of this function is the free module $\|^r$ where $r = dim(M_n)$.

The isomorphism between k^r and M_n is given by the bijection taking the standard basis element e_i to the i-th element of the array returned by *basis_elements()*.

INPUT:

- `n` – integer degree

OUTPUT:

A free module over the ground ring of the algebra over which `self` is defined, isomorphic to the free module of module elements of degree `n`.

See also

basis_elements(), *element_from_coordinates()*

EXAMPLES:

```
sage: A1 = SteenrodAlgebra(2, profile=[2,1])
sage: M.<x> = A1.free_graded_module((0,))
sage: M.vector_presentation(3)
Vector space of dimension 2 over Finite Field of size 2
sage: M.basis_elements(3)
(Sq(0,1)*x, Sq(3)*x)
sage: [M.vector_presentation(i).dimension() for i in range(-2, 9)]
[0, 0, 1, 1, 1, 2, 1, 1, 1, 0, 0]
```

```
>>> from sage.all import *
>>> A1 = SteenrodAlgebra(Integer(2), profile=[Integer(2),Integer(1)])
>>> M = A1.free_graded_module((Integer(0),), names=('x',)); (x,) = M._first_
↪ngens(1)
>>> M.vector_presentation(Integer(3))
Vector space of dimension 2 over Finite Field of size 2
>>> M.basis_elements(Integer(3))
(Sq(0,1)*x, Sq(3)*x)
>>> [M.vector_presentation(i).dimension() for i in range(-Integer(2),␣
↪Integer(9))]
[0, 0, 1, 1, 1, 2, 1, 1, 1, 0, 0]
```

5.2 Elements of finitely generated free graded left modules

For an overview, see the *free graded modules documentation*.

AUTHORS:

- Robert R. Bruner, Michael J. Catanzaro (2012): Initial version.
- Sverre Lunoee–Nielsen and Koen van Woerden (2019-11-29): Updated the original software to Sage version 8.9.
- Sverre Lunoee–Nielsen (2020-07-01): Refactored the code and added new documentation and tests.

class sage.modules.fp_graded.free_element.**FreeGradedModuleElement**

Bases: *IndexedFreeModuleElement*

Create a module element of a finitely generated free graded left module over a connected graded algebra.

EXAMPLES:

```
sage: from sage.modules.fp_graded.free_module import FreeGradedModule
sage: M = FreeGradedModule(SteenrodAlgebra(2), (0, 1))

sage: M([0, 0])
0

sage: M([1, 0])
g[0]
```

(continues on next page)

(continued from previous page)

```
sage: M([0, 1])
g[1]

sage: M([Sq(1), 1])
Sq(1)*g[0] + g[1]
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.free_module import FreeGradedModule
>>> M = FreeGradedModule(SteenrodAlgebra(Integer(2)), (Integer(0), Integer(1)))

>>> M([Integer(0), Integer(0)])
0

>>> M([Integer(1), Integer(0)])
g[0]

>>> M([Integer(0), Integer(1)])
g[1]

>>> M([Sq(Integer(1)), Integer(1)])
Sq(1)*g[0] + g[1]
```

degree()

The degree of `self`.

OUTPUT:

The integer degree of this element, or raise an error if this is the zero element.

EXAMPLES:

```
sage: from sage.modules.fp_graded.free_module import *
sage: A = SteenrodAlgebra(2)
sage: M = FreeGradedModule(A, (0,1))
sage: x = M.an_element(7); x
Sq(0,0,1)*g[0] + Sq(3,1)*g[1]
sage: x.degree()
7
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.free_module import *
>>> A = SteenrodAlgebra(Integer(2))
>>> M = FreeGradedModule(A, (Integer(0),Integer(1)))
>>> x = M.an_element(Integer(7)); x
Sq(0,0,1)*g[0] + Sq(3,1)*g[1]
>>> x.degree()
7
```

The zero element has no degree:

```
sage: (x-x).degree()
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: the zero element does not have a well-defined degree
```

```
>>> from sage.all import *
>>> (x-x).degree()
Traceback (most recent call last):
...
ValueError: the zero element does not have a well-defined degree
```

Neither do non-homogeneous elements:

```
sage: y = M.an_element(4)
sage: (x+y).degree()
Traceback (most recent call last):
...
ValueError: this is a nonhomogeneous element, no well-defined degree
```

```
>>> from sage.all import *
>>> y = M.an_element(Integer(4))
>>> (x+y).degree()
Traceback (most recent call last):
...
ValueError: this is a nonhomogeneous element, no well-defined degree
```

dense_coefficient_list(*order=None*)

Return a list of all coefficients of `self`.

INPUT:

- `order` – (optional) an ordering of the basis indexing set

Note that this includes *all* of the coefficients, not just the nonzero ones. By default they appear in the same order as the module generators.

EXAMPLES:

```
sage: from sage.modules.fp_graded.free_module import FreeGradedModule
sage: A = SteenrodAlgebra()
sage: M.<Y,Z> = FreeGradedModule(SteenrodAlgebra(2), (0, 1))
sage: x = M.an_element(7); x
Sq(0,0,1)*Y + Sq(3,1)*Z
sage: x.dense_coefficient_list()
[Sq(0,0,1), Sq(3,1)]
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.free_module import FreeGradedModule
>>> A = SteenrodAlgebra()
>>> M = FreeGradedModule(SteenrodAlgebra(Integer(2)), (Integer(0),␣
↪Integer(1)), names=('Y', 'Z',)); (Y, Z,) = M._first_ngens(2)
>>> x = M.an_element(Integer(7)); x
Sq(0,0,1)*Y + Sq(3,1)*Z
>>> x.dense_coefficient_list()
[Sq(0,0,1), Sq(3,1)]
```

lift_to_free()

Return `self`.

It is provided for compatibility with the method of the same name for *sage.modules.fp_graded.module.FPModule*.

EXAMPLES:

```
sage: from sage.modules.fp_graded.free_module import FreeGradedModule
sage: A = SteenrodAlgebra(2)
sage: M = FreeGradedModule(A, (0,1))
sage: x = M.an_element()
sage: x.lift_to_free() == x
True
sage: x.lift_to_free() is x
True
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.free_module import FreeGradedModule
>>> A = SteenrodAlgebra(Integer(2))
>>> M = FreeGradedModule(A, (Integer(0),Integer(1)))
>>> x = M.an_element()
>>> x.lift_to_free() == x
True
>>> x.lift_to_free() is x
True
```

vector_presentation()

A coordinate vector representing `self` when it is a nonzero homogeneous element.

These are coordinates with respect to the basis chosen by *basis_elements()*. When the element is zero, it has no well defined degree, and this function returns `None`.

OUTPUT:

A vector of elements in the ground ring of the algebra for this module when this element is nonzero. Otherwise, the value `None`.

See also

- *sage.modules.fp_graded.free_module.FreeGradedModule.vector_presentation()*
- *sage.modules.fp_graded.free_module.FreeGradedModule.basis_elements()*
- *sage.modules.fp_graded.free_module.FreeGradedModule.element_from_coordinates()*

EXAMPLES:

```
sage: A2 = SteenrodAlgebra(2, profile=(3,2,1))
sage: M = A2.free_graded_module((0,1))
sage: x = M.an_element(7)
sage: v = x.vector_presentation(); v
(1, 0, 0, 0, 0, 1, 0)
```

(continues on next page)

(continued from previous page)

```
sage: type(v)
<class 'sage.modules.vector_mod2_dense.Vector_mod2_dense'>
sage: M.gen(0).vector_presentation()
(1)
sage: M.gen(1).vector_presentation()
(0, 1)

sage: V = M.vector_presentation(7)
sage: v in V
True

sage: M.element_from_coordinates(v, 7) == x
True
```

```
>>> from sage.all import *
>>> A2 = SteenrodAlgebra(Integer(2), profile=(Integer(3),Integer(2),
↪Integer(1)))
>>> M = A2.free_graded_module((Integer(0),Integer(1)))
>>> x = M.an_element(Integer(7))
>>> v = x.vector_presentation(); v
(1, 0, 0, 0, 0, 1, 0)
>>> type(v)
<class 'sage.modules.vector_mod2_dense.Vector_mod2_dense'>
>>> M.gen(Integer(0)).vector_presentation()
(1)
>>> M.gen(Integer(1)).vector_presentation()
(0, 1)

>>> V = M.vector_presentation(Integer(7))
>>> v in V
True

>>> M.element_from_coordinates(v, Integer(7)) == x
True
```

We can use the basis for the module elements in the degree of x, together with the coefficients v to recreate the element x:

```
sage: basis = M.basis_elements(7)
sage: x_ = sum( [c*b for (c,b) in zip(v, basis)] ); x_
Sq(0,0,1)*g[0] + Sq(3,1)*g[1]
sage: x__ = M.linear_combination(zip(basis, v)); x__
Sq(0,0,1)*g[0] + Sq(3,1)*g[1]
sage: x == x_ == x__
True
```

```
>>> from sage.all import *
>>> basis = M.basis_elements(Integer(7))
>>> x_ = sum( [c*b for (c,b) in zip(v, basis)] ); x_
Sq(0,0,1)*g[0] + Sq(3,1)*g[1]
>>> x__ = M.linear_combination(zip(basis, v)); x__
Sq(0,0,1)*g[0] + Sq(3,1)*g[1]
```

(continues on next page)

(continued from previous page)

```
>>> x == x_ == x__
True
```

This is not defined for elements that are not homogeneous:

```
sage: sum(M.basis()).vector_presentation()
Traceback (most recent call last):
...
ValueError: this is a nonhomogeneous element, no well-defined degree
```

```
>>> from sage.all import *
>>> sum(M.basis()).vector_presentation()
Traceback (most recent call last):
...
ValueError: this is a nonhomogeneous element, no well-defined degree
```

5.3 Homomorphisms of finitely generated free graded left modules

AUTHORS:

- Robert R. Bruner, Michael J. Catanzaro (2012): Initial version.
- Sverre Lunoee–Nielsen and Koen van Woerden (2019-11-29): Updated the original software to Sage version 8.9.
- Sverre Lunoee–Nielsen (2020-07-01): Refactored the code and added new documentation and tests.

class sage.modules.fp_graded.free_morphism.**FreeGradedModuleMorphism**(*parent*, *values*)

Bases: *FPModuleMorphism*

Create a homomorphism from a finitely generated free graded module to a graded module.

INPUT:

- `parent` – a homspace in the category of finitely generated free modules
- `values` – list of elements in the codomain; each element corresponds (by their ordering) to a module generator in the domain

EXAMPLES:

```
sage: from sage.modules.fp_graded.free_module import FreeGradedModule
sage: A = SteenrodAlgebra(2)
sage: F1 = FreeGradedModule(A, (4,5), names='b')
sage: F2 = FreeGradedModule(A, (3,4), names='c')
sage: F3 = FreeGradedModule(A, (2,3), names='d')
sage: H1 = Hom(F1, F2)
sage: H2 = Hom(F2, F3)
sage: f = H1((F2((Sq(4), 0)), F2((0, Sq(4)))))
sage: g = H2((F3((Sq(2), 0)), F3((Sq(3), Sq(2)))))
sage: g*f
Module morphism:
  From: Free graded left module on 2 generators over mod 2 Steenrod algebra,␣
↪milnor basis
  To:   Free graded left module on 2 generators over mod 2 Steenrod algebra,␣
↪milnor basis
```

(continues on next page)

(continued from previous page)

```
  Defn: b[4] |--> (Sq(0,2)+Sq(3,1)+Sq(6))*d[2]
        b[5] |--> (Sq(1,2)+Sq(7))*d[2] + (Sq(0,2)+Sq(3,1)+Sq(6))*d[3]
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.free_module import FreeGradedModule
>>> A = SteenrodAlgebra(Integer(2))
>>> F1 = FreeGradedModule(A, (Integer(4),Integer(5)), names='b')
>>> F2 = FreeGradedModule(A, (Integer(3),Integer(4)), names='c')
>>> F3 = FreeGradedModule(A, (Integer(2),Integer(3)), names='d')
>>> H1 = Hom(F1, F2)
>>> H2 = Hom(F2, F3)
>>> f = H1((F2((Sq(Integer(4)), Integer(0))), F2((Integer(0), Sq(Integer(4))))))
>>> g = H2((F3((Sq(Integer(2)), Integer(0))), F3((Sq(Integer(3)),␣
↪Sq(Integer(2))))))
>>> g*f
Module morphism:
  From: Free graded left module on 2 generators over mod 2 Steenrod algebra,␣
↪milnor basis
  To:   Free graded left module on 2 generators over mod 2 Steenrod algebra,␣
↪milnor basis
  Defn: b[4] |--> (Sq(0,2)+Sq(3,1)+Sq(6))*d[2]
        b[5] |--> (Sq(1,2)+Sq(7))*d[2] + (Sq(0,2)+Sq(3,1)+Sq(6))*d[3]
```

degree()

The degree of `self`.

OUTPUT:

The degree of this homomorphism. Raise an error if this is the trivial homomorphism.

EXAMPLES:

```
sage: from sage.modules.fp_graded.free_module import FreeGradedModule
sage: A = SteenrodAlgebra(2)
sage: homspace = Hom(FreeGradedModule(A, (0,1)), FreeGradedModule(A, (0,)))
sage: N = homspace.codomain()
sage: values = [Sq(5)*N.generator(0), Sq(3,1)*N.generator(0)]
sage: f = homspace(values)
sage: f.degree()
5
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.free_module import FreeGradedModule
>>> A = SteenrodAlgebra(Integer(2))
>>> homspace = Hom(FreeGradedModule(A, (Integer(0),Integer(1))),␣
↪FreeGradedModule(A, (Integer(0),)))
>>> N = homspace.codomain()
>>> values = [Sq(Integer(5))*N.generator(Integer(0)), Sq(Integer(3),
↪Integer(1))*N.generator(Integer(0))]
>>> f = homspace(values)
>>> f.degree()
5
```

The zero homomorphism has no degree:

```
sage: homspace.zero().degree()
Traceback (most recent call last):
...
ValueError: the zero morphism does not have a well-defined degree
```

```
>>> from sage.all import *
>>> homspace.zero().degree()
Traceback (most recent call last):
...
ValueError: the zero morphism does not have a well-defined degree
```

fp_module()

Create a finitely presented module from `self`.

OUTPUT: the finitely presented module with presentation equal to `self`

EXAMPLES:

```
sage: A = SteenrodAlgebra(2)
sage: F1 = A.free_graded_module([2])
sage: F2 = A.free_graded_module([0])
sage: v = F2([Sq(2)])
sage: pres = Hom(F1, F2)([v])
sage: M = pres.fp_module(); M
Finitely presented left module on 1 generator and 1 relation over
 mod 2 Steenrod algebra, milnor basis
sage: M.generator_degrees()
(0,)
sage: M.relations()
(Sq(2)*g[0],)

sage: from sage.modules.fp_graded.module import FPModule
sage: A = SteenrodAlgebra(2)
sage: F1 = A.free_graded_module((2,))
sage: F2 = FPModule(A, (0,), [[Sq(4)]])
sage: v = F2([Sq(2)])
sage: pres = Hom(F1, F2)([v])
sage: pres.fp_module()
Traceback (most recent call last):
...
ValueError: this is not a morphism between free modules
```

```
>>> from sage.all import *
>>> A = SteenrodAlgebra(Integer(2))
>>> F1 = A.free_graded_module([Integer(2)])
>>> F2 = A.free_graded_module([Integer(0)])
>>> v = F2([Sq(Integer(2))])
>>> pres = Hom(F1, F2)([v])
>>> M = pres.fp_module(); M
Finitely presented left module on 1 generator and 1 relation over
 mod 2 Steenrod algebra, milnor basis
>>> M.generator_degrees()
(0,)
```

(continues on next page)

(continued from previous page)

```
>>> M.relations()
(Sq(2)*g[0],)

>>> from sage.modules.fp_graded.module import FPModule
>>> A = SteenrodAlgebra(Integer(2))
>>> F1 = A.free_graded_module((Integer(2),))
>>> F2 = FPModule(A, (Integer(0),), [[Sq(Integer(4))]])
>>> v = F2([Sq(Integer(2))])
>>> pres = Hom(F1, F2)([v])
>>> pres.fp_module()
Traceback (most recent call last):
...
ValueError: this is not a morphism between free modules
```

5.4 Homsets of finitely generated free graded left modules

For an overview, see the *free graded modules documentation*.

EXAMPLES:

```
sage: from sage.modules.fp_graded.free_module import FreeGradedModule
sage: A = SteenrodAlgebra(2)
sage: F1 = FreeGradedModule(A, (1,3), names='g')
sage: F2 = FreeGradedModule(A, (2,3), names='h')
sage: homset = Hom(F1, F2)
sage: homset
Set of Morphisms from Free graded left module on 2 generators ...
sage: homset([F2((Sq(1), 1)), F2((0, Sq(2)))])
Module morphism:
  From: Free graded left module on 2 generators over mod 2 Steenrod algebra, milnor␣
↪basis
  To:   Free graded left module on 2 generators over mod 2 Steenrod algebra, milnor␣
↪basis
  Defn: g[1] |--> Sq(1)*h[2] + h[3]
        g[3] |--> Sq(2)*h[3]
sage: TestSuite(homset).run()
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.free_module import FreeGradedModule
>>> A = SteenrodAlgebra(Integer(2))
>>> F1 = FreeGradedModule(A, (Integer(1),Integer(3)), names='g')
>>> F2 = FreeGradedModule(A, (Integer(2),Integer(3)), names='h')
>>> homset = Hom(F1, F2)
>>> homset
Set of Morphisms from Free graded left module on 2 generators ...
>>> homset([F2((Sq(Integer(1)), Integer(1))), F2((Integer(0), Sq(Integer(2))))])
Module morphism:
  From: Free graded left module on 2 generators over mod 2 Steenrod algebra, milnor␣
↪basis
  To:   Free graded left module on 2 generators over mod 2 Steenrod algebra, milnor␣
↪basis
```

(continues on next page)

(continued from previous page)

```
  Defn: g[1] |--> Sq(1)*h[2] + h[3]
        g[3] |--> Sq(2)*h[3]
>>> TestSuite(homset).run()
```

AUTHORS:

- Robert R. Bruner, Michael J. Catanzaro (2012): Initial version.
- Sverre Lunoee–Nielsen and Koen van Woerden (2019-11-29): Updated the original software to Sage version 8.9.
- Sverre Lunoee–Nielsen (2020-07-01): Refactored the code and added new documentation and tests.

class sage.modules.fp_graded.free_homspace.**FreeGradedModuleHomspace**(*X*, *Y*, *category=None*, *base=None*, *check=True*)

Bases: *FPModuleHomspace*

Homspace between two free graded modules.

Element

alias of *FreeGradedModuleMorphism*

5.5 Finitely presented graded modules

Let R be a connected graded algebra. A finitely presented module over R is a module isomorphic to the cokernel of an R-linear homomorphism $f : F_1 \to F_0$ of finitely generated free modules: the generators of F_0 correspond to the generators of the module, and the generators of F_1 correspond to its relations, via the map f.

This module class takes as input a set of generators and relations and uses them to construct a presentation, using the class *FreeGradedModuleMorphism*. It also allows such a morphism as input.

This package was designed with homological algebra in mind, and its API focuses on maps rather than objects. A good example of this is the kernel function *sage.modules.fp_graded.morphism.FPModuleMorphism.kernel_inclusion()*, which computes the kernel of a homomorphism $f : M \to N$. Its return value is not an instance of the module class, but rather an injective homomorphism $i : K \to M$ with the property that $\mathrm{im}(i) = \ker(f)$.

Note

Some methods here require in addition that R be an algebra over a field or a PID and that Sage has a description of a basis for R.

AUTHORS:

- Robert R. Bruner, Michael J. Catanzaro (2012): Initial version.
- Sverre Lunoee–Nielsen and Koen van Woerden (2019-11-29): Updated the original software to Sage version 8.9.
- Sverre Lunoee–Nielsen (2020-07-01): Refactored the code and added new documentation and tests.

class sage.modules.fp_graded.module.**FPModule**(*j*, *names*)

Bases: UniqueRepresentation, IndexedGenerators, *Module*

A finitely presented module over a connected graded algebra.

INPUT:

One of the following:

- `arg0` – a morphism such that the module is the cokernel, or a free graded module, in which case the output is the same module, viewed as finitely presented

Otherwise:

- `arg0` – the graded connected algebra over which the module is defined; this algebra must be equipped with a graded basis
- `generator_degrees` – tuple of integer degrees
- `relations` – tuple of relations; a relation is a tuple of coefficients $(c_1, \ldots, c_n)$, ordered so that they correspond to the module generators, that is, such a tuple corresponds to the relation

$$c_1 g_1 + \ldots + c_n g_n = 0$$

if the generators are $(g_1, \ldots, g_n)$

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule

sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: M = FPModule(E, [0, 1], [[x, 1]])
sage: a, b = M.generators()
sage: x*a + b == 0
True
sage: (x*a + b).normalize()
0

sage: A3 = SteenrodAlgebra(2, profile=(4,3,2,1))
sage: M = FPModule(A3, [0, 1], [[Sq(2), Sq(1)]])
sage: M.generators()
(g[0], g[1])
sage: M.relations()
(Sq(2)*g[0] + Sq(1)*g[1],)
sage: M.is_trivial()
False

sage: Z = FPModule(A3, [])
sage: Z.generators()
()
sage: Z.relations()
()
sage: Z.is_trivial()
True

sage: from sage.modules.fp_graded.free_module import FreeGradedModule
sage: F = FreeGradedModule(E, [0, 1])
sage: one = Hom(F, F).identity()
sage: Z = FPModule(one)
sage: Z.is_trivial()
True

sage: FPModule(E.free_graded_module([0, 1]))
Free graded left module on 2 generators over The exterior algebra of rank 2 over␣
↪Rational Field
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule

>>> E = ExteriorAlgebra(QQ, names=('x', 'y',)); (x, y,) = E._first_ngens(2)
>>> M = FPModule(E, [Integer(0), Integer(1)], [[x, Integer(1)]])
>>> a, b = M.generators()
>>> x*a + b == Integer(0)
True
>>> (x*a + b).normalize()
0

>>> A3 = SteenrodAlgebra(Integer(2), profile=(Integer(4),Integer(3),Integer(2),
↪Integer(1)))
>>> M = FPModule(A3, [Integer(0), Integer(1)], [[Sq(Integer(2)), Sq(Integer(1))]])
>>> M.generators()
(g[0], g[1])
>>> M.relations()
(Sq(2)*g[0] + Sq(1)*g[1],)
>>> M.is_trivial()
False

>>> Z = FPModule(A3, [])
>>> Z.generators()
()
>>> Z.relations()
()
>>> Z.is_trivial()
True

>>> from sage.modules.fp_graded.free_module import FreeGradedModule
>>> F = FreeGradedModule(E, [Integer(0), Integer(1)])
>>> one = Hom(F, F).identity()
>>> Z = FPModule(one)
>>> Z.is_trivial()
True

>>> FPModule(E.free_graded_module([Integer(0), Integer(1)]))
Free graded left module on 2 generators over The exterior algebra of rank 2 over␣
↪Rational Field
```

Element

alias of *FPElement*

an_element(*n=None*)

An element of this module.

This function chooses deterministically an element, i.e. the output depends only on the module and its input `n`.

INPUT:

- `n` – (optional) the degree of the element to construct

OUTPUT: a module element of the given degree

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A2 = SteenrodAlgebra(2, profile=(3,2,1))
sage: M = FPModule(A2, [0,2,4], [[0, Sq(5), Sq(3)], [Sq(7), 0, Sq(2)*Sq(1)]])

sage: [M.an_element(i) for i in range(10)]
[g[0],
 Sq(1)*g[0],
 Sq(2)*g[0] + g[2],
 Sq(0,1)*g[0] + Sq(1)*g[2],
 Sq(1,1)*g[0] + Sq(2)*g[2] + g[4],
 Sq(2,1)*g[0] + Sq(0,1)*g[2] + Sq(1)*g[4],
 Sq(0,2)*g[0] + Sq(1,1)*g[2] + Sq(2)*g[4],
 Sq(0,0,1)*g[0] + Sq(2,1)*g[2] + Sq(0,1)*g[4],
 Sq(1,0,1)*g[0] + Sq(6)*g[2] + Sq(1,1)*g[4],
 Sq(2,0,1)*g[0] + Sq(4,1)*g[2] + Sq(2,1)*g[4]]
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A2 = SteenrodAlgebra(Integer(2), profile=(Integer(3),Integer(2),
↪Integer(1)))
>>> M = FPModule(A2, [Integer(0),Integer(2),Integer(4)], [[Integer(0),␣
↪Sq(Integer(5)), Sq(Integer(3))], [Sq(Integer(7)), Integer(0),␣
↪Sq(Integer(2))*Sq(Integer(1))]])

>>> [M.an_element(i) for i in range(Integer(10))]
[g[0],
 Sq(1)*g[0],
 Sq(2)*g[0] + g[2],
 Sq(0,1)*g[0] + Sq(1)*g[2],
 Sq(1,1)*g[0] + Sq(2)*g[2] + g[4],
 Sq(2,1)*g[0] + Sq(0,1)*g[2] + Sq(1)*g[4],
 Sq(0,2)*g[0] + Sq(1,1)*g[2] + Sq(2)*g[4],
 Sq(0,0,1)*g[0] + Sq(2,1)*g[2] + Sq(0,1)*g[4],
 Sq(1,0,1)*g[0] + Sq(6)*g[2] + Sq(1,1)*g[4],
 Sq(2,0,1)*g[0] + Sq(4,1)*g[2] + Sq(2,1)*g[4]]
```

basis_elements(*n*, *verbose=False*)

Return a basis for the free module of degree `n` module elements.

Note

Suppose `self` is a module over the graded algebra A with base ring R. This returns a basis as a free module over R.

INPUT:

- `n` – integer
- `verbose` – boolean (default: `False`); controls whether log messages should be emitted

OUTPUT:

A list of homogeneous module elements of degree `n` which is a basis for the free module of all degree `n` module elements.

See also

vector_presentation(), *element_from_coordinates()*

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A2 = SteenrodAlgebra(2, profile=(3,2,1))
sage: M.<m0,m2> = FPModule(A2, [0,2], [[Sq(4), Sq(2)], [0, Sq(6)]])

sage: M.basis_elements(4)
(Sq(1,1)*m0, Sq(4)*m0)

sage: M.basis_elements(5)
(Sq(2,1)*m0, Sq(5)*m0, Sq(0,1)*m2)

sage: M.basis_elements(25)
()

sage: M.basis_elements(0)
(m0,)

sage: M.basis_elements(2)
(Sq(2)*m0, m2)
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A2 = SteenrodAlgebra(Integer(2), profile=(Integer(3),Integer(2),
↪Integer(1)))
>>> M = FPModule(A2, [Integer(0),Integer(2)], [[Sq(Integer(4)),␣
↪Sq(Integer(2))], [Integer(0), Sq(Integer(6))]], names=('m0', 'm2',)); (m0,␣
↪m2,) = M._first_ngens(2)

>>> M.basis_elements(Integer(4))
(Sq(1,1)*m0, Sq(4)*m0)

>>> M.basis_elements(Integer(5))
(Sq(2,1)*m0, Sq(5)*m0, Sq(0,1)*m2)

>>> M.basis_elements(Integer(25))
()

>>> M.basis_elements(Integer(0))
(m0,)

>>> M.basis_elements(Integer(2))
(Sq(2)*m0, m2)
```

change_ring(*algebra*)

Change the base ring of `self`.

INPUT:

- `algebra` – a connected graded algebra

OUTPUT:

The finitely presented module over `algebra` defined with the exact same number of generators of the same degrees and relations as `self`.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A = SteenrodAlgebra(2)
sage: A2 = SteenrodAlgebra(2, profile=(3,2,1))

sage: M = FPModule(A, [0,1], [[Sq(2), Sq(1)]])
sage: N = M.change_ring(A2); N
Finitely presented left module on 2 generators and 1 relation over
 sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function␣
↪[3, 2, 1]
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A = SteenrodAlgebra(Integer(2))
>>> A2 = SteenrodAlgebra(Integer(2), profile=(Integer(3),Integer(2),
↪Integer(1)))

>>> M = FPModule(A, [Integer(0),Integer(1)], [[Sq(Integer(2)),␣
↪Sq(Integer(1))]])
>>> N = M.change_ring(A2); N
Finitely presented left module on 2 generators and 1 relation over
 sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function␣
↪[3, 2, 1]
```

Changing back yields the original module:

```
sage: N.change_ring(A) is M
True
```

```
>>> from sage.all import *
>>> N.change_ring(A) is M
True
```

connectivity()

The connectivity of `self`.

Since a finitely presented module over a connected algebra is in particular bounded below, the connectivity is an integer when the module is non-trivial, and $+\infty$ when the module is trivial.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A = SteenrodAlgebra(2)

sage: M = FPModule(A, [0,2,4], [[0, Sq(5), Sq(3)], [Sq(7), 0, Sq(2)*Sq(1)]])
sage: M.connectivity()
0

sage: G = FPModule(A, [0,2], [[1,0]])
```

(continues on next page)

(continued from previous page)

```
sage: G.connectivity()
2
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A = SteenrodAlgebra(Integer(2))

>>> M = FPModule(A, [Integer(0),Integer(2),Integer(4)], [[Integer(0),␣
↪Sq(Integer(5)), Sq(Integer(3))], [Sq(Integer(7)), Integer(0),␣
↪Sq(Integer(2))*Sq(Integer(1))]])
>>> M.connectivity()
0

>>> G = FPModule(A, [Integer(0),Integer(2)], [[Integer(1),Integer(0)]])
>>> G.connectivity()
2
```

defining_homomorphism()

Return the homomorphism defining `self`.

`self` is a finitely presented module defined as the cokernel of a map $j : F_0 o F_1$ of free modules, and this returns that map.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: M = FPModule(E, [0, 1], [[x, 1]])
sage: M.defining_homomorphism()
Module morphism:
  From: Free graded left module on 1 generator over The exterior algebra of␣
↪rank 2 over Rational Field
  To:   Free graded left module on 2 generators over The exterior algebra of␣
↪rank 2 over Rational Field
  Defn: g[1] |--> x*g[0] + g[1]
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> E = ExteriorAlgebra(QQ, names=('x', 'y',)); (x, y,) = E._first_ngens(2)
>>> M = FPModule(E, [Integer(0), Integer(1)], [[x, Integer(1)]])
>>> M.defining_homomorphism()
Module morphism:
  From: Free graded left module on 1 generator over The exterior algebra of␣
↪rank 2 over Rational Field
  To:   Free graded left module on 2 generators over The exterior algebra of␣
↪rank 2 over Rational Field
  Defn: g[1] |--> x*g[0] + g[1]
```

element_from_coordinates(*coordinates*, *n*)

Return the module element in degree `n` having the given coordinates with respect to the basis returned by *basis_elements()*.

This function is inverse to *sage.modules.fp_graded.element.FPElement.vector_presentation()*.

INPUT:

- `coordinates` – a vector of coordinates
- `n` – the degree of the element to construct

OUTPUT:

A module element of degree `n` having the given coordinates with respect to the basis returned by *basis_elements()*.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A = SteenrodAlgebra(2)
sage: M = FPModule(A, [0], [[Sq(4)], [Sq(7)], [Sq(4)*Sq(9)]])

sage: M.vector_presentation(12).dimension()
3
sage: x = M.element_from_coordinates((0,1,0), 12); x
Sq(0,4)*g[0]
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A = SteenrodAlgebra(Integer(2))
>>> M = FPModule(A, [Integer(0)], [[Sq(Integer(4))], [Sq(Integer(7))],␣
↪[Sq(Integer(4))*Sq(Integer(9))]])

>>> M.vector_presentation(Integer(12)).dimension()
3
>>> x = M.element_from_coordinates((Integer(0),Integer(1),Integer(0)),␣
↪Integer(12)); x
Sq(0,4)*g[0]
```

Applying the inverse function brings us back to the coordinate form:

```
sage: x.vector_presentation()
(0, 1, 0)
```

```
>>> from sage.all import *
>>> x.vector_presentation()
(0, 1, 0)
```

See also

sage.modules.fp_graded.module.FPModule.vector_presentation()

gen(*index*)

Return the module generator with the given index.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A4 = SteenrodAlgebra(2, profile=(4,3,2,1))
```

(continues on next page)

(continued from previous page)

```
sage: M = FPModule(A4, [0,2,3])
sage: M.generator(0)
g[0]

sage: N = FPModule(A4, [0, 1], [[Sq(2), Sq(1)]], names='h')
sage: N.generator(1)
h[1]
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A4 = SteenrodAlgebra(Integer(2), profile=(Integer(4),Integer(3),
↪Integer(2),Integer(1)))

>>> M = FPModule(A4, [Integer(0),Integer(2),Integer(3)])
>>> M.generator(Integer(0))
g[0]

>>> N = FPModule(A4, [Integer(0), Integer(1)], [[Sq(Integer(2)),␣
↪Sq(Integer(1))]], names='h')
>>> N.generator(Integer(1))
h[1]
```

generator(*index*)

Return the module generator with the given index.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A4 = SteenrodAlgebra(2, profile=(4,3,2,1))

sage: M = FPModule(A4, [0,2,3])
sage: M.generator(0)
g[0]

sage: N = FPModule(A4, [0, 1], [[Sq(2), Sq(1)]], names='h')
sage: N.generator(1)
h[1]
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A4 = SteenrodAlgebra(Integer(2), profile=(Integer(4),Integer(3),
↪Integer(2),Integer(1)))

>>> M = FPModule(A4, [Integer(0),Integer(2),Integer(3)])
>>> M.generator(Integer(0))
g[0]

>>> N = FPModule(A4, [Integer(0), Integer(1)], [[Sq(Integer(2)),␣
↪Sq(Integer(1))]], names='h')
>>> N.generator(Integer(1))
h[1]
```

generator_degrees()

Return the degrees of the generators for `self`.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A4 = SteenrodAlgebra(2, profile=(4,3,2,1))
sage: N = FPModule(A4, [0, 1], [[Sq(2), Sq(1)]])

sage: N.generator_degrees()
(0, 1)
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A4 = SteenrodAlgebra(Integer(2), profile=(Integer(4),Integer(3),
↪Integer(2),Integer(1)))
>>> N = FPModule(A4, [Integer(0), Integer(1)], [[Sq(Integer(2)),␣
↪Sq(Integer(1))]])

>>> N.generator_degrees()
(0, 1)
```

generators()

Return the generators of `self`.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A4 = SteenrodAlgebra(2, profile=(4,3,2,1))

sage: M = FPModule(A4, [0,0,2,3])
sage: M.generators()
(g[0, 0], g[0, 1], g[2, 0], g[3, 0])

sage: N = FPModule(A4, [0, 1], [[Sq(2), Sq(1)]], names='h')
sage: N.generators()
(h[0], h[1])

sage: Z = FPModule(A4, [])
sage: Z.generators()
()
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A4 = SteenrodAlgebra(Integer(2), profile=(Integer(4),Integer(3),
↪Integer(2),Integer(1)))

>>> M = FPModule(A4, [Integer(0),Integer(0),Integer(2),Integer(3)])
>>> M.generators()
(g[0, 0], g[0, 1], g[2, 0], g[3, 0])

>>> N = FPModule(A4, [Integer(0), Integer(1)], [[Sq(Integer(2)),␣
↪Sq(Integer(1))]], names='h')
>>> N.generators()
```

(continues on next page)

(continued from previous page)

```
(h[0], h[1])

>>> Z = FPModule(A4, [])
>>> Z.generators()
()
```

gens()

Return the generators of `self`.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A4 = SteenrodAlgebra(2, profile=(4,3,2,1))

sage: M = FPModule(A4, [0,0,2,3])
sage: M.generators()
(g[0, 0], g[0, 1], g[2, 0], g[3, 0])

sage: N = FPModule(A4, [0, 1], [[Sq(2), Sq(1)]], names='h')
sage: N.generators()
(h[0], h[1])

sage: Z = FPModule(A4, [])
sage: Z.generators()
()
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A4 = SteenrodAlgebra(Integer(2), profile=(Integer(4),Integer(3),
↪Integer(2),Integer(1)))

>>> M = FPModule(A4, [Integer(0),Integer(0),Integer(2),Integer(3)])
>>> M.generators()
(g[0, 0], g[0, 1], g[2, 0], g[3, 0])

>>> N = FPModule(A4, [Integer(0), Integer(1)], [[Sq(Integer(2)),␣
↪Sq(Integer(1))]], names='h')
>>> N.generators()
(h[0], h[1])

>>> Z = FPModule(A4, [])
>>> Z.generators()
()
```

has_relations()

Return `True` if no relations are defined, and `False` otherwise.

Note

This module is free if this function returns `False`, but a free module can have (redundant) relations.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A2 = SteenrodAlgebra(2, profile=(3,2,1))

sage: F = FPModule(A2, [1,2])
sage: F.has_relations()
False

sage: M = FPModule(A2, [1,2], [[Sq(2), Sq(1)]])
sage: M.has_relations()
True
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A2 = SteenrodAlgebra(Integer(2), profile=(Integer(3),Integer(2),
↪Integer(1)))

>>> F = FPModule(A2, [Integer(1),Integer(2)])
>>> F.has_relations()
False

>>> M = FPModule(A2, [Integer(1),Integer(2)], [[Sq(Integer(2)),␣
↪Sq(Integer(1))]])
>>> M.has_relations()
True
```

A free module constructed with a redundant generator and relation:

```
sage: N = FPModule(A2, [0, 0], [[0, 1]])
sage: N.has_relations()
True
sage: # Computing a minimal presentation reveals an
....: # isomorphic module with no relations.
sage: N_min = N.minimal_presentation().domain()
sage: N_min.has_relations()
False
```

```
>>> from sage.all import *
>>> N = FPModule(A2, [Integer(0), Integer(0)], [[Integer(0), Integer(1)]])
>>> N.has_relations()
True
>>> # Computing a minimal presentation reveals an
... # isomorphic module with no relations.
>>> N_min = N.minimal_presentation().domain()
>>> N_min.has_relations()
False
```

is_trivial()

Return `True` if `self` is isomorphic to the trivial module and `False` otherwise.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A = SteenrodAlgebra(2)
```

(continues on next page)

(continued from previous page)

```
sage: A2 = SteenrodAlgebra(2, profile=(3,2,1))

sage: M = FPModule(A2, [])
sage: M.is_trivial()
True

sage: N = FPModule(A, [1,2])
sage: N.is_trivial()
False

sage: P = FPModule(A, [1,2], [[1,0], [0,1]])
sage: P.is_trivial()
True
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A = SteenrodAlgebra(Integer(2))
>>> A2 = SteenrodAlgebra(Integer(2), profile=(Integer(3),Integer(2),
↪Integer(1)))

>>> M = FPModule(A2, [])
>>> M.is_trivial()
True

>>> N = FPModule(A, [Integer(1),Integer(2)])
>>> N.is_trivial()
False

>>> P = FPModule(A, [Integer(1),Integer(2)], [[Integer(1),Integer(0)],␣
↪[Integer(0),Integer(1)]])
>>> P.is_trivial()
True
```

minimal_presentation(*top_dim=None*, *verbose=False*)

Return a minimal presentation of `self`.

OUTPUT:

An isomorphism $M \to S$, where M has minimal presentation and S is `self`.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A2 = SteenrodAlgebra(2, profile=(3,2,1))

sage: M = FPModule(A2, [0,1], [[Sq(2),Sq(1)],[0,Sq(2)],[Sq(3),0]])
sage: i = M.minimal_presentation()
sage: M_min = i.domain()
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A2 = SteenrodAlgebra(Integer(2), profile=(Integer(3),Integer(2),
↪Integer(1)))
```

(continues on next page)

(continued from previous page)

```
>>> M = FPModule(A2, [Integer(0),Integer(1)], [[Sq(Integer(2)),
↪Sq(Integer(1))],[Integer(0),Sq(Integer(2))],[Sq(Integer(3)),Integer(0)]])
>>> i = M.minimal_presentation()
>>> M_min = i.domain()
```

`i` is an isomorphism between `M_min` and `M`:

```
sage: i.codomain() is M
True
sage: i.is_injective()
True
sage: i.is_surjective()
True
```

```
>>> from sage.all import *
>>> i.codomain() is M
True
>>> i.is_injective()
True
>>> i.is_surjective()
True
```

There are more relations in `M` than in `M_min`:

```
sage: M.relations()
(Sq(2)*g[0] + Sq(1)*g[1], Sq(2)*g[1], Sq(3)*g[0])
sage: M_min.relations()
(Sq(2)*g[0] + Sq(1)*g[1], Sq(2)*g[1])
```

```
>>> from sage.all import *
>>> M.relations()
(Sq(2)*g[0] + Sq(1)*g[1], Sq(2)*g[1], Sq(3)*g[0])
>>> M_min.relations()
(Sq(2)*g[0] + Sq(1)*g[1], Sq(2)*g[1])
```

monomial()

Return the basis element indexed by `i`.

INPUT:

- `i` – an element of the index set

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: M = FPModule(SteenrodAlgebra(2), [0,1], [[Sq(4), Sq(3)]])
sage: M.monomial(0)
g[0]
sage: M.monomial(1)
g[1]
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> M = FPModule(SteenrodAlgebra(Integer(2)), [Integer(0),Integer(1)],␣
↪[[Sq(Integer(4)), Sq(Integer(3))]])
>>> M.monomial(Integer(0))
g[0]
>>> M.monomial(Integer(1))
g[1]
```

relation(*index*)

Return the module relation of the given index.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A4 = SteenrodAlgebra(2, profile=(4,3,2,1))
sage: N = FPModule(A4, [0, 1], [[Sq(2), Sq(1)]])
sage: N.relation(0)
Sq(2)*g[0] + Sq(1)*g[1]
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A4 = SteenrodAlgebra(Integer(2), profile=(Integer(4),Integer(3),
↪Integer(2),Integer(1)))
>>> N = FPModule(A4, [Integer(0), Integer(1)], [[Sq(Integer(2)),␣
↪Sq(Integer(1))]])
>>> N.relation(Integer(0))
Sq(2)*g[0] + Sq(1)*g[1]
```

relations()

Return the relations of `self`.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A4 = SteenrodAlgebra(2, profile=(4,3,2,1))

sage: M = FPModule(A4, [0,2,3])
sage: M.relations()
()

sage: N = FPModule(A4, [0, 1], [[Sq(2), Sq(1)]])
sage: N.relations()
(Sq(2)*g[0] + Sq(1)*g[1],)

sage: Z = FPModule(A4, [])
sage: Z.relations()
()
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A4 = SteenrodAlgebra(Integer(2), profile=(Integer(4),Integer(3),
↪Integer(2),Integer(1)))
```

(continues on next page)

(continued from previous page)

```
>>> M = FPModule(A4, [Integer(0),Integer(2),Integer(3)])
>>> M.relations()
()

>>> N = FPModule(A4, [Integer(0), Integer(1)], [[Sq(Integer(2)),␣
↪Sq(Integer(1))]])
>>> N.relations()
(Sq(2)*g[0] + Sq(1)*g[1],)

>>> Z = FPModule(A4, [])
>>> Z.relations()
()
```

resolution(*k*, *top_dim=None*, *verbose=False*)

Return a free resolution of this module of length `k`.

INPUT:

- `k` – nonnegative integer
- `top_dim` – stop the computation at this degree (default: `None`, but required if the algebra is not finite-dimensional)
- `verbose` – boolean (default: `False`); control if log messages should be emitted

OUTPUT: list of homomorphisms $[\epsilon, f_1, \ldots, f_k]$ such that

$$f_i : F_i \to F_{i-1} \text{ for } 1 < i \leq k, \qquad \epsilon : F_0 \to M,$$

where each F_i is a finitely generated free module, and the sequence

$$F_k \xrightarrow{f_k} F_{k-1} \xrightarrow{f_{k-1}} \ldots \to F_0 \xrightarrow{\epsilon} M \to 0$$

is exact. Note that the 0th element in this list is the map ϵ, and the rest of the maps are between free modules.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule

sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: M = FPModule(E, [0], [[x], [y]])
sage: res = M.resolution(3); res
[Module morphism:
   From: Free graded left module on 1 generator over The exterior algebra of␣
↪rank 2 over Rational Field
   To:   Finitely presented left module on 1 generator and 2 relations over␣
↪The exterior algebra of rank 2 over Rational Field
   Defn: g[0] |--> g[0],
 Module morphism:
   From: Free graded left module on 2 generators over The exterior algebra of␣
↪rank 2 over Rational Field
   To:   Free graded left module on 1 generator over The exterior algebra of␣
↪rank 2 over Rational Field
   Defn: g[1, 0] |--> x*g[0]
```

(continues on next page)

(continued from previous page)

```
         g[1, 1] |--> y*g[0],
 Module morphism:
   From: Free graded left module on 3 generators over The exterior algebra of␣
↪rank 2 over Rational Field
   To:   Free graded left module on 2 generators over The exterior algebra of␣
↪rank 2 over Rational Field
   Defn: g[2, 0] |--> x*g[1, 0]
         g[2, 1] |--> y*g[1, 0] + x*g[1, 1]
         g[2, 2] |--> y*g[1, 1],
 Module morphism:
   From: Free graded left module on 4 generators over The exterior algebra of␣
↪rank 2 over Rational Field
   To:   Free graded left module on 3 generators over The exterior algebra of␣
↪rank 2 over Rational Field
   Defn: g[3, 0] |--> x*g[2, 0]
         g[3, 1] |--> y*g[2, 0] + x*g[2, 1]
         g[3, 2] |--> y*g[2, 1] + x*g[2, 2]
         g[3, 3] |--> y*g[2, 2]]
sage: all((res[i] * res[i+1]).is_zero() for i in range(len(res)-1))
True

sage: e = SymmetricFunctions(QQ).e()
sage: M = FPModule(e, [0], [[e[2]+e[1,1]], [e[1,1]]])
sage: res = M.resolution(3, top_dim=10)
sage: all((res[i] * res[i+1]).is_zero() for i in range(2))
True
sage: res[-1].domain().is_trivial()
True
sage: M = FPModule(e, [0,2], [[e[2]+e[1,1], 0], [e[2,1], e[1]], [0, e[1,1]]])
sage: res = M.resolution(3, top_dim=10)
sage: all((res[i] * res[i+1]).is_zero() for i in range(2))
True
sage: res[-1].domain().is_trivial()
True

sage: A2 = SteenrodAlgebra(2, profile=(3,2,1))
sage: M = FPModule(A2, [0,1], [[Sq(2), Sq(1)]])
sage: M.resolution(0)
[Module morphism:
   From: Free graded left module on 2 generators over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
   To:   Finitely presented left module on 2 generators and 1 relation over␣
↪sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function␣
↪[3, 2, 1]
   Defn: g[0] |--> g[0]
         g[1] |--> g[1]]
sage: res = M.resolution(4, verbose=True)
Computing f_1 (1/4)
Computing f_2 (2/4)
Computing using the profile:
(2, 1)
Resolving the kernel in the range of dimensions [2, 8]: 2 3 4 5 6 7 8.
```

(continues on next page)

(continued from previous page)

```
Computing f_3 (3/4)
Computing using the profile:
(2, 1)
Resolving the kernel in the range of dimensions [8, 14]: 8 9 10 11 12 13 14.
Computing f_4 (4/4)
Computing using the profile:
(2, 1)
Resolving the kernel in the range of dimensions [9, 16]: 9 10 11 12 13 14 15␣
↪16.
sage: len(res)
5
sage: res
[Module morphism:
   From: Free graded left module on 2 generators over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
   To:   Finitely presented left module on 2 generators and 1 relation over␣
↪sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function␣
↪[3, 2, 1]
   Defn: g[0] |--> g[0]
         g[1] |--> g[1],
 Module morphism:
   From: Free graded left module on 1 generator over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
   To:   Free graded left module on 2 generators over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
   Defn: g[2] |--> Sq(2)*g[0] + Sq(1)*g[1],
 Module morphism:
   From: Free graded left module on 1 generator over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
   To:   Free graded left module on 1 generator over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
   Defn: g[8] |--> Sq(3,1)*g[2],
 Module morphism:
   From: Free graded left module on 2 generators over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
   To:   Free graded left module on 1 generator over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
   Defn: g[9] |--> Sq(1)*g[8]
         g[10] |--> Sq(2)*g[8],
 Module morphism:
   From: Free graded left module on 2 generators over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
   To:   Free graded left module on 2 generators over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
   Defn: g[10] |--> Sq(1)*g[9]
         g[12] |--> Sq(0,1)*g[9] + Sq(2)*g[10]]
sage: for i in range(len(res)-1):
....:     assert (res[i] * res[i+1]).is_zero(), 'the result is not a complex'
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
```

(continues on next page)

(continued from previous page)

```
>>> E = ExteriorAlgebra(QQ, names=('x', 'y',)); (x, y,) = E._first_ngens(2)
>>> M = FPModule(E, [Integer(0)], [[x], [y]])
>>> res = M.resolution(Integer(3)); res
[Module morphism:
   From: Free graded left module on 1 generator over The exterior algebra of␣
↪rank 2 over Rational Field
   To:   Finitely presented left module on 1 generator and 2 relations over␣
↪The exterior algebra of rank 2 over Rational Field
   Defn: g[0] |--> g[0],
 Module morphism:
   From: Free graded left module on 2 generators over The exterior algebra of␣
↪rank 2 over Rational Field
   To:   Free graded left module on 1 generator over The exterior algebra of␣
↪rank 2 over Rational Field
   Defn: g[1, 0] |--> x*g[0]
         g[1, 1] |--> y*g[0],
 Module morphism:
   From: Free graded left module on 3 generators over The exterior algebra of␣
↪rank 2 over Rational Field
   To:   Free graded left module on 2 generators over The exterior algebra of␣
↪rank 2 over Rational Field
   Defn: g[2, 0] |--> x*g[1, 0]
         g[2, 1] |--> y*g[1, 0] + x*g[1, 1]
         g[2, 2] |--> y*g[1, 1],
 Module morphism:
   From: Free graded left module on 4 generators over The exterior algebra of␣
↪rank 2 over Rational Field
   To:   Free graded left module on 3 generators over The exterior algebra of␣
↪rank 2 over Rational Field
   Defn: g[3, 0] |--> x*g[2, 0]
         g[3, 1] |--> y*g[2, 0] + x*g[2, 1]
         g[3, 2] |--> y*g[2, 1] + x*g[2, 2]
         g[3, 3] |--> y*g[2, 2]]
>>> all((res[i] * res[i+Integer(1)]).is_zero() for i in range(len(res)-
↪Integer(1)))
True

>>> e = SymmetricFunctions(QQ).e()
>>> M = FPModule(e, [Integer(0)], [[e[Integer(2)]+e[Integer(1),Integer(1)]],␣
↪[e[Integer(1),Integer(1)]]])
>>> res = M.resolution(Integer(3), top_dim=Integer(10))
>>> all((res[i] * res[i+Integer(1)]).is_zero() for i in range(Integer(2)))
True
>>> res[-Integer(1)].domain().is_trivial()
True
>>> M = FPModule(e, [Integer(0),Integer(2)], [[e[Integer(2)]+e[Integer(1),
↪Integer(1)], Integer(0)], [e[Integer(2),Integer(1)], e[Integer(1)]],␣
↪[Integer(0), e[Integer(1),Integer(1)]]])
>>> res = M.resolution(Integer(3), top_dim=Integer(10))
>>> all((res[i] * res[i+Integer(1)]).is_zero() for i in range(Integer(2)))
True
>>> res[-Integer(1)].domain().is_trivial()
```

(continues on next page)

(continued from previous page)

```
True

>>> A2 = SteenrodAlgebra(Integer(2), profile=(Integer(3),Integer(2),
↪Integer(1)))
>>> M = FPModule(A2, [Integer(0),Integer(1)], [[Sq(Integer(2)),␣
↪Sq(Integer(1))]])
>>> M.resolution(Integer(0))
[Module morphism:
   From: Free graded left module on 2 generators over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
   To:   Finitely presented left module on 2 generators and 1 relation over␣
↪sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function␣
↪[3, 2, 1]
   Defn: g[0] |--> g[0]
         g[1] |--> g[1]]
>>> res = M.resolution(Integer(4), verbose=True)
Computing f_1 (1/4)
Computing f_2 (2/4)
Computing using the profile:
(2, 1)
Resolving the kernel in the range of dimensions [2, 8]: 2 3 4 5 6 7 8.
Computing f_3 (3/4)
Computing using the profile:
(2, 1)
Resolving the kernel in the range of dimensions [8, 14]: 8 9 10 11 12 13 14.
Computing f_4 (4/4)
Computing using the profile:
(2, 1)
Resolving the kernel in the range of dimensions [9, 16]: 9 10 11 12 13 14 15␣
↪16.
>>> len(res)
5
>>> res
[Module morphism:
   From: Free graded left module on 2 generators over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
   To:   Finitely presented left module on 2 generators and 1 relation over␣
↪sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function␣
↪[3, 2, 1]
   Defn: g[0] |--> g[0]
         g[1] |--> g[1],
 Module morphism:
   From: Free graded left module on 1 generator over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
   To:   Free graded left module on 2 generators over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
   Defn: g[2] |--> Sq(2)*g[0] + Sq(1)*g[1],
 Module morphism:
   From: Free graded left module on 1 generator over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
   To:   Free graded left module on 1 generator over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
```

(continues on next page)

(continued from previous page)

```
   Defn: g[8] |--> Sq(3,1)*g[2],
 Module morphism:
   From: Free graded left module on 2 generators over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
   To:   Free graded left module on 1 generator over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
   Defn: g[9] |--> Sq(1)*g[8]
         g[10] |--> Sq(2)*g[8],
 Module morphism:
   From: Free graded left module on 2 generators over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
   To:   Free graded left module on 2 generators over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
   Defn: g[10] |--> Sq(1)*g[9]
         g[12] |--> Sq(0,1)*g[9] + Sq(2)*g[10]]
>>> for i in range(len(res)-Integer(1)):
...     assert (res[i] * res[i+Integer(1)]).is_zero(), 'the result is not a␣
↪complex'
```

We construct $\mathbf{F}_3$ as a $\mathbf{Z}$-module (with trivial grading concentrated in degree 0) and compute its resolution:

```
sage: E = ExteriorAlgebra(ZZ, 0)
sage: M = FPModule(E, [0], [[3]])
sage: res = M.resolution(3)
sage: res
[Module morphism:
   From: Free graded left module on 1 generator over The exterior algebra of␣
↪rank 0 over Integer Ring
   To:   Finitely presented left module on 1 generator and 1 relation over␣
↪The exterior algebra of rank 0 over Integer Ring
   Defn: g[0] |--> g[0],
 Module endomorphism of Free graded left module on 1 generator over The␣
↪exterior algebra of rank 0 over Integer Ring
   Defn: g[0] |--> 3*g[0],
 Module morphism:
   From: Free graded left module on 0 generators over The exterior algebra of␣
↪rank 0 over Integer Ring
   To:   Free graded left module on 1 generator over The exterior algebra of␣
↪rank 0 over Integer Ring,
 Module endomorphism of Free graded left module on 0 generators over The␣
↪exterior algebra of rank 0 over Integer Ring]
```

```
>>> from sage.all import *
>>> E = ExteriorAlgebra(ZZ, Integer(0))
>>> M = FPModule(E, [Integer(0)], [[Integer(3)]])
>>> res = M.resolution(Integer(3))
>>> res
[Module morphism:
   From: Free graded left module on 1 generator over The exterior algebra of␣
↪rank 0 over Integer Ring
   To:   Finitely presented left module on 1 generator and 1 relation over␣
↪The exterior algebra of rank 0 over Integer Ring
```

(continues on next page)

(continued from previous page)

```
  Defn: g[0] |--> g[0],
 Module endomorphism of Free graded left module on 1 generator over The␣
↪exterior algebra of rank 0 over Integer Ring
  Defn: g[0] |--> 3*g[0],
 Module morphism:
  From: Free graded left module on 0 generators over The exterior algebra of␣
↪rank 0 over Integer Ring
  To:   Free graded left module on 1 generator over The exterior algebra of␣
↪rank 0 over Integer Ring,
 Module endomorphism of Free graded left module on 0 generators over The␣
↪exterior algebra of rank 0 over Integer Ring]
```

submodule_inclusion(*spanning_elements*)

Return the inclusion morphism of the submodule of `self` spanned by the given elements.

INPUT:

- `spanning_elements` – an iterable of elements

OUTPUT: the inclusion of the submodule into this module

Because a submodule of a finitely presented module need not be finitely presented, this method will only work if the underlying algebra is finite-dimensional. Indeed, the current implementation only works if the algebra has a `top_class` method, which gets used in `sage.modules.fp_graded.morphism._resolve_kernel()`.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A2 = SteenrodAlgebra(2, profile=(3,2,1))

sage: M = FPModule(A2, [0,1], [[Sq(2),Sq(1)]])
sage: i = M.submodule_inclusion([M.generator(0)])
sage: i.codomain() is M
True
sage: i.is_injective()
True
sage: i.domain().generator_degrees()
(0,)
sage: i.domain().relations()
(Sq(3)*g[0],)
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A2 = SteenrodAlgebra(Integer(2), profile=(Integer(3),Integer(2),
↪Integer(1)))

>>> M = FPModule(A2, [Integer(0),Integer(1)], [[Sq(Integer(2)),
↪Sq(Integer(1))]])
>>> i = M.submodule_inclusion([M.generator(Integer(0))])
>>> i.codomain() is M
True
>>> i.is_injective()
True
```

(continues on next page)

(continued from previous page)

```
>>> i.domain().generator_degrees()
(0,)
>>> i.domain().relations()
(Sq(3)*g[0],)
```

suspension(*t*)

Return the suspension of `self` by degree `t`.

INPUT:

- `t` – integer degree by which the module is suspended

OUTPUT:

A module which is identical to this module by a shift of degrees by the integer `t`.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A = SteenrodAlgebra(2)
sage: A2 = SteenrodAlgebra(2, profile=(3,2,1))

sage: Y = FPModule(A2, [0], [[Sq(1)]])
sage: X = Y.suspension(4)
sage: X.generator_degrees()
(4,)
sage: X.relations()
(Sq(1)*g[4],)

sage: M = FPModule(A, [2,3], [[Sq(2), Sq(1)], [0, Sq(2)]])
sage: Q = M.suspension(1)
sage: Q.generator_degrees()
(3, 4)
sage: Q.relations()
(Sq(2)*g[3] + Sq(1)*g[4], Sq(2)*g[4])
sage: Q = M.suspension(-3)
sage: Q.generator_degrees()
(-1, 0)
sage: Q = M.suspension(0)
sage: Q.generator_degrees()
(2, 3)
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A = SteenrodAlgebra(Integer(2))
>>> A2 = SteenrodAlgebra(Integer(2), profile=(Integer(3),Integer(2),
↪Integer(1)))

>>> Y = FPModule(A2, [Integer(0)], [[Sq(Integer(1))]])
>>> X = Y.suspension(Integer(4))
>>> X.generator_degrees()
(4,)
>>> X.relations()
(Sq(1)*g[4],)
```

(continues on next page)

(continued from previous page)

```
>>> M = FPModule(A, [Integer(2),Integer(3)], [[Sq(Integer(2)),␣
↪Sq(Integer(1))], [Integer(0), Sq(Integer(2))]])
>>> Q = M.suspension(Integer(1))
>>> Q.generator_degrees()
(3, 4)
>>> Q.relations()
(Sq(2)*g[3] + Sq(1)*g[4], Sq(2)*g[4])
>>> Q = M.suspension(-Integer(3))
>>> Q.generator_degrees()
(-1, 0)
>>> Q = M.suspension(Integer(0))
>>> Q.generator_degrees()
(2, 3)
```

vector_presentation(*n*, *verbose=False*)

Return a free module isomorphic to the free module of module elements of degree `n`.

INPUT:

- `n` – the degree of the presentation

OUTPUT: a vector space

See also

basis_elements(), *element_from_coordinates()*

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A = SteenrodAlgebra(2)
sage: M = FPModule(A, [0,2,4], [[Sq(4),Sq(2),0]])

sage: V = M.vector_presentation(4)
sage: V.dimension()
3

sage: len(M.basis_elements(4))
3
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A = SteenrodAlgebra(Integer(2))
>>> M = FPModule(A, [Integer(0),Integer(2),Integer(4)], [[Sq(Integer(4)),
↪Sq(Integer(2)),Integer(0)]])

>>> V = M.vector_presentation(Integer(4))
>>> V.dimension()
3

>>> len(M.basis_elements(Integer(4)))
3
```

zero()

Return the zero element.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: M = FPModule(SteenrodAlgebra(2), [0,1], [[Sq(4), Sq(3)]])
sage: M.zero()
0
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> M = FPModule(SteenrodAlgebra(Integer(2)), [Integer(0),Integer(1)],␣
↪[[Sq(Integer(4)), Sq(Integer(3))]])
>>> M.zero()
0
```

5.6 Elements of finitely presented graded modules

AUTHORS:

- Robert R. Bruner, Michael J. Catanzaro (2012): Initial version.
- Sverre Lunoee–Nielsen and Koen van Woerden (2019-11-29): Updated the original software to Sage version 8.9.
- Sverre Lunoee–Nielsen (2020-07-01): Refactored the code and added new documentation and tests.

class sage.modules.fp_graded.element.**FPElement**

Bases: *IndexedFreeModuleElement*

A module element of a finitely presented graded module over a connected graded algebra.

degree()

The degree of `self`.

OUTPUT: the integer degree of `self` or raise an error if the zero element

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: M = FPModule(SteenrodAlgebra(2), [0,1], [[Sq(4), Sq(3)]])
sage: x = M.an_element(7)

sage: x
Sq(0,0,1)*g[0] + Sq(3,1)*g[1]
sage: x.degree()
7
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> M = FPModule(SteenrodAlgebra(Integer(2)), [Integer(0),Integer(1)],␣
↪[[Sq(Integer(4)), Sq(Integer(3))]])
>>> x = M.an_element(Integer(7))

>>> x
```

(continues on next page)

(continued from previous page)

```
Sq(0,0,1)*g[0] + Sq(3,1)*g[1]
>>> x.degree()
7
```

The zero element has no degree:

```
sage: (x-x).degree()
Traceback (most recent call last):
...
ValueError: the zero element does not have a well-defined degree
```

```
>>> from sage.all import *
>>> (x-x).degree()
Traceback (most recent call last):
...
ValueError: the zero element does not have a well-defined degree
```

dense_coefficient_list(*order=None*)

Return a list of all coefficients of `self`.

INPUT:

- `order` – (optional) an ordering of the basis indexing set

Note that this includes *all* of the coefficients, not just the nonzero ones. By default they appear in the same order as the module generators.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A = SteenrodAlgebra()
sage: M = FPModule(SteenrodAlgebra(2), [0,1], [[Sq(4), Sq(3)]])
sage: x = M([Sq(1), 1])
sage: x.dense_coefficient_list()
[Sq(1), 1]
sage: y = Sq(2) * M.generator(1)
sage: y.dense_coefficient_list()
[0, Sq(2)]
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A = SteenrodAlgebra()
>>> M = FPModule(SteenrodAlgebra(Integer(2)), [Integer(0),Integer(1)],␣
↪[[Sq(Integer(4)), Sq(Integer(3))]])
>>> x = M([Sq(Integer(1)), Integer(1)])
>>> x.dense_coefficient_list()
[Sq(1), 1]
>>> y = Sq(Integer(2)) * M.generator(Integer(1))
>>> y.dense_coefficient_list()
[0, Sq(2)]
```

lift_to_free()

Return the lift of `self` to the free module `F`, where `self` is in a quotient of `F`.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: M = FPModule(SteenrodAlgebra(2), [0,1], [[Sq(4), Sq(3)]])
sage: x = M([Sq(1), 1])
sage: x
Sq(1)*g[0] + g[1]
sage: x.parent()
Finitely presented left module on 2 generators and 1 relation over mod 2␣
↪Steenrod algebra, milnor basis
sage: x.lift_to_free()
Sq(1)*g[0] + g[1]
sage: x.lift_to_free().parent()
Free graded left module on 2 generators over mod 2 Steenrod algebra, milnor␣
↪basis
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> M = FPModule(SteenrodAlgebra(Integer(2)), [Integer(0),Integer(1)],␣
↪[[Sq(Integer(4)), Sq(Integer(3))]])
>>> x = M([Sq(Integer(1)), Integer(1)])
>>> x
Sq(1)*g[0] + g[1]
>>> x.parent()
Finitely presented left module on 2 generators and 1 relation over mod 2␣
↪Steenrod algebra, milnor basis
>>> x.lift_to_free()
Sq(1)*g[0] + g[1]
>>> x.lift_to_free().parent()
Free graded left module on 2 generators over mod 2 Steenrod algebra, milnor␣
↪basis
```

normalize()

A normalized form of `self`.

OUTPUT:

An instance representing the same module element as `self` in normalized form.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: M.<a0,b2,c4> = FPModule(SteenrodAlgebra(2), [0,2,4], [[Sq(4),Sq(2),0]])

sage: m = M((Sq(6), 0, Sq(2))); m
Sq(6)*a0 + Sq(2)*c4
sage: m.normalize()
Sq(6)*a0 + Sq(2)*c4
sage: m == m.normalize()
True

sage: n = M((Sq(4), Sq(2), 0)); n
Sq(4)*a0 + Sq(2)*b2
sage: n.normalize()
0
sage: n == n.normalize()
```

(continues on next page)

(continued from previous page)

```
True
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> M = FPModule(SteenrodAlgebra(Integer(2)), [Integer(0),Integer(2),
↪Integer(4)], [[Sq(Integer(4)),Sq(Integer(2)),Integer(0)]], names=('a0', 'b2
↪', 'c4',)); (a0, b2, c4,) = M._first_ngens(3)

>>> m = M((Sq(Integer(6)), Integer(0), Sq(Integer(2)))); m
Sq(6)*a0 + Sq(2)*c4
>>> m.normalize()
Sq(6)*a0 + Sq(2)*c4
>>> m == m.normalize()
True

>>> n = M((Sq(Integer(4)), Sq(Integer(2)), Integer(0))); n
Sq(4)*a0 + Sq(2)*b2
>>> n.normalize()
0
>>> n == n.normalize()
True
```

vector_presentation()

A coordinate vector representing `self` when it is nonzero.

These are coordinates with respect to the basis chosen by *basis_elements()*. When the element is zero, it has no well defined degree, and this function returns `None`.

OUTPUT:

A vector of elements in the ground ring of the algebra for this module when this element is nonzero. Otherwise, the value `None`.

See also

- *sage.modules.fp_graded.module.FPModule.vector_presentation()*
- *sage.modules.fp_graded.module.FPModule.basis_elements()*
- *sage.modules.fp_graded.module.FPModule.element_from_coordinates()*

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A2 = SteenrodAlgebra(2, profile=(3,2,1))
sage: M.<m0,m1> = FPModule(A2, (0,1))

sage: x = M.an_element(7)
sage: v = x.vector_presentation(); v
(1, 0, 0, 0, 0, 1, 0)
sage: type(v)
<class 'sage.modules.vector_mod2_dense.Vector_mod2_dense'>
```

(continues on next page)

(continued from previous page)

```
sage: V = M.vector_presentation(7)
sage: v in V
True

sage: M.element_from_coordinates(v, 7) == x
True
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A2 = SteenrodAlgebra(Integer(2), profile=(Integer(3),Integer(2),
↪Integer(1)))
>>> M = FPModule(A2, (Integer(0),Integer(1)), names=('m0', 'm1',)); (m0, m1,)␣
↪= M._first_ngens(2)

>>> x = M.an_element(Integer(7))
>>> v = x.vector_presentation(); v
(1, 0, 0, 0, 0, 1, 0)
>>> type(v)
<class 'sage.modules.vector_mod2_dense.Vector_mod2_dense'>

>>> V = M.vector_presentation(Integer(7))
>>> v in V
True

>>> M.element_from_coordinates(v, Integer(7)) == x
True
```

We can use the basis for the module elements in the degree of x, together with the coefficients v to recreate the element x:

```
sage: basis = M.basis_elements(7)
sage: x_ = sum( [c*b for (c,b) in zip(v, basis)] ); x_
Sq(0,0,1)*m0 + Sq(3,1)*m1
sage: x__ = M.linear_combination(zip(basis, v)); x__
Sq(0,0,1)*m0 + Sq(3,1)*m1
sage: x == x_ == x__
True
```

```
>>> from sage.all import *
>>> basis = M.basis_elements(Integer(7))
>>> x_ = sum( [c*b for (c,b) in zip(v, basis)] ); x_
Sq(0,0,1)*m0 + Sq(3,1)*m1
>>> x__ = M.linear_combination(zip(basis, v)); x__
Sq(0,0,1)*m0 + Sq(3,1)*m1
>>> x == x_ == x__
True
```

5.7 Homomorphisms of finitely presented graded modules

AUTHORS:

- Robert R. Bruner, Michael J. Catanzaro (2012): Initial version.
- Sverre Lunoee–Nielsen and Koen van Woerden (2019-11-29): Updated the original software to Sage version 8.9.
- Sverre Lunoee–Nielsen (2020-07-01): Refactored the code and added new documentation and tests.

class sage.modules.fp_graded.morphism.**FPModuleMorphism**(*parent*, *values*, *check=True*)

Bases: Morphism

Create a homomorphism between finitely presented graded modules.

INPUT:

- parent – a homspace of finitely presented graded modules
- values – list of elements in the codomain; each element corresponds to a module generator in the domain
- check – boolean (default: True); if True, check that the morphism is well-defined

change_ring(*algebra*)

Change the base ring of self.

INPUT:

- algebra – a graded algebra

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A2 = SteenrodAlgebra(2, profile=(3,2,1))
sage: A3 = SteenrodAlgebra(2, profile=(4,3,2,1))
sage: M = FPModule(A2, [0], relations=[[Sq(1)]])
sage: N = FPModule(A2, [0], relations=[[Sq(4)],[Sq(1)]])

sage: f = Hom(M,N)([A2.Sq(3)*N.generator(0)]); f
Module morphism:
  From: Finitely presented left module on 1 generator and 1 relation over sub-
↪Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function [3,␣
↪2, 1]
  To:   Finitely presented left module on 1 generator and 2 relations over␣
↪sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function␣
↪[3, 2, 1]
  Defn: g[0] |--> Sq(3)*g[0]

sage: f.base_ring()
sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function [3,
↪ 2, 1]

sage: g = f.change_ring(A3)
sage: g.base_ring()
sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function [4,
↪ 3, 2, 1]
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
```

(continues on next page)

(continued from previous page)

```
>>> A2 = SteenrodAlgebra(Integer(2), profile=(Integer(3),Integer(2),
↪Integer(1)))
>>> A3 = SteenrodAlgebra(Integer(2), profile=(Integer(4),Integer(3),
↪Integer(2),Integer(1)))
>>> M = FPModule(A2, [Integer(0)], relations=[[Sq(Integer(1))]])
>>> N = FPModule(A2, [Integer(0)], relations=[[Sq(Integer(4))],
↪[Sq(Integer(1))]])

>>> f = Hom(M,N)([A2.Sq(Integer(3))*N.generator(Integer(0))]); f
Module morphism:
  From: Finitely presented left module on 1 generator and 1 relation over sub-
↪Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function [3,␣
↪2, 1]
  To:   Finitely presented left module on 1 generator and 2 relations over␣
↪sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function␣
↪[3, 2, 1]
  Defn: g[0] |--> Sq(3)*g[0]

>>> f.base_ring()
sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function [3,
↪ 2, 1]

>>> g = f.change_ring(A3)
>>> g.base_ring()
sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function [4,
↪ 3, 2, 1]
```

cokernel_projection()

Return the map to the cokernel of `self`.

OUTPUT:

The natural projection from the codomain of this homomorphism to its cokernel.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A1 = SteenrodAlgebra(2, profile=(2,1))
sage: M = FPModule(A1, [0], [[Sq(2)]])
sage: F = FPModule(A1, [0])

sage: r = Hom(F, M)([A1.Sq(1)*M.generator(0)])
sage: co = r.cokernel_projection(); co
Module morphism:
  From: Finitely presented left module on 1 generator and 1 relation over sub-
↪Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function [2,␣
↪1]
  To:   Finitely presented left module on 1 generator and 2 relations over␣
↪sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function␣
↪[2, 1]
  Defn: g[0] |--> g[0]

sage: co.domain().is_trivial()
False
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A1 = SteenrodAlgebra(Integer(2), profile=(Integer(2),Integer(1)))
>>> M = FPModule(A1, [Integer(0)], [[Sq(Integer(2))]])
>>> F = FPModule(A1, [Integer(0)])

>>> r = Hom(F, M)([A1.Sq(Integer(1))*M.generator(Integer(0))])
>>> co = r.cokernel_projection(); co
Module morphism:
  From: Finitely presented left module on 1 generator and 1 relation over sub-
↪Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function [2,␣
↪1]
  To:   Finitely presented left module on 1 generator and 2 relations over␣
↪sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function␣
↪[2, 1]
  Defn: g[0] |--> g[0]

>>> co.domain().is_trivial()
False
```

degree()

The degree of `self`.

OUTPUT: the integer degree of `self`

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import *
sage: A = SteenrodAlgebra(2)
sage: M = FPModule(A, [0,1], [[Sq(2), Sq(1)]])
sage: N = FPModule(A, [2], [[Sq(4)]])
sage: homspace = Hom(M, N)

sage: values = [Sq(5)*N.generator(0), Sq(3,1)*N.generator(0)]
sage: f = homspace(values)
sage: f.degree()
7
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import *
>>> A = SteenrodAlgebra(Integer(2))
>>> M = FPModule(A, [Integer(0),Integer(1)], [[Sq(Integer(2)),␣
↪Sq(Integer(1))]])
>>> N = FPModule(A, [Integer(2)], [[Sq(Integer(4))]])
>>> homspace = Hom(M, N)

>>> values = [Sq(Integer(5))*N.generator(Integer(0)), Sq(Integer(3),
↪Integer(1))*N.generator(Integer(0))]
>>> f = homspace(values)
>>> f.degree()
7
```

The trivial homomorphism has no degree:

```
sage: homspace.zero().degree()
Traceback (most recent call last):
...
ValueError: the zero morphism does not have a well-defined degree
```

```
>>> from sage.all import *
>>> homspace.zero().degree()
Traceback (most recent call last):
...
ValueError: the zero morphism does not have a well-defined degree
```

fp_module()

Create a finitely presented module from `self`.

OUTPUT:

The finitely presented module having presentation equal to `self` as long as the domain and codomain are free.

EXAMPLES:

We construct examples with free modules that are presented with a redundant relation:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A = SteenrodAlgebra(2)
sage: F1 = FPModule(A, (2,), [[0]])
sage: F2 = FPModule(A, (0,), [[0]])
sage: v = F2([Sq(2)])
sage: pres = Hom(F1, F2)([v])
sage: M = pres.fp_module(); M
Finitely presented left module on 1 generator and 1 relation over
 mod 2 Steenrod algebra, milnor basis
sage: M.generator_degrees()
(0,)
sage: M.relations()
(Sq(2)*g[0],)

sage: F2 = A.free_graded_module((0,))
sage: v = F2([Sq(2)])
sage: pres = Hom(F1, F2)([v])
sage: M = pres.fp_module(); M
Finitely presented left module on 1 generator and 1 relation over
 mod 2 Steenrod algebra, milnor basis
sage: M.generator_degrees()
(0,)
sage: M.relations()
(Sq(2)*g[0],)

sage: F3 = FPModule(A, (0,), [[Sq(4)]])
sage: v = F3([Sq(2)])
sage: pres = Hom(F1, F3)([v])
sage: pres.fp_module()
Traceback (most recent call last):
...
ValueError: this is not a morphism between free modules
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A = SteenrodAlgebra(Integer(2))
>>> F1 = FPModule(A, (Integer(2),), [[Integer(0)]])
>>> F2 = FPModule(A, (Integer(0),), [[Integer(0)]])
>>> v = F2([Sq(Integer(2))])
>>> pres = Hom(F1, F2)([v])
>>> M = pres.fp_module(); M
Finitely presented left module on 1 generator and 1 relation over
 mod 2 Steenrod algebra, milnor basis
>>> M.generator_degrees()
(0,)
>>> M.relations()
(Sq(2)*g[0],)

>>> F2 = A.free_graded_module((Integer(0),))
>>> v = F2([Sq(Integer(2))])
>>> pres = Hom(F1, F2)([v])
>>> M = pres.fp_module(); M
Finitely presented left module on 1 generator and 1 relation over
 mod 2 Steenrod algebra, milnor basis
>>> M.generator_degrees()
(0,)
>>> M.relations()
(Sq(2)*g[0],)

>>> F3 = FPModule(A, (Integer(0),), [[Sq(Integer(4))]])
>>> v = F3([Sq(Integer(2))])
>>> pres = Hom(F1, F3)([v])
>>> pres.fp_module()
Traceback (most recent call last):
...
ValueError: this is not a morphism between free modules
```

homology(*f*, *top_dim=None*, *verbose=False*)

Compute the sub-quotient module of $H(self, f) = \ker(self)/\operatorname{im}(f)$ in a range of degrees.

For a pair of composable morphisms $f : M \to N$ and $g : N \to Q$ of finitely presented modules, the homology module is a finitely presented quotient of the kernel sub module $\ker(g) \subset N$.

INPUT:

- `f` – a homomorphism with codomain equal to the domain of `self` and image contained in the kernel of this homomorphism
- `top_dim` – integer (optional); used by this function to stop the computation at the given degree
- `verbose` – boolean (default: `False`); enable progress messages

OUTPUT:

A quotient homomorphism $\ker(self) \to H$, where H is isomorphic to $H(self, f)$ in degrees less than or equal to `top_dim`.

Note

If the algebra for this module is finite, then no `top_dim` needs to be specified in order to ensure that this function terminates.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A = SteenrodAlgebra(2, profile=(3,2,1))
sage: M = FPModule(A, [0], [[Sq(3)]])
sage: N = FPModule(A, [0], [[Sq(2,2)]])
sage: F = FPModule(A, [0])
sage: f = Hom(M,N)([A.Sq(2)*N.generator(0)])
sage: g = Hom(F, M)([A.Sq(4)*A.Sq(1,2)*M.generator(0)])
sage: ho = f.homology(g)
sage: ho.codomain()
Finitely presented left module on 1 generator and 5 relations over
 sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function␣
↪[3, 2, 1]
sage: ho.codomain().is_trivial()
False
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A = SteenrodAlgebra(Integer(2), profile=(Integer(3),Integer(2),
↪Integer(1)))
>>> M = FPModule(A, [Integer(0)], [[Sq(Integer(3))]])
>>> N = FPModule(A, [Integer(0)], [[Sq(Integer(2),Integer(2))]])
>>> F = FPModule(A, [Integer(0)])
>>> f = Hom(M,N)([A.Sq(Integer(2))*N.generator(Integer(0))])
>>> g = Hom(F, M)([A.Sq(Integer(4))*A.Sq(Integer(1),Integer(2))*M.
↪generator(Integer(0))])
>>> ho = f.homology(g)
>>> ho.codomain()
Finitely presented left module on 1 generator and 5 relations over
 sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function␣
↪[3, 2, 1]
>>> ho.codomain().is_trivial()
False
```

image(*top_dim=None*, *verbose=False*)

Compute the image of `self`.

INPUT:

- `top_dim` – integer (optional); used by this function to stop the computation at the given degree
- `verbose` – boolean (default: `False`); enable progress messages

OUTPUT:

A homomorphism into $\mathrm{im}(self)$ that is an isomorphism in degrees less than or equal to `top_dim`

Note

If the algebra for this module is finite, then no `top_dim` needs to be specified in order to ensure that this function terminates.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A3 = SteenrodAlgebra(2, profile=(4,3,2,1))
sage: F = FPModule(A3, [1,3]);
sage: L = FPModule(A3, [2,3], [[Sq(2),Sq(1)], [0,Sq(2)]]);
sage: H = Hom(F, L);

sage: H([L((A3.Sq(1), 1)), L((0, A3.Sq(2)))]).image() # long time
Module morphism:
  From: Finitely presented left module on 1 generator and 1 relation over sub-
 ↪Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function [4,␣
 ↪3, 2, 1]
  To:   Finitely presented left module on 2 generators and 2 relations over␣
 ↪sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function␣
 ↪[4, 3, 2, 1]
  Defn: g[3] |--> Sq(1)*g[2] + g[3]

sage: M = FPModule(A3, [0,7], [[Sq(1), 0], [Sq(2), 0], [Sq(4), 0], [Sq(8),␣
 ↪Sq(1)], [0, Sq(7)], [0, Sq(0,1,1)+Sq(4,2)]])
sage: F2 = FPModule(A3, [0], [[Sq(1)], [Sq(2)], [Sq(4)], [Sq(8)], [Sq(15)]])
sage: H = Hom(M, F2)
sage: f = H([F2([1]), F2([0])])
sage: K = f.image(verbose=True, top_dim=17)
1. Computing the generators of the image presentation:
Resolving the image in the range of dimensions [0, 17]:
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17.
2. Computing the relations of the image presentation:
Computing using the profile:
(4, 3, 2, 1)
Resolving the kernel in the range of dimensions [0, 17]:
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17.

sage: K.is_injective()  # long time
True
sage: K.domain().generator_degrees()
(0,)
sage: K.domain().relations()
(Sq(1)*g[0], Sq(2)*g[0], Sq(4)*g[0], Sq(8)*g[0])
sage: K.domain().is_trivial()
False
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A3 = SteenrodAlgebra(Integer(2), profile=(Integer(4),Integer(3),
 ↪Integer(2),Integer(1)))
>>> F = FPModule(A3, [Integer(1),Integer(3)]);
```

(continues on next page)

(continued from previous page)

```
>>> L = FPModule(A3, [Integer(2),Integer(3)], [[Sq(Integer(2)),
↪Sq(Integer(1))], [Integer(0),Sq(Integer(2))]]);
>>> H = Hom(F, L);

>>> H([L((A3.Sq(Integer(1)), Integer(1))), L((Integer(0), A3.
↪Sq(Integer(2))))]).image() # long time
Module morphism:
  From: Finitely presented left module on 1 generator and 1 relation over sub-
↪Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function [4,␣
↪3, 2, 1]
  To:   Finitely presented left module on 2 generators and 2 relations over␣
↪sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function␣
↪[4, 3, 2, 1]
  Defn: g[3] |--> Sq(1)*g[2] + g[3]

>>> M = FPModule(A3, [Integer(0),Integer(7)], [[Sq(Integer(1)), Integer(0)],␣
↪[Sq(Integer(2)), Integer(0)], [Sq(Integer(4)), Integer(0)], [Sq(Integer(8)),
↪ Sq(Integer(1))], [Integer(0), Sq(Integer(7))], [Integer(0), Sq(Integer(0),
↪Integer(1),Integer(1))+Sq(Integer(4),Integer(2))]])
>>> F2 = FPModule(A3, [Integer(0)], [[Sq(Integer(1))], [Sq(Integer(2))],␣
↪[Sq(Integer(4))], [Sq(Integer(8))], [Sq(Integer(15))]])
>>> H = Hom(M, F2)
>>> f = H([F2([Integer(1)]), F2([Integer(0)])])
>>> K = f.image(verbose=True, top_dim=Integer(17))
1. Computing the generators of the image presentation:
Resolving the image in the range of dimensions [0, 17]:
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17.
2. Computing the relations of the image presentation:
Computing using the profile:
(4, 3, 2, 1)
Resolving the kernel in the range of dimensions [0, 17]:
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17.

>>> K.is_injective()  # long time
True
>>> K.domain().generator_degrees()
(0,)
>>> K.domain().relations()
(Sq(1)*g[0], Sq(2)*g[0], Sq(4)*g[0], Sq(8)*g[0])
>>> K.domain().is_trivial()
False
```

is_identity()

Decide if `self` is the identity endomorphism.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import *
sage: A = SteenrodAlgebra(2)
sage: M = FPModule(A, [0,1], [[Sq(2), Sq(1)]])
sage: N = FPModule(A, [2], [[Sq(4)]])
sage: values = [Sq(5)*N.generator(0), Sq(3,1)*N.generator(0)]
```

(continues on next page)

(continued from previous page)

```
sage: f = Hom(M, N)(values)
sage: f.is_identity()
False

sage: one = Hom(M, M)(M.generators()); one
Module endomorphism of Finitely presented left module on 2 generators and 1␣
↪relation over mod 2 Steenrod algebra, milnor basis
  Defn: g[0] |--> g[0]
        g[1] |--> g[1]

sage: one.is_identity()
True

sage: M = A.free_graded_module((0,1))
sage: N = A.free_graded_module((2,))
sage: v = N.generator(0)
sage: values = [Sq(5)*v, Sq(3,1)*v]
sage: f = Hom(M, N)(values)
sage: f.is_identity()
False
sage: one = Hom(M, M)(M.generators()); one
Module endomorphism of Free graded left module on 2 generators over mod 2␣
↪Steenrod algebra, milnor basis
  Defn: g[0] |--> g[0]
        g[1] |--> g[1]
sage: one.is_identity()
True
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import *
>>> A = SteenrodAlgebra(Integer(2))
>>> M = FPModule(A, [Integer(0),Integer(1)], [[Sq(Integer(2)),␣
↪Sq(Integer(1))]])
>>> N = FPModule(A, [Integer(2)], [[Sq(Integer(4))]])
>>> values = [Sq(Integer(5))*N.generator(Integer(0)), Sq(Integer(3),
↪Integer(1))*N.generator(Integer(0))]

>>> f = Hom(M, N)(values)
>>> f.is_identity()
False

>>> one = Hom(M, M)(M.generators()); one
Module endomorphism of Finitely presented left module on 2 generators and 1␣
↪relation over mod 2 Steenrod algebra, milnor basis
  Defn: g[0] |--> g[0]
        g[1] |--> g[1]

>>> one.is_identity()
True

>>> M = A.free_graded_module((Integer(0),Integer(1)))
```

(continues on next page)

(continued from previous page)

```
>>> N = A.free_graded_module((Integer(2),))
>>> v = N.generator(Integer(0))
>>> values = [Sq(Integer(5))*v, Sq(Integer(3),Integer(1))*v]
>>> f = Hom(M, N)(values)
>>> f.is_identity()
False
>>> one = Hom(M, M)(M.generators()); one
Module endomorphism of Free graded left module on 2 generators over mod 2␣
↪Steenrod algebra, milnor basis
  Defn: g[0] |--> g[0]
        g[1] |--> g[1]
>>> one.is_identity()
True
```

is_injective(*top_dim=None*, *verbose=False*)

Return `True` if and only if `self` has a trivial kernel.

INPUT:

- `top_dim` – integer (optional); used by this function to stop the computation at the given degree
- `verbose` – boolean (default: `False`); enable progress messages

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A = SteenrodAlgebra(2)

sage: K = FPModule(A, [2], [[Sq(2)]])
sage: HZ = FPModule(A, [0], [[Sq(1)]])

sage: f = Hom(K, HZ)([Sq(2)*HZ([1])])
sage: f.is_injective(top_dim=23)
True
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A = SteenrodAlgebra(Integer(2))

>>> K = FPModule(A, [Integer(2)], [[Sq(Integer(2))]])
>>> HZ = FPModule(A, [Integer(0)], [[Sq(Integer(1))]])

>>> f = Hom(K, HZ)([Sq(Integer(2))*HZ([Integer(1)])])
>>> f.is_injective(top_dim=Integer(23))
True
```

is_surjective()

Return `True` if and only if `self` has a trivial cokernel.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A = SteenrodAlgebra(2)
sage: F = FPModule(A, [0])
```

(continues on next page)

(continued from previous page)

```
sage: f = Hom(F,F)([Sq(1)*F.generator(0)])
sage: f.is_surjective()
False
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A = SteenrodAlgebra(Integer(2))
>>> F = FPModule(A, [Integer(0)])

>>> f = Hom(F,F)([Sq(Integer(1))*F.generator(Integer(0))])
>>> f.is_surjective()
False
```

is_zero()

Decide if `self` is the zero homomorphism.

OUTPUT: the boolean value `True` if `self` is trivial and `False` otherwise

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A = SteenrodAlgebra(2)
sage: M = FPModule(A, [0,1], [[Sq(2), Sq(1)]])
sage: N = FPModule(A, [2], [[Sq(4)]])
sage: values = [Sq(5)*N.generator(0), Sq(3,1)*N.generator(0)]

sage: f = Hom(M, N)(values)
sage: f.is_zero()
False

sage: (f-f).is_zero()
True

sage: M = A.free_graded_module((0,1))
sage: N = A.free_graded_module((2,))
sage: v = N.generator(0)
sage: values = [Sq(5)*v, Sq(3,1)*v]
sage: f = Hom(M, N)(values)
sage: f.is_zero()
False
sage: (f-f).is_zero()
True
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A = SteenrodAlgebra(Integer(2))
>>> M = FPModule(A, [Integer(0),Integer(1)], [[Sq(Integer(2)),␣
↪Sq(Integer(1))]])
>>> N = FPModule(A, [Integer(2)], [[Sq(Integer(4))]])
>>> values = [Sq(Integer(5))*N.generator(Integer(0)), Sq(Integer(3),
↪Integer(1))*N.generator(Integer(0))]
```

(continues on next page)

(continued from previous page)

```
>>> f = Hom(M, N)(values)
>>> f.is_zero()
False

>>> (f-f).is_zero()
True

>>> M = A.free_graded_module((Integer(0),Integer(1)))
>>> N = A.free_graded_module((Integer(2),))
>>> v = N.generator(Integer(0))
>>> values = [Sq(Integer(5))*v, Sq(Integer(3),Integer(1))*v]
>>> f = Hom(M, N)(values)
>>> f.is_zero()
False
>>> (f-f).is_zero()
True
```

kernel_inclusion(*top_dim=None*, *verbose=False*)

Return the kernel of `self`.

INPUT:

- `top_dim` – integer (optional); used by this function to stop the computation at the given degree
- `verbose` – boolean (default: `False`); enable progress messages

OUTPUT:

A homomorphism into $\ker(self)$ which is an isomorphism in degrees less than or equal to `top_dim`.

Note

If the algebra for this module is finite, then no `top_dim` needs to be specified in order to ensure that this function terminates.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A3 = SteenrodAlgebra(2, profile=(4,3,2,1))
sage: F = FPModule(A3, [1,3]);
sage: L = FPModule(A3, [2,3], [[Sq(2),Sq(1)], [0,Sq(2)]]);
sage: H = Hom(F, L);

sage: H([L((A3.Sq(1), 1)), L((0, A3.Sq(2)))]).kernel_inclusion() # long time
Module morphism:
  From: Finitely presented left module on 2 generators and 1 relation over␣
↪sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function␣
↪[4, 3, 2, 1]
  To:   Free graded left module on 2 generators over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [4, 3, 2, 1]
  Defn: g[3] |--> g[3]
        g[4] |--> Sq(0,1)*g[1]
```

(continues on next page)

(continued from previous page)

```
sage: M = FPModule(A3, [0,7], [[Sq(1), 0], [Sq(2), 0], [Sq(4), 0], [Sq(8),␣
↪Sq(1)], [0, Sq(7)], [0, Sq(0,1,1)+Sq(4,2)]])
sage: F2 = FPModule(A3, [0], [[Sq(1)], [Sq(2)], [Sq(4)], [Sq(8)], [Sq(15)]])
sage: H = Hom(M, F2)
sage: f = H([F2([1]), F2([0])])

sage: K = f.kernel_inclusion(verbose=True, top_dim=17)
1. Computing the generators of the kernel presentation:
Resolving the kernel in the range of dimensions [0, 17]:
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17.
2. Computing the relations of the kernel presentation:
Computing using the profile:
(4, 3, 2, 1)
Resolving the kernel in the range of dimensions [7, 17]:
 7 8 9 10 11 12 13 14 15 16 17.

sage: K.domain().generators()
(g[7],)
sage: K.domain().relations()
((Sq(0,1)+Sq(3))*g[7],
 (Sq(0,0,1)+Sq(1,2)+Sq(4,1))*g[7],
 Sq(9)*g[7],
 (Sq(0,1,1)+Sq(4,2))*g[7])

sage: K
Module morphism:
  From: Finitely presented left module on 1 generator and 4 relations over␣
↪sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function␣
↪[4, 3, 2, 1]
  To:   Finitely presented left module on 2 generators and 6 relations over␣
↪sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function␣
↪[4, 3, 2, 1]
  Defn: g[7] |--> g[7]
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A3 = SteenrodAlgebra(Integer(2), profile=(Integer(4),Integer(3),
↪Integer(2),Integer(1)))
>>> F = FPModule(A3, [Integer(1),Integer(3)]);
>>> L = FPModule(A3, [Integer(2),Integer(3)], [[Sq(Integer(2)),
↪Sq(Integer(1))], [Integer(0),Sq(Integer(2))]]);
>>> H = Hom(F, L);

>>> H([L((A3.Sq(Integer(1)), Integer(1))), L((Integer(0), A3.
↪Sq(Integer(2))))]).kernel_inclusion() # long time
Module morphism:
  From: Finitely presented left module on 2 generators and 1 relation over␣
↪sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function␣
↪[4, 3, 2, 1]
  To:   Free graded left module on 2 generators over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [4, 3, 2, 1]
  Defn: g[3] |--> g[3]
```

(continues on next page)

(continued from previous page)

```
        g[4] |--> Sq(0,1)*g[1]

>>> M = FPModule(A3, [Integer(0),Integer(7)], [[Sq(Integer(1)), Integer(0)],␣
↪[Sq(Integer(2)), Integer(0)], [Sq(Integer(4)), Integer(0)], [Sq(Integer(8)),
↪ Sq(Integer(1))], [Integer(0), Sq(Integer(7))], [Integer(0), Sq(Integer(0),
↪Integer(1),Integer(1))+Sq(Integer(4),Integer(2))]])
>>> F2 = FPModule(A3, [Integer(0)], [[Sq(Integer(1))], [Sq(Integer(2))],␣
↪[Sq(Integer(4))], [Sq(Integer(8))], [Sq(Integer(15))]])
>>> H = Hom(M, F2)
>>> f = H([F2([Integer(1)]), F2([Integer(0)])])

>>> K = f.kernel_inclusion(verbose=True, top_dim=Integer(17))
1. Computing the generators of the kernel presentation:
Resolving the kernel in the range of dimensions [0, 17]:
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17.
2. Computing the relations of the kernel presentation:
Computing using the profile:
(4, 3, 2, 1)
Resolving the kernel in the range of dimensions [7, 17]:
 7 8 9 10 11 12 13 14 15 16 17.

>>> K.domain().generators()
(g[7],)
>>> K.domain().relations()
((Sq(0,1)+Sq(3))*g[7],
 (Sq(0,0,1)+Sq(1,2)+Sq(4,1))*g[7],
 Sq(9)*g[7],
 (Sq(0,1,1)+Sq(4,2))*g[7])

>>> K
Module morphism:
  From: Finitely presented left module on 1 generator and 4 relations over␣
↪sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function␣
↪[4, 3, 2, 1]
  To:   Finitely presented left module on 2 generators and 6 relations over␣
↪sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function␣
↪[4, 3, 2, 1]
  Defn: g[7] |--> g[7]
```

lift(*f*, *verbose=False*)

Return a lift of this homomorphism over the given homomorphism `f`.

INPUT:

- `f` – a homomorphism with codomain equal to the codomain of `self`
- `verbose` – boolean (default: `False`); enable progress messages

OUTPUT:

A homomorphism g with the property that `self` equals $f \circ g$. If no lift exist, `None` is returned.

ALGORITHM:

Let s be this homomorphism with L the domain of s. Choose $x_1, \ldots, x_N$ such that $f(x_i) = s(g_i)$, where the g_i's are the module generators of L.

The linear function sending g_i to x_i for every i is well defined if and only if the vector $x = (x_1, \ldots, x_N)$ lies in the nullspace of the coefficient matrix $R = (r_{ij})$ given by the relations of L.

Let $k \in \ker(f)$ solve the matrix equation:

$$R \cdot k = R \cdot x.$$

Define a module homomorphism by sending the generators of L to $x_1 - k_1, \ldots, x_N - k_N$. This is well defined, and is also a lift of this homomorphism over f.

Note that it does not matter how we choose the initial elements x_i: If x' is another choice then $x' - x \in \ker(f)$ and $R \cdot k = R \cdot x$ if and only if $R \cdot (k + x' - x) = R \cdot x'$.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A = SteenrodAlgebra(2)
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A = SteenrodAlgebra(Integer(2))
```

Lifting a map from a free module is always possible:

```
sage: M = FPModule(A, [0], [[Sq(3)]])
sage: N = FPModule(A, [0], [[Sq(2,2)]])
sage: F = FPModule(A, [0])
sage: f = Hom(M,N)([Sq(2)*N.generator(0)])
sage: k = Hom(F,N)([Sq(1)*Sq(2)*N.generator(0)])
sage: f_ = k.lift(f)
sage: f*f_ == k
True
sage: f_
Module morphism:
  From: Free graded left module on 1 generator over mod 2 Steenrod algebra,␣
↪milnor basis
  To:   Finitely presented left module on 1 generator and 1 relation over mod␣
↪2 Steenrod algebra, milnor basis
  Defn: g[0] |--> Sq(1)*g[0]
```

```
>>> from sage.all import *
>>> M = FPModule(A, [Integer(0)], [[Sq(Integer(3))]])
>>> N = FPModule(A, [Integer(0)], [[Sq(Integer(2),Integer(2))]])
>>> F = FPModule(A, [Integer(0)])
>>> f = Hom(M,N)([Sq(Integer(2))*N.generator(Integer(0))])
>>> k = Hom(F,N)([Sq(Integer(1))*Sq(Integer(2))*N.generator(Integer(0))])
>>> f_ = k.lift(f)
>>> f*f_ == k
True
>>> f_
Module morphism:
  From: Free graded left module on 1 generator over mod 2 Steenrod algebra,␣
↪milnor basis
  To:   Finitely presented left module on 1 generator and 1 relation over mod␣
↪2 Steenrod algebra, milnor basis
  Defn: g[0] |--> Sq(1)*g[0]
```

A split projection:

```
sage: A_plus_HZ = FPModule(A, [0,0], [[0, Sq(1)]])
sage: HZ = FPModule(A, [0], [[Sq(1)]])
sage: q = Hom(A_plus_HZ, HZ)([HZ([1]), HZ([1])])
sage: # We can construct a splitting of `q` manually:
sage: split = Hom(HZ,A_plus_HZ)([A_plus_HZ.generator(1)])
sage: (q*split).is_identity()
True
```

```
>>> from sage.all import *
>>> A_plus_HZ = FPModule(A, [Integer(0),Integer(0)], [[Integer(0),␣
↪Sq(Integer(1))]])
>>> HZ = FPModule(A, [Integer(0)], [[Sq(Integer(1))]])
>>> q = Hom(A_plus_HZ, HZ)([HZ([Integer(1)]), HZ([Integer(1)])])
>>> # We can construct a splitting of `q` manually:
>>> split = Hom(HZ,A_plus_HZ)([A_plus_HZ.generator(Integer(1))])
>>> (q*split).is_identity()
True
```

Thus, lifting the identity homomorphism over q should be possible:

```
sage: one = Hom(HZ,HZ).identity()
sage: j = one.lift(q); j
Module morphism:
  From: Finitely presented left module on 1 generator and 1 relation over mod␣
↪2 Steenrod algebra, milnor basis
  To:   Finitely presented left module on 2 generators and 1 relation over␣
↪mod 2 Steenrod algebra, milnor basis
  Defn: g[0] |--> g[0, 1]
sage: q*j
Module endomorphism of Finitely presented left module on 1 generator and 1␣
↪relation over mod 2 Steenrod algebra, milnor basis
  Defn: g[0] |--> g[0]
```

```
>>> from sage.all import *
>>> one = Hom(HZ,HZ).identity()
>>> j = one.lift(q); j
Module morphism:
  From: Finitely presented left module on 1 generator and 1 relation over mod␣
↪2 Steenrod algebra, milnor basis
  To:   Finitely presented left module on 2 generators and 1 relation over␣
↪mod 2 Steenrod algebra, milnor basis
  Defn: g[0] |--> g[0, 1]
>>> q*j
Module endomorphism of Finitely presented left module on 1 generator and 1␣
↪relation over mod 2 Steenrod algebra, milnor basis
  Defn: g[0] |--> g[0]
```

Lifting over the inclusion of the image sub module:

```
sage: A = SteenrodAlgebra(2)
sage: M = FPModule(A, [0], relations=[[Sq(0,1)]])
```

(continues on next page)

(continued from previous page)

```
sage: f = Hom(M,M)([Sq(2)*M.generator(0)])
sage: im = f.image(top_dim=10)
sage: f.lift(im)
Module morphism:
  From: Finitely presented left module on 1 generator and 1 relation over mod␣
↪2 Steenrod algebra, milnor basis
  To:   Finitely presented left module on 1 generator and 2 relations over␣
↪mod 2 Steenrod algebra, milnor basis
  Defn: g[0] |--> g[2]
```

```
>>> from sage.all import *
>>> A = SteenrodAlgebra(Integer(2))
>>> M = FPModule(A, [Integer(0)], relations=[[Sq(Integer(0),Integer(1))]])
>>> f = Hom(M,M)([Sq(Integer(2))*M.generator(Integer(0))])
>>> im = f.image(top_dim=Integer(10))
>>> f.lift(im)
Module morphism:
  From: Finitely presented left module on 1 generator and 1 relation over mod␣
↪2 Steenrod algebra, milnor basis
  To:   Finitely presented left module on 1 generator and 2 relations over␣
↪mod 2 Steenrod algebra, milnor basis
  Defn: g[0] |--> g[2]
```

When a lift cannot be found, the `None` value is returned. By setting the verbose argument to `True`, an explanation of why the lifting failed will be displayed:

```
sage: F2 = FPModule(A, [0,0])
sage: non_surjection = Hom(F2, F2)([F2([1, 0]), F2([0, 0])])
sage: lift = Hom(F2, F2).identity().lift(non_surjection, verbose=True)
The generators of the domain of this homomorphism do not map into
 the image of the homomorphism we are lifting over.
sage: lift is None
True
```

```
>>> from sage.all import *
>>> F2 = FPModule(A, [Integer(0),Integer(0)])
>>> non_surjection = Hom(F2, F2)([F2([Integer(1), Integer(0)]),␣
↪F2([Integer(0), Integer(0)])])
>>> lift = Hom(F2, F2).identity().lift(non_surjection, verbose=True)
The generators of the domain of this homomorphism do not map into
 the image of the homomorphism we are lifting over.
>>> lift is None
True
```

See also

split()

solve(*x*)

Return an element in the inverse image of `x`.

INPUT:

- x – an element of the codomain of this morphism

OUTPUT:

An element of the domain which maps to x under this morphism or None if x was not in the image of this morphism.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A = SteenrodAlgebra(2)
sage: M = FPModule(A, [0], [[Sq(3)]])
sage: N = FPModule(A, [0], [[Sq(2,2)]])
sage: f = Hom(M, N)( [Sq(2)*N.generator(0)] )
sage: y = Sq(1,1)*N.generator(0); y
Sq(1,1)*g[0]
sage: x = f.solve(y); x
Sq(2)*g[0]
sage: y == f(x)
True
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A = SteenrodAlgebra(Integer(2))
>>> M = FPModule(A, [Integer(0)], [[Sq(Integer(3))]])
>>> N = FPModule(A, [Integer(0)], [[Sq(Integer(2),Integer(2))]])
>>> f = Hom(M, N)( [Sq(Integer(2))*N.generator(Integer(0))] )
>>> y = Sq(Integer(1),Integer(1))*N.generator(Integer(0)); y
Sq(1,1)*g[0]
>>> x = f.solve(y); x
Sq(2)*g[0]
>>> y == f(x)
True
```

Trying to lift an element which is not in the image results in a None value:

```
sage: z = f.solve(Sq(1)*N.generator(0))
sage: z is None
True
```

```
>>> from sage.all import *
>>> z = f.solve(Sq(Integer(1))*N.generator(Integer(0)))
>>> z is None
True
```

split(*verbose=False*)

Return a split of self.

INPUT:

- verbose – boolean (default: False); enable progress messages

OUTPUT:

A homomorphism with the property that the composite homomorphism $S \circ f = id$, where S is self, is The identity homomorphism If no such split exist, None is returned.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A = SteenrodAlgebra(2)
sage: M = FPModule(A, [0,0], [[0, Sq(1)]])
sage: N = FPModule(A, [0], [[Sq(1)]])
sage: p = Hom(M, N)([N.generator(0), N.generator(0)])
sage: s = p.split(); s
Module morphism:
  From: Finitely presented left module on 1 generator and 1 relation over mod␣
↪2 Steenrod algebra, milnor basis
  To:   Finitely presented left module on 2 generators and 1 relation over␣
↪mod 2 Steenrod algebra, milnor basis
  Defn: g[0] |--> g[0, 1]
sage: # Verify that `s` is a splitting:
sage: p*s
Module endomorphism of Finitely presented left module on 1 generator and 1␣
↪relation over mod 2 Steenrod algebra, milnor basis
  Defn: g[0] |--> g[0]
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A = SteenrodAlgebra(Integer(2))
>>> M = FPModule(A, [Integer(0),Integer(0)], [[Integer(0), Sq(Integer(1))]])
>>> N = FPModule(A, [Integer(0)], [[Sq(Integer(1))]])
>>> p = Hom(M, N)([N.generator(Integer(0)), N.generator(Integer(0))])
>>> s = p.split(); s
Module morphism:
  From: Finitely presented left module on 1 generator and 1 relation over mod␣
↪2 Steenrod algebra, milnor basis
  To:   Finitely presented left module on 2 generators and 1 relation over␣
↪mod 2 Steenrod algebra, milnor basis
  Defn: g[0] |--> g[0, 1]
>>> # Verify that `s` is a splitting:
>>> p*s
Module endomorphism of Finitely presented left module on 1 generator and 1␣
↪relation over mod 2 Steenrod algebra, milnor basis
  Defn: g[0] |--> g[0]
```

See also

lift()

suspension(*t*)

The suspension of this morphism by the given degree `t`.

INPUT:

- `t` – integer by which the morphism is suspended

OUTPUT: the morphism which is the suspension of `self` by the degree `t`

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A = SteenrodAlgebra(2)
sage: F1 = FPModule(A, [4,5])
sage: F2 = FPModule(A, [5])

sage: f = Hom(F1, F2)( ( F2([Sq(4)]), F2([Sq(5)]) ) ); f
Module morphism:
  From: Free graded left module on 2 generators over mod 2 Steenrod algebra,␣
↪milnor basis
  To:   Free graded left module on 1 generator over mod 2 Steenrod algebra,␣
↪milnor basis
  Defn: g[4] |--> Sq(4)*g[5]
        g[5] |--> Sq(5)*g[5]

sage: e1 = F1([1, 0])
sage: e2 = F1([0, 1])
sage: f(e1)
Sq(4)*g[5]
sage: f(e2)
Sq(5)*g[5]

sage: sf = f.suspension(4); sf
Module morphism:
  From: Free graded left module on 2 generators over mod 2 Steenrod algebra,␣
↪milnor basis
  To:   Free graded left module on 1 generator over mod 2 Steenrod algebra,␣
↪milnor basis
  Defn: g[8] |--> Sq(4)*g[9]
        g[9] |--> Sq(5)*g[9]

sage: sf.domain() is f.domain().suspension(4)
True

sage: sf.codomain() is f.codomain().suspension(4)
True
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A = SteenrodAlgebra(Integer(2))
>>> F1 = FPModule(A, [Integer(4),Integer(5)])
>>> F2 = FPModule(A, [Integer(5)])

>>> f = Hom(F1, F2)( ( F2([Sq(Integer(4))]), F2([Sq(Integer(5))]) ) ); f
Module morphism:
  From: Free graded left module on 2 generators over mod 2 Steenrod algebra,␣
↪milnor basis
  To:   Free graded left module on 1 generator over mod 2 Steenrod algebra,␣
↪milnor basis
  Defn: g[4] |--> Sq(4)*g[5]
        g[5] |--> Sq(5)*g[5]

>>> e1 = F1([Integer(1), Integer(0)])
```

(continues on next page)

(continued from previous page)

```
>>> e2 = F1([Integer(0), Integer(1)])
>>> f(e1)
Sq(4)*g[5]
>>> f(e2)
Sq(5)*g[5]

>>> sf = f.suspension(Integer(4)); sf
Module morphism:
  From: Free graded left module on 2 generators over mod 2 Steenrod algebra,␣
↪milnor basis
  To:   Free graded left module on 1 generator over mod 2 Steenrod algebra,␣
↪milnor basis
  Defn: g[8] |--> Sq(4)*g[9]
        g[9] |--> Sq(5)*g[9]

>>> sf.domain() is f.domain().suspension(Integer(4))
True

>>> sf.codomain() is f.codomain().suspension(Integer(4))
True
```

values()

The values under `self` of the module generators of the domain module.

OUTPUT: a sequence of module elements of the codomain

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A = SteenrodAlgebra(2)
sage: M = FPModule(A, [0,1], [[Sq(2), Sq(1)]])
sage: N = FPModule(A, [2], [[Sq(4)]])
sage: homspace = Hom(M, N)

sage: values = [Sq(5)*N.generator(0), Sq(3,1)*N.generator(0)]
sage: f = homspace(values)

sage: f.values()
(Sq(5)*g[2], Sq(3,1)*g[2])

sage: homspace.zero().values()
(0, 0)

sage: homspace = Hom(A.free_graded_module((0,1)), A.free_graded_module((2,)))
sage: N = homspace.codomain()
sage: values = [Sq(5)*N.generator(0), Sq(3,1)*N.generator(0)]
sage: f = homspace(values)
sage: f.values()
(Sq(5)*g[2], Sq(3,1)*g[2])
sage: homspace.zero().values()
(0, 0)
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A = SteenrodAlgebra(Integer(2))
>>> M = FPModule(A, [Integer(0),Integer(1)], [[Sq(Integer(2)),␣
↪Sq(Integer(1))]])
>>> N = FPModule(A, [Integer(2)], [[Sq(Integer(4))]])
>>> homspace = Hom(M, N)

>>> values = [Sq(Integer(5))*N.generator(Integer(0)), Sq(Integer(3),
↪Integer(1))*N.generator(Integer(0))]
>>> f = homspace(values)

>>> f.values()
(Sq(5)*g[2], Sq(3,1)*g[2])

>>> homspace.zero().values()
(0, 0)

>>> homspace = Hom(A.free_graded_module((Integer(0),Integer(1))), A.free_
↪graded_module((Integer(2),)))
>>> N = homspace.codomain()
>>> values = [Sq(Integer(5))*N.generator(Integer(0)), Sq(Integer(3),
↪Integer(1))*N.generator(Integer(0))]
>>> f = homspace(values)
>>> f.values()
(Sq(5)*g[2], Sq(3,1)*g[2])
>>> homspace.zero().values()
(0, 0)
```

vector_presentation(*n*)

Return the restriction of `self` to the domain module elements of degree `n`.

The restriction of a nonzero module homomorphism to the free module of module elements of degree n is a linear function into the free module of elements of degree $n + d$ belonging to the codomain. Here d is the degree of this homomorphism.

When this homomorphism is zero, it has no well defined degree so the function cannot be presented since we do not know the degree of its codomain. In this case, an error is raised.

INPUT:

- `n` – integer degree

OUTPUT:

A linear function of finite dimensional free modules over the ground ring of the algebra for this module. The domain is isomorphic to the free module of domain elements of degree `n` of `self` via the choice of basis given by *basis_elements()*. If the morphism is zero, an error is raised.

See also

- *sage.modules.fp_graded.module.FPModule.vector_presentation()*
- *sage.modules.fp_graded.module.FPModule.basis_elements()*

- *sage.modules.fp_graded.free_module.FreeGradedModule.vector_presentation()*
- *sage.modules.fp_graded.free_module.FreeGradedModule.basis_elements()*

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A = SteenrodAlgebra(2)
sage: M = FPModule(A, [0,1], [[Sq(2), Sq(1)]])
sage: N = FPModule(A, [2], [[Sq(4)]])
sage: values = [Sq(5)*N.generator(0), Sq(3,1)*N.generator(0)]
sage: f = Hom(M, N)(values)
sage: f.vector_presentation(0)
Vector space morphism represented by the matrix:
[0]
Domain: Vector space quotient V/W of dimension 1 over Finite Field of size 2␣
↪where
V: Vector space of dimension 1 over Finite Field of size 2
W: Vector space of degree 1 and dimension 0 over Finite Field of size 2
Basis matrix:
[]
Codomain: Vector space quotient V/W of dimension 1 over Finite Field of size␣
↪2 where
V: Vector space of dimension 2 over Finite Field of size 2
W: Vector space of degree 2 and dimension 1 over Finite Field of size 2
Basis matrix:
[0 1]
sage: f.vector_presentation(1)
Vector space morphism represented by the matrix:
[0 0]
[0 1]
Domain: Vector space quotient V/W of dimension 2 over Finite Field of size 2␣
↪where
V: Vector space of dimension 2 over Finite Field of size 2
W: Vector space of degree 2 and dimension 0 over Finite Field of size 2
Basis matrix:
[]
Codomain: Vector space quotient V/W of dimension 2 over Finite Field of size␣
↪2 where
V: Vector space of dimension 3 over Finite Field of size 2
W: Vector space of degree 3 and dimension 1 over Finite Field of size 2
Basis matrix:
[0 1 1]
sage: f.vector_presentation(2)
Vector space morphism represented by the matrix:
[0 0]
Domain: Vector space quotient V/W of dimension 1 over Finite Field of size 2␣
↪where
V: Vector space of dimension 2 over Finite Field of size 2
W: Vector space of degree 2 and dimension 1 over Finite Field of size 2
Basis matrix:
[1 1]
```

(continues on next page)

(continued from previous page)

```
Codomain: Vector space quotient V/W of dimension 2 over Finite Field of size␣
↪2 where
V: Vector space of dimension 4 over Finite Field of size 2
W: Vector space of degree 4 and dimension 2 over Finite Field of size 2
Basis matrix:
[0 0 1 0]
[0 0 0 1]

sage: M = A.free_graded_module((0,1))
sage: N = A.free_graded_module((2,))
sage: v = N.generator(0)
sage: values = [Sq(5)*v, Sq(3,1)*v]
sage: f = Hom(M, N)(values)
sage: f.vector_presentation(0)
Vector space morphism represented by the matrix:
[0 1]
Domain: Vector space of dimension 1 over Finite Field of size 2
Codomain: Vector space of dimension 2 over Finite Field of size 2
sage: f.vector_presentation(1)
Vector space morphism represented by the matrix:
[0 0 0]
[0 1 0]
Domain: Vector space of dimension 2 over Finite Field of size 2
Codomain: Vector space of dimension 3 over Finite Field of size 2
sage: f.vector_presentation(2)
Vector space morphism represented by the matrix:
[0 0 1 1]
[0 0 0 0]
Domain: Vector space of dimension 2 over Finite Field of size 2
Codomain: Vector space of dimension 4 over Finite Field of size 2
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A = SteenrodAlgebra(Integer(2))
>>> M = FPModule(A, [Integer(0),Integer(1)], [[Sq(Integer(2)),␣
↪Sq(Integer(1))]])
>>> N = FPModule(A, [Integer(2)], [[Sq(Integer(4))]])
>>> values = [Sq(Integer(5))*N.generator(Integer(0)), Sq(Integer(3),
↪Integer(1))*N.generator(Integer(0))]
>>> f = Hom(M, N)(values)
>>> f.vector_presentation(Integer(0))
Vector space morphism represented by the matrix:
[0]
Domain: Vector space quotient V/W of dimension 1 over Finite Field of size 2␣
↪where
V: Vector space of dimension 1 over Finite Field of size 2
W: Vector space of degree 1 and dimension 0 over Finite Field of size 2
Basis matrix:
[]
Codomain: Vector space quotient V/W of dimension 1 over Finite Field of size␣
↪2 where
V: Vector space of dimension 2 over Finite Field of size 2
```

(continues on next page)

(continued from previous page)

```
W: Vector space of degree 2 and dimension 1 over Finite Field of size 2
Basis matrix:
[0 1]
>>> f.vector_presentation(Integer(1))
Vector space morphism represented by the matrix:
[0 0]
[0 1]
Domain: Vector space quotient V/W of dimension 2 over Finite Field of size 2␣
↪where
V: Vector space of dimension 2 over Finite Field of size 2
W: Vector space of degree 2 and dimension 0 over Finite Field of size 2
Basis matrix:
[]
Codomain: Vector space quotient V/W of dimension 2 over Finite Field of size␣
↪2 where
V: Vector space of dimension 3 over Finite Field of size 2
W: Vector space of degree 3 and dimension 1 over Finite Field of size 2
Basis matrix:
[0 1 1]
>>> f.vector_presentation(Integer(2))
Vector space morphism represented by the matrix:
[0 0]
Domain: Vector space quotient V/W of dimension 1 over Finite Field of size 2␣
↪where
V: Vector space of dimension 2 over Finite Field of size 2
W: Vector space of degree 2 and dimension 1 over Finite Field of size 2
Basis matrix:
[1 1]
Codomain: Vector space quotient V/W of dimension 2 over Finite Field of size␣
↪2 where
V: Vector space of dimension 4 over Finite Field of size 2
W: Vector space of degree 4 and dimension 2 over Finite Field of size 2
Basis matrix:
[0 0 1 0]
[0 0 0 1]

>>> M = A.free_graded_module((Integer(0),Integer(1)))
>>> N = A.free_graded_module((Integer(2),))
>>> v = N.generator(Integer(0))
>>> values = [Sq(Integer(5))*v, Sq(Integer(3),Integer(1))*v]
>>> f = Hom(M, N)(values)
>>> f.vector_presentation(Integer(0))
Vector space morphism represented by the matrix:
[0 1]
Domain: Vector space of dimension 1 over Finite Field of size 2
Codomain: Vector space of dimension 2 over Finite Field of size 2
>>> f.vector_presentation(Integer(1))
Vector space morphism represented by the matrix:
[0 0 0]
[0 1 0]
Domain: Vector space of dimension 2 over Finite Field of size 2
Codomain: Vector space of dimension 3 over Finite Field of size 2
```

(continues on next page)

(continued from previous page)

```
>>> f.vector_presentation(Integer(2))
Vector space morphism represented by the matrix:
[0 0 1 1]
[0 0 0 0]
Domain: Vector space of dimension 2 over Finite Field of size 2
Codomain: Vector space of dimension 4 over Finite Field of size 2
```

5.8 Homsets of finitely presented graded modules

AUTHORS:

- Robert R. Bruner, Michael J. Catanzaro (2012): Initial version.
- Sverre Lunoee–Nielsen and Koen van Woerden (2019-11-29): Updated the original software to Sage version 8.9.
- Sverre Lunoee–Nielsen (2020-07-01): Refactored the code and added new documentation and tests.

class sage.modules.fp_graded.homspace.**FPModuleHomspace**(*X*, *Y*, *category=None*, *base=None*, *check=True*)

Bases: Homset

Element

alias of *FPModuleMorphism*

an_element(*n=0*)

Create a homomorphism belonging to `self`.

INPUT:

- `n` – (default: 0) an integer degree

OUTPUT: a module homomorphism of degree `n`

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A = SteenrodAlgebra(2)
sage: HZ = FPModule(A, [0], relations=[[Sq(1)]])

sage: Hom(HZ, HZ).an_element(3)
Module endomorphism of Finitely presented left module on 1 generator and 1␣
↪relation over mod 2 Steenrod algebra, milnor basis
  Defn: g[0] |--> Sq(0,1)*g[0]
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A = SteenrodAlgebra(Integer(2))
>>> HZ = FPModule(A, [Integer(0)], relations=[[Sq(Integer(1))]])

>>> Hom(HZ, HZ).an_element(Integer(3))
Module endomorphism of Finitely presented left module on 1 generator and 1␣
↪relation over mod 2 Steenrod algebra, milnor basis
  Defn: g[0] |--> Sq(0,1)*g[0]
```

basis_elements(*n*)

Return a basis for the free module of degree n morphisms.

INPUT:

- n – integer degree

OUTPUT: a basis for the set of all module homomorphisms of degree n

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A = SteenrodAlgebra(2)
sage: Hko = FPModule(A, [0], relations=[[Sq(2)], [Sq(1)]])

sage: Hom(Hko, Hko).basis_elements(21)
[Module endomorphism of Finitely presented left module on 1 generator and 2␣
↪relations over mod 2 Steenrod algebra, milnor basis
   Defn: g[0] |--> (Sq(0,0,3)+Sq(0,2,0,1))*g[0],
 Module endomorphism of Finitely presented left module on 1 generator and 2␣
↪relations over mod 2 Steenrod algebra, milnor basis
   Defn: g[0] |--> Sq(8,2,1)*g[0]]
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A = SteenrodAlgebra(Integer(2))
>>> Hko = FPModule(A, [Integer(0)], relations=[[Sq(Integer(2))],␣
↪[Sq(Integer(1))]])

>>> Hom(Hko, Hko).basis_elements(Integer(21))
[Module endomorphism of Finitely presented left module on 1 generator and 2␣
↪relations over mod 2 Steenrod algebra, milnor basis
   Defn: g[0] |--> (Sq(0,0,3)+Sq(0,2,0,1))*g[0],
 Module endomorphism of Finitely presented left module on 1 generator and 2␣
↪relations over mod 2 Steenrod algebra, milnor basis
   Defn: g[0] |--> Sq(8,2,1)*g[0]]
```

identity()

Return the identity homomorphism.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A2 = SteenrodAlgebra(2, profile=(3,2,1))
sage: L = FPModule(A2, [2,3], [[Sq(2),Sq(1)], [0,Sq(2)]])

sage: one = Hom(L, L).identity(); one
Module endomorphism of Finitely presented left module on 2 generators and 2␣
↪relations over sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis,␣
↪profile function [3, 2, 1]
  Defn: g[2] |--> g[2]
        g[3] |--> g[3]

sage: e = L.an_element(5)
sage: e == one(e)
True
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A2 = SteenrodAlgebra(Integer(2), profile=(Integer(3),Integer(2),
↪Integer(1)))
>>> L = FPModule(A2, [Integer(2),Integer(3)], [[Sq(Integer(2)),
↪Sq(Integer(1))], [Integer(0),Sq(Integer(2))]])

>>> one = Hom(L, L).identity(); one
Module endomorphism of Finitely presented left module on 2 generators and 2␣
↪relations over sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis,␣
↪profile function [3, 2, 1]
  Defn: g[2] |--> g[2]
        g[3] |--> g[3]

>>> e = L.an_element(Integer(5))
>>> e == one(e)
True
```

It is an error to call this function when the homset is not a set of endomorphisms:

```
sage: F = FPModule(A2, [1,3])
sage: Hom(F,L).identity()
Traceback (most recent call last):
...
TypeError: this homspace does not consist of endomorphisms
```

```
>>> from sage.all import *
>>> F = FPModule(A2, [Integer(1),Integer(3)])
>>> Hom(F,L).identity()
Traceback (most recent call last):
...
TypeError: this homspace does not consist of endomorphisms
```

An example with free graded modules:

```
sage: A2 = SteenrodAlgebra(2, profile=(3,2,1))
sage: L = A2.free_graded_module((2,3))
sage: H = Hom(L, L)
sage: H.identity()
Module endomorphism of Free graded left module on 2 generators
 over sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile␣
↪function [3, 2, 1]
  Defn: g[2] |--> g[2]
        g[3] |--> g[3]
```

```
>>> from sage.all import *
>>> A2 = SteenrodAlgebra(Integer(2), profile=(Integer(3),Integer(2),
↪Integer(1)))
>>> L = A2.free_graded_module((Integer(2),Integer(3)))
>>> H = Hom(L, L)
>>> H.identity()
Module endomorphism of Free graded left module on 2 generators
 over sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile␣
```

(continues on next page)

(continued from previous page)

```
↪function [3, 2, 1]
  Defn: g[2] |--> g[2]
        g[3] |--> g[3]
```

zero()

Create the trivial homomorphism in `self`.

EXAMPLES:

```
sage: from sage.modules.fp_graded.module import FPModule
sage: A2 = SteenrodAlgebra(2, profile=(3,2,1))
sage: F = FPModule(A2, [1,3])
sage: L = FPModule(A2, [2,3], [[Sq(2),Sq(1)], [0,Sq(2)]])

sage: z = Hom(F, L).zero()
sage: z(F.an_element(5))
0
sage: z(F.an_element(23))
0
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.module import FPModule
>>> A2 = SteenrodAlgebra(Integer(2), profile=(Integer(3),Integer(2),
↪Integer(1)))
>>> F = FPModule(A2, [Integer(1),Integer(3)])
>>> L = FPModule(A2, [Integer(2),Integer(3)], [[Sq(Integer(2)),
↪Sq(Integer(1))], [Integer(0),Sq(Integer(2))]])

>>> z = Hom(F, L).zero()
>>> z(F.an_element(Integer(5)))
0
>>> z(F.an_element(Integer(23)))
0
```

Example with free modules:

```
sage: A2 = SteenrodAlgebra(2, profile=(3,2,1))
sage: F = A2.free_graded_module((1,3))
sage: L = A2.free_graded_module((2,3))
sage: H = Hom(F, L)
sage: H.zero()
Module morphism:
  From: Free graded left module on 2 generators over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
  To:   Free graded left module on 2 generators over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
  Defn: g[1] |--> 0
        g[3] |--> 0
```

```
>>> from sage.all import *
>>> A2 = SteenrodAlgebra(Integer(2), profile=(Integer(3),Integer(2),
↪Integer(1)))
```

(continues on next page)

(continued from previous page)

```
>>> F = A2.free_graded_module((Integer(1),Integer(3)))
>>> L = A2.free_graded_module((Integer(2),Integer(3)))
>>> H = Hom(F, L)
>>> H.zero()
Module morphism:
  From: Free graded left module on 2 generators over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
  To:   Free graded left module on 2 generators over sub-Hopf algebra of mod␣
↪2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
  Defn: g[1] |--> 0
        g[3] |--> 0
```

5.9 Finitely presented graded modules over the Steenrod algebra

This package allows the user to define finitely presented modules over the mod p Steenrod algebra, elements of them, and morphisms between them. Methods are provided for doing basic homological algebra, e.g. computing kernels and images of homomorphisms, and finding free resolutions of modules.

Theoretical background

The category of finitely presented graded modules over an arbitrary non-Noetherian graded ring R is not abelian in general, since kernels of homomorphisms are not necessarily finitely presented.

The mod p Steenrod algebra is non-Noetherian, but it is the union of a countable set of finite sub-Hopf algebras ([Mar1983] Ch. 15, Sect. 1, Prop 7). It is therefore an example of a P-algebra ([Mar1983] Ch. 13).

Any finitely presented module over the Steenrod algebra is defined over one of these finite sub-Hopf algebras. Similarly, any homomorphism between finitely presented modules over the Steenrod algebra is defined over a finite sub-Hopf algebra of the Steenrod algebra. Further, tensoring up from the category of modules over a sub-Hopf algebra is an exact functor, since the Steenrod algebra is free over any sub-Hopf algebra.

It follows that kernels, cokernels, images, and, more generally, any finite limits or colimits can be computed in the category of modules over the Steenrod algebra by working in the category of modules over an appropriate finite sub-Hopf algebra.

It is also the case that presentations of modules and the images of the generators of the domain of a homomorphism are the same over the sub-Hopf algebra and over the whole Steenrod algebra, so that the tensoring up is entirely implicit and requires no computation.

The definitions and computations carried out by this package are thus done as follows. First, find a small finite sub-Hopf algebra over which the computation can be done. Then, carry out the calculation there, where it is a finite problem and can be reduced to linear algebra over a finite prime field.

For examples, see the Steenrod algebra modules thematic tutorial.

AUTHORS:

- Robert R. Bruner, Michael J. Catanzaro (2012): Initial version.
- Sverre Lunoee–Nielsen and Koen van Woerden (2019-11-29): Updated the original software to Sage version 8.9.
- Sverre Lunoee–Nielsen (2020-07-01): Refactored the code and added new documentation and tests.

class sage.modules.fp_graded.steenrod.module.**SteenrodFPModule**(*j*, *names*)

Bases: *FPModule*, *SteenrodModuleMixin*

Create a finitely presented module over the Steenrod algebra.

See also

The thematic tutorial on Steenrod algebra modules.

INPUT:

One of the following:

- `arg0` – a morphism such that the module is the cokernel, or a free graded module, in which case the output is the same module, viewed as finitely presented

Otherwise:

- `arg0` – the graded connected algebra over which the module is defined; this algebra must be equipped with a graded basis
- `generator_degrees` – tuple of integer degrees
- `relations` – tuple of relations; a relation is a tuple of coefficients $(c_1, \ldots, c_n)$, ordered so that they correspond to the module generators, that is, such a tuple corresponds to the relation

$$c_1 g_1 + \ldots + c_n g_n = 0$$

if the generators are $(g_1, \ldots, g_n)$

resolution(*k*, *top_dim=None*, *verbose=False*)

A free resolution of `self` of the given length.

INPUT:

- `k` – nonnegative integer
- `top_dim` – (optional) stop the computation at this degree
- `verbose` – boolean (default: `False`); whether log messages are printed

OUTPUT: list of homomorphisms $[\epsilon, f_1, \ldots, f_k]$ such that

$$f_i : F_i \to F_{i-1} \text{ for } 1 \leq i \leq k,$$
$$\epsilon : F_0 \to M,$$

where each F_i is a finitely generated free module, and the sequence

$$F_k \xrightarrow{f_k} F_{k-1} \xrightarrow{f_{k-1}} \ldots \to F_0 \xrightarrow{\epsilon} M \to 0$$

is exact. Note that the 0th element in this list is the map ϵ, and the rest of the maps are between free modules.

EXAMPLES:

```
sage: from sage.modules.fp_graded.steenrod.module import SteenrodFPModule
sage: A = SteenrodAlgebra(2)
sage: Hko = SteenrodFPModule(A, [0], [[Sq(1)], [Sq(2)]])

sage: res = Hko.resolution(5, verbose=True)
Computing f_1 (1/5)
Computing f_2 (2/5)
Computing using the profile:
(2, 1)
Resolving the kernel in the range of dimensions [1, 8]: 1 2 3 4 5 6 7 8.
```

(continues on next page)

(continued from previous page)

```
Computing f_3 (3/5)
Computing using the profile:
(2, 1)
Resolving the kernel in the range of dimensions [2, 10]: 2 3 4 5 6 7 8 9 10.
Computing f_4 (4/5)
Computing using the profile:
(2, 1)
Resolving the kernel in the range of dimensions [3, 13]: 3 4 5 6 7 8 9 10 11␣
↪12 13.
Computing f_5 (5/5)
Computing using the profile:
(2, 1)
Resolving the kernel in the range of dimensions [4, 18]: 4 5 6 7 8 9 10 11 12␣
↪13 14 15 16 17 18.
sage: [x.domain() for x in res]
[Free graded left module on 1 generator over mod 2 Steenrod algebra, milnor␣
↪basis,
 Free graded left module on 2 generators over mod 2 Steenrod algebra, milnor␣
↪basis,
 Free graded left module on 2 generators over mod 2 Steenrod algebra, milnor␣
↪basis,
 Free graded left module on 2 generators over mod 2 Steenrod algebra, milnor␣
↪basis,
 Free graded left module on 3 generators over mod 2 Steenrod algebra, milnor␣
↪basis,
 Free graded left module on 4 generators over mod 2 Steenrod algebra, milnor␣
↪basis]
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.steenrod.module import SteenrodFPModule
>>> A = SteenrodAlgebra(Integer(2))
>>> Hko = SteenrodFPModule(A, [Integer(0)], [[Sq(Integer(1))],␣
↪[Sq(Integer(2))]])

>>> res = Hko.resolution(Integer(5), verbose=True)
Computing f_1 (1/5)
Computing f_2 (2/5)
Computing using the profile:
(2, 1)
Resolving the kernel in the range of dimensions [1, 8]: 1 2 3 4 5 6 7 8.
Computing f_3 (3/5)
Computing using the profile:
(2, 1)
Resolving the kernel in the range of dimensions [2, 10]: 2 3 4 5 6 7 8 9 10.
Computing f_4 (4/5)
Computing using the profile:
(2, 1)
Resolving the kernel in the range of dimensions [3, 13]: 3 4 5 6 7 8 9 10 11␣
↪12 13.
Computing f_5 (5/5)
Computing using the profile:
(2, 1)
```

(continues on next page)

(continued from previous page)

```
Resolving the kernel in the range of dimensions [4, 18]: 4 5 6 7 8 9 10 11 12␣
↪13 14 15 16 17 18.
>>> [x.domain() for x in res]
[Free graded left module on 1 generator over mod 2 Steenrod algebra, milnor␣
↪basis,
 Free graded left module on 2 generators over mod 2 Steenrod algebra, milnor␣
↪basis,
 Free graded left module on 2 generators over mod 2 Steenrod algebra, milnor␣
↪basis,
 Free graded left module on 2 generators over mod 2 Steenrod algebra, milnor␣
↪basis,
 Free graded left module on 3 generators over mod 2 Steenrod algebra, milnor␣
↪basis,
 Free graded left module on 4 generators over mod 2 Steenrod algebra, milnor␣
↪basis]
```

When there are no relations, the resolution is trivial:

```
sage: M = SteenrodFPModule(A, [0])
sage: M.resolution(4)
[Module endomorphism of Free graded left module on 1 generator over mod 2␣
↪Steenrod algebra, milnor basis
   Defn: g[0] |--> g[0],
 Module morphism:
   From: Free graded left module on 0 generators over mod 2 Steenrod algebra,␣
↪milnor basis
   To:   Free graded left module on 1 generator over mod 2 Steenrod algebra,␣
↪milnor basis,
 Module endomorphism of Free graded left module on 0 generators over mod 2␣
↪Steenrod algebra, milnor basis,
 Module endomorphism of Free graded left module on 0 generators over mod 2␣
↪Steenrod algebra, milnor basis,
 Module endomorphism of Free graded left module on 0 generators over mod 2␣
↪Steenrod algebra, milnor basis]
```

```
>>> from sage.all import *
>>> M = SteenrodFPModule(A, [Integer(0)])
>>> M.resolution(Integer(4))
[Module endomorphism of Free graded left module on 1 generator over mod 2␣
↪Steenrod algebra, milnor basis
   Defn: g[0] |--> g[0],
 Module morphism:
   From: Free graded left module on 0 generators over mod 2 Steenrod algebra,␣
↪milnor basis
   To:   Free graded left module on 1 generator over mod 2 Steenrod algebra,␣
↪milnor basis,
 Module endomorphism of Free graded left module on 0 generators over mod 2␣
↪Steenrod algebra, milnor basis,
 Module endomorphism of Free graded left module on 0 generators over mod 2␣
↪Steenrod algebra, milnor basis,
 Module endomorphism of Free graded left module on 0 generators over mod 2␣
↪Steenrod algebra, milnor basis]
```

class sage.modules.fp_graded.steenrod.module.**SteenrodFreeModule**(*algebra*, *generator_degrees*, *category*, *names=None*, ***kwds*)

Bases: *FreeGradedModule*, *SteenrodModuleMixin*

class sage.modules.fp_graded.steenrod.module.**SteenrodModuleMixin**

Bases: object

Mixin class for common methods of the Steenrod algebra modules.

export_module_definition(*powers_of_two_only=True*)

Return the module to the input format used by R. Bruner's Ext software as a string.

INPUT:

- powers_of_two_only – boolean (default: True); if the output should contain the action of all Steenrod squaring operations (restricted by the profile), or only the action of the operations of degree equal to a power of two

EXAMPLES:

```
sage: from sage.modules.fp_graded.steenrod.module import SteenrodFPModule
sage: A1 = algebra=SteenrodAlgebra(p=2, profile=[2,1])
sage: M = SteenrodFPModule(A1, [0])
sage: print(M.export_module_definition())
8 0 1 2 3 3 4 5 6
0 1 1 1
2 1 1 4
3 1 1 5
6 1 1 7
0 2 1 2
1 2 2 3 4
2 2 1 5
3 2 1 6
4 2 1 6
5 2 1 7
sage: N = SteenrodFPModule(A1, [0], [[Sq(1)]])
sage: print(N.export_module_definition())
4 0 2 3 5
1 1 1 2
0 2 1 1
2 2 1 3
sage: print(N.export_module_definition(powers_of_two_only=False))
4 0 2 3 5
1 1 1 2
0 2 1 1
2 2 1 3
0 3 1 2
sage: A2 = SteenrodAlgebra(p=2, profile=[3,2,1])
sage: Hko = SteenrodFPModule(A2, [0], [[Sq(1)], [Sq(2)]])
sage: print(Hko.export_module_definition())
8 0 4 6 7 10 11 13 17
2 1 1 3
4 1 1 5
1 2 1 2
5 2 1 6
```

(continues on next page)

(continued from previous page)

```
0 4 1 1
2 4 1 4
3 4 1 5
6 4 1 7
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.steenrod.module import SteenrodFPModule
>>> A1 = algebra=SteenrodAlgebra(p=Integer(2), profile=[Integer(2),
↪Integer(1)])
>>> M = SteenrodFPModule(A1, [Integer(0)])
>>> print(M.export_module_definition())
8 0 1 2 3 3 4 5 6
0 1 1 1
2 1 1 4
3 1 1 5
6 1 1 7
0 2 1 2
1 2 2 3 4
2 2 1 5
3 2 1 6
4 2 1 6
5 2 1 7
>>> N = SteenrodFPModule(A1, [Integer(0)], [[Sq(Integer(1))]])
>>> print(N.export_module_definition())
4 0 2 3 5
1 1 1 2
0 2 1 1
2 2 1 3
>>> print(N.export_module_definition(powers_of_two_only=False))
4 0 2 3 5
1 1 1 2
0 2 1 1
2 2 1 3
0 3 1 2
>>> A2 = SteenrodAlgebra(p=Integer(2), profile=[Integer(3),Integer(2),
↪Integer(1)])
>>> Hko = SteenrodFPModule(A2, [Integer(0)], [[Sq(Integer(1))],␣
↪[Sq(Integer(2))]])
>>> print(Hko.export_module_definition())
8 0 4 6 7 10 11 13 17
2 1 1 3
4 1 1 5
1 2 1 2
5 2 1 6
0 4 1 1
2 4 1 4
3 4 1 5
6 4 1 7
```

profile()

Return a finite profile over which `self` can be defined.

Any finitely presented module over the Steenrod algebra can be defined over a finite-dimensional sub-Hopf

algebra, and this method identifies such a sub-Hopf algebra and returns its profile function.

Note

The profile produced by this function is reasonably small but is not guaranteed to be minimal.

EXAMPLES:

```
sage: from sage.modules.fp_graded.steenrod.module import SteenrodFPModule
sage: A = SteenrodAlgebra(2)
sage: M = SteenrodFPModule(A, [0,1], [[Sq(2),Sq(1)],[0,Sq(2)],[Sq(3),0]])
sage: M.profile()
(2, 1)
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.steenrod.module import SteenrodFPModule
>>> A = SteenrodAlgebra(Integer(2))
>>> M = SteenrodFPModule(A, [Integer(0),Integer(1)], [[Sq(Integer(2)),
↪Sq(Integer(1))],[Integer(0),Sq(Integer(2))],[Sq(Integer(3)),Integer(0)]])
>>> M.profile()
(2, 1)
```

5.10 Homomorphisms of finitely presented modules over the Steenrod algebra

This class implements construction and basic manipulation of homomorphisms between *finitely presented graded modules* over the mod p Steenrod algebra.

AUTHORS:

- Robert R. Bruner, Michael J. Catanzaro (2012): Initial version.
- Sverre Lunoee–Nielsen and Koen van Woerden (2019-11-29): Updated the original software to Sage version 8.9.
- Sverre Lunoee–Nielsen (2020-07-01): Refactored the code and added new documentation and tests.

class sage.modules.fp_graded.steenrod.morphism.**SteenrodFPModuleMorphism**(*parent*, *values*, *check=True*)

Bases: *FPModuleMorphism*

cokernel_projection(*verbose=False*)

Compute the map to the cokernel of `self`.

INPUT:

- `verbose` – boolean (default: `False`); whether log messages are printed

OUTPUT:

The natural projection from the codomain of this homomorphism to its cokernel.

EXAMPLES:

```
sage: from sage.modules.fp_graded.steenrod.module import SteenrodFPModule
sage: A1 = SteenrodAlgebra(2, profile=(2,1))
```

(continues on next page)

(continued from previous page)

```
sage: M = SteenrodFPModule(A1, [0], [[Sq(2)]])
sage: F = SteenrodFPModule(A1, [0])

sage: r = Hom(F, M)([A1.Sq(1)*M.generator(0)])
sage: co = r.cokernel_projection(); co
Module morphism:
  From: Finitely presented left module on 1 generator and 1 relation over sub-
↪Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function [2,␣
↪1]
  To:   Finitely presented left module on 1 generator and 2 relations over␣
↪sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function␣
↪[2, 1]
  Defn: g[0] |--> g[0]

sage: co.domain().is_trivial()
False
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.steenrod.module import SteenrodFPModule
>>> A1 = SteenrodAlgebra(Integer(2), profile=(Integer(2),Integer(1)))
>>> M = SteenrodFPModule(A1, [Integer(0)], [[Sq(Integer(2))]])
>>> F = SteenrodFPModule(A1, [Integer(0)])

>>> r = Hom(F, M)([A1.Sq(Integer(1))*M.generator(Integer(0))])
>>> co = r.cokernel_projection(); co
Module morphism:
  From: Finitely presented left module on 1 generator and 1 relation over sub-
↪Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function [2,␣
↪1]
  To:   Finitely presented left module on 1 generator and 2 relations over␣
↪sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function␣
↪[2, 1]
  Defn: g[0] |--> g[0]

>>> co.domain().is_trivial()
False
```

image(*top_dim=None*, *verbose=False*)

Return the image of `self`.

INPUT:

- `top_dim` – integer (optional); used by this function to stop the computation at the given degree
- `verbose` – boolean (default: `False`); whether log messages are printed

OUTPUT:

An injective homomorphism into the codomain of `self` which is onto the image of `self`.

EXAMPLES:

```
sage: from sage.modules.fp_graded.steenrod.module import SteenrodFPModule
sage: A = SteenrodAlgebra(2)
sage: M = SteenrodFPModule(A, [0,1], [[Sq(2),Sq(1)], [0,Sq(2)]])
```

(continues on next page)

(continued from previous page)

```
sage: S = SteenrodFPModule(A, [0], [[Sq(2)]])
sage: f = Hom(S, M)([M([0,1])])
sage: f.is_injective()
True
sage: i = f.image(); i
Module morphism:
  From: Finitely presented left module on 1 generator and 1 relation over mod␣
↪2 Steenrod algebra, milnor basis
  To:   Finitely presented left module on 2 generators and 2 relations over␣
↪mod 2 Steenrod algebra, milnor basis
  Defn: g[1] |--> g[1]
sage: i.codomain() is M
True
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.steenrod.module import SteenrodFPModule
>>> A = SteenrodAlgebra(Integer(2))
>>> M = SteenrodFPModule(A, [Integer(0),Integer(1)], [[Sq(Integer(2)),
↪Sq(Integer(1))], [Integer(0),Sq(Integer(2))]])
>>> S = SteenrodFPModule(A, [Integer(0)], [[Sq(Integer(2))]])
>>> f = Hom(S, M)([M([Integer(0),Integer(1)])])
>>> f.is_injective()
True
>>> i = f.image(); i
Module morphism:
  From: Finitely presented left module on 1 generator and 1 relation over mod␣
↪2 Steenrod algebra, milnor basis
  To:   Finitely presented left module on 2 generators and 2 relations over␣
↪mod 2 Steenrod algebra, milnor basis
  Defn: g[1] |--> g[1]
>>> i.codomain() is M
True
```

Lift the map `f` over the inclusion `i`:

```
sage: f_ = f.lift(i)
sage: f_.is_injective()
True
sage: f_.is_surjective()
True

sage: g = Hom(S, M)([M([Sq(3),Sq(2)])])
sage: j = g.image(); j
Module morphism:
  From: Free graded left module on 0 generators over mod 2 Steenrod algebra,␣
↪milnor basis
  To:   Finitely presented left module on 2 generators and 2 relations over␣
↪mod 2 Steenrod algebra, milnor basis
```

```
>>> from sage.all import *
>>> f_ = f.lift(i)
>>> f_.is_injective()
```

(continues on next page)

(continued from previous page)

```
True
>>> f_.is_surjective()
True

>>> g = Hom(S, M)([M([Sq(Integer(3)),Sq(Integer(2))])])
>>> j = g.image(); j
Module morphism:
  From: Free graded left module on 0 generators over mod 2 Steenrod algebra,␣
↪milnor basis
  To:   Finitely presented left module on 2 generators and 2 relations over␣
↪mod 2 Steenrod algebra, milnor basis
```

So `g` had to be trivial:

```
sage: g.is_zero()
True
```

```
>>> from sage.all import *
>>> g.is_zero()
True
```

is_injective(*top_dim=None*, *verbose=False*)

Return `True` if `self` is injective.

INPUT:

- `top_dim` – (optional) stop the computation at this degree; if not specified, this is determined using *profile()*
- `verbose` – boolean (default: `False`); whether log messages are printed

EXAMPLES:

```
sage: from sage.modules.fp_graded.steenrod.module import SteenrodFPModule
sage: A = SteenrodAlgebra(2)
sage: M = SteenrodFPModule(A, [0,1], [[Sq(2),Sq(1)], [0,Sq(2)]])
sage: S = SteenrodFPModule(A, [0], [[Sq(2)]])
sage: f = Hom(S, M)([M([0,1])])
sage: f.is_injective()
True
sage: g = Hom(S, M).zero()
sage: g.is_injective()
False
sage: z = Hom(SteenrodFPModule(A, []), M).zero()
sage: z.is_injective()
True
sage: z.is_zero()
True
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.steenrod.module import SteenrodFPModule
>>> A = SteenrodAlgebra(Integer(2))
>>> M = SteenrodFPModule(A, [Integer(0),Integer(1)], [[Sq(Integer(2)),
↪Sq(Integer(1))], [Integer(0),Sq(Integer(2))]])
```

(continues on next page)

(continued from previous page)

```
>>> S = SteenrodFPModule(A, [Integer(0)], [[Sq(Integer(2))]])
>>> f = Hom(S, M)([M([Integer(0),Integer(1)])])
>>> f.is_injective()
True
>>> g = Hom(S, M).zero()
>>> g.is_injective()
False
>>> z = Hom(SteenrodFPModule(A, []), M).zero()
>>> z.is_injective()
True
>>> z.is_zero()
True
```

kernel_inclusion(*top_dim=None*, *verbose=False*)

Return the kernel of `self` as a morphism.

INPUT:

- `top_dim` – (optional) stop the computation at this degree; if not specified, this is determined using *profile()*
- `verbose` – boolean (default: `False`); whether log messages are printed

OUTPUT: an injective homomorphism into the domain `self` which is onto the kernel of this homomorphism.

EXAMPLES:

```
sage: from sage.modules.fp_graded.steenrod.module import SteenrodFPModule
sage: A = SteenrodAlgebra(2)
sage: M = SteenrodFPModule(A, [0,1], [[Sq(2),Sq(1)], [0,Sq(2)]])
sage: S = SteenrodFPModule(A, [0], [[Sq(2)]])
sage: f = Hom(S, M)([M([0,1])])
sage: f.is_injective()
True
sage: k = f.kernel_inclusion()
sage: k == 0
True
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.steenrod.module import SteenrodFPModule
>>> A = SteenrodAlgebra(Integer(2))
>>> M = SteenrodFPModule(A, [Integer(0),Integer(1)], [[Sq(Integer(2)),
↪Sq(Integer(1))], [Integer(0),Sq(Integer(2))]])
>>> S = SteenrodFPModule(A, [Integer(0)], [[Sq(Integer(2))]])
>>> f = Hom(S, M)([M([Integer(0),Integer(1)])])
>>> f.is_injective()
True
>>> k = f.kernel_inclusion()
>>> k == Integer(0)
True
```

Since k is both trivial and injective, its domain should be the zero module:

```
sage: k.domain().is_trivial()
True

sage: g = Hom(S, M)([M([Sq(3),Sq(2)])])
sage: h = g.kernel_inclusion()
sage: h.is_identity()
True
sage: ker = h.domain();
sage: ker is S
True
```

```
>>> from sage.all import *
>>> k.domain().is_trivial()
True

>>> g = Hom(S, M)([M([Sq(Integer(3)),Sq(Integer(2))])])
>>> h = g.kernel_inclusion()
>>> h.is_identity()
True
>>> ker = h.domain();
>>> ker is S
True
```

So g had to be trivial:

```
sage: g.is_zero()
True
```

```
>>> from sage.all import *
>>> g.is_zero()
True
```

profile()

Return a finite profile over which `self` can be defined.

This is in some ways the key method for these morphisms. As discussed in the "Theoretical background" section of *sage.modules.fp_graded.steenrod.module*, any homomorphism of finitely presented modules over the Steenrod algebra can be defined over a finite-dimensional sub-Hopf algebra, and this method identifies such a sub-Hopf algebra and returns its profile function.

EXAMPLES:

```
sage: from sage.modules.fp_graded.steenrod.module import SteenrodFPModule
sage: A = SteenrodAlgebra(2)
sage: M = SteenrodFPModule(A, [0,1], [[Sq(2),Sq(1)], [0,Sq(2)]])
sage: one = Hom(M,M).identity()
sage: one.profile()
(2, 1)
sage: zero = Hom(M,M).zero()
sage: zero.profile()
(2, 1)
sage: A_fin = SteenrodAlgebra(2, profile=(2,1))
sage: M_fin = M.change_ring(A_fin)
```

```
>>> from sage.all import *
>>> from sage.modules.fp_graded.steenrod.module import SteenrodFPModule
>>> A = SteenrodAlgebra(Integer(2))
>>> M = SteenrodFPModule(A, [Integer(0),Integer(1)], [[Sq(Integer(2)),
↪Sq(Integer(1))], [Integer(0),Sq(Integer(2))]])
>>> one = Hom(M,M).identity()
>>> one.profile()
(2, 1)
>>> zero = Hom(M,M).zero()
>>> zero.profile()
(2, 1)
>>> A_fin = SteenrodAlgebra(Integer(2), profile=(Integer(2),Integer(1)))
>>> M_fin = M.change_ring(A_fin)
```

Change the ring of the module `M`:

```
sage: M_fin.change_ring(A) is M
True
```

```
>>> from sage.all import *
>>> M_fin.change_ring(A) is M
True
```

We can change rings to the finite sub-Hopf algebra defined by the profile we just computed:

```
sage: one_fin = one.change_ring(A_fin)
sage: one_fin.domain()
Finitely presented left module on 2 generators and 2 relations over
 sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function␣
↪[2, 1]
```

```
>>> from sage.all import *
>>> one_fin = one.change_ring(A_fin)
>>> one_fin.domain()
Finitely presented left module on 2 generators and 2 relations over
 sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function␣
↪[2, 1]
```

If we change back to the full Steenrod algebra, we are back where we started:

```
sage: one_fin.change_ring(A) == one
True
```

```
>>> from sage.all import *
>>> one_fin.change_ring(A) == one
True
```

class sage.modules.fp_graded.steenrod.morphism.**SteenrodFreeModuleMorphism**(*parent*, *values*)

Bases: *FreeGradedModuleMorphism*, *SteenrodFPModuleMorphism*

CHAPTER

SIX

SPECIAL MODULES

6.1 Discrete subgroups of $\mathbf{Z}^n$

AUTHORS:

- Martin Albrecht (2014-03): initial version
- Jan Pöschko (2012-08): some code in this module was taken from Jan Pöschko's 2012 GSoC project

class sage.modules.free_module_integer.**FreeModule_submodule_with_basis_integer**(*ambient*, *basis*, *check=True*, *echelonize=False*, *echelonized_basis=None*, *already_echelonized=False*, *lll_reduce=True*)

Bases: *FreeModule_submodule_with_basis_pid*

This class represents submodules of $\mathbf{Z}^n$ with a distinguished basis.

However, most functionality in excess of standard submodules over PID is for these submodules considered as discrete subgroups of $\mathbf{Z}^n$, i.e. as lattices. That is, this class provides functions for computing LLL and BKZ reduced bases for this free module with respect to the standard Euclidean norm.

EXAMPLES:

```
sage: from sage.modules.free_module_integer import IntegerLattice
sage: L = IntegerLattice(sage.crypto.gen_lattice(type='modular', m=10,
....:                                            seed=1337, dual=True)); L
Free module of degree 10 and rank 10 over Integer Ring
User basis matrix:
[-1  1  2 -2  0  1  0 -1  2  1]
[ 1  0  0 -1 -2  1 -2  3 -1  0]
[ 1  2  0  2 -1  1 -2  2  2  0]
[ 1  0 -1  0  2  3  0  0 -1 -2]
[ 1 -3  0  0  2  1 -2 -1  0  0]
[-3  0 -1  0 -1  2 -2  0  0  2]
```

(continues on next page)

(continued from previous page)

```
[ 0  0  0  1  0  2 -3 -3 -2 -1]
[ 0 -1 -4 -1 -1  1  2 -1  0  1]
[ 1  1 -2  1  1  2  1  1 -2  3]
[ 2 -1  1  2 -3  2  2  1  0  1]
sage: L.shortest_vector()
(-1, 1, 2, -2, 0, 1, 0, -1, 2, 1)
```

```
>>> from sage.all import *
>>> from sage.modules.free_module_integer import IntegerLattice
>>> L = IntegerLattice(sage.crypto.gen_lattice(type='modular', m=Integer(10),
...                                            seed=Integer(1337), dual=True)); L
Free module of degree 10 and rank 10 over Integer Ring
User basis matrix:
[-1  1  2 -2  0  1  0 -1  2  1]
[ 1  0  0 -1 -2  1 -2  3 -1  0]
[ 1  2  0  2 -1  1 -2  2  2  0]
[ 1  0 -1  0  2  3  0  0 -1 -2]
[ 1 -3  0  0  2  1 -2 -1  0  0]
[-3  0 -1  0 -1  2 -2  0  0  2]
[ 0  0  0  1  0  2 -3 -3 -2 -1]
[ 0 -1 -4 -1 -1  1  2 -1  0  1]
[ 1  1 -2  1  1  2  1  1 -2  3]
[ 2 -1  1  2 -3  2  2  1  0  1]
>>> L.shortest_vector()
(-1, 1, 2, -2, 0, 1, 0, -1, 2, 1)
```

BKZ(**args*, ***kwds*)

Return a Block Korkine-Zolotareff reduced basis for `self`.

INPUT:

- `*args` – passed through to `sage.matrix.matrix_integer_dense.Matrix_integer_dense.BKZ()`
- `*kwds` – passed through to `sage.matrix.matrix_integer_dense.Matrix_integer_dense.BKZ()`

OUTPUT: integer matrix which is a BKZ-reduced basis for this lattice

EXAMPLES:

```
sage: # needs sage.libs.linbox (o/w timeout)
sage: from sage.modules.free_module_integer import IntegerLattice
sage: A = sage.crypto.gen_lattice(type='random', n=1, m=60, q=2^60, seed=42)
sage: L = IntegerLattice(A, lll_reduce=False)
sage: min(v.norm().n() for v in L.reduced_basis)
4.17330740711759e15
sage: L.LLL()
60 x 60 dense matrix over Integer Ring (use the '.str()' method to see the␣
↪entries)
sage: min(v.norm().n() for v in L.reduced_basis)
5.19615242270663
sage: L.BKZ(block_size=10)
60 x 60 dense matrix over Integer Ring (use the '.str()' method to see the␣
```

(continues on next page)

(continued from previous page)

```
↪entries)
sage: min(v.norm().n() for v in L.reduced_basis)
4.12310562561766
```

```
>>> from sage.all import *
>>> # needs sage.libs.linbox (o/w timeout)
>>> from sage.modules.free_module_integer import IntegerLattice
>>> A = sage.crypto.gen_lattice(type='random', n=Integer(1), m=Integer(60),␣
↪q=Integer(2)**Integer(60), seed=Integer(42))
>>> L = IntegerLattice(A, lll_reduce=False)
>>> min(v.norm().n() for v in L.reduced_basis)
4.17330740711759e15
>>> L.LLL()
60 x 60 dense matrix over Integer Ring (use the '.str()' method to see the␣
↪entries)
>>> min(v.norm().n() for v in L.reduced_basis)
5.19615242270663
>>> L.BKZ(block_size=Integer(10))
60 x 60 dense matrix over Integer Ring (use the '.str()' method to see the␣
↪entries)
>>> min(v.norm().n() for v in L.reduced_basis)
4.12310562561766
```

Note

If `block_size == L.rank()` where `L` is this lattice, then this function performs Hermite-Korkine-Zolotareff (HKZ) reduction.

HKZ(**args*, ***kwds*)

Hermite-Korkine-Zolotarev (HKZ) reduce the basis.

A basis B of a lattice L, with orthogonalized basis B^* such that $B = M \cdot B^*$ is HKZ reduced, if and only if, the following properties are satisfied:

1. The basis B is size-reduced, i.e., all off-diagonal coefficients of M satisfy $|\mu_{i,j}| \leq 1/2$
2. The vector b_1 realizes the first minimum $\lambda_1(L)$.
3. The projection of the vectors $b_2, \ldots, b_r$ orthogonally to b_1 form an HKZ reduced basis.

Note

This is realized by calling *sage.modules.free_module_integer.FreeModule_submodule_with_basis_integer.BKZ()* with `block_size == self.rank()`.

INPUT:

- `*args` – passed through to *BKZ()*
- `*kwds` – passed through to *BKZ()*

OUTPUT: integer matrix which is a HKZ-reduced basis for this lattice

EXAMPLES:

```
sage: from sage.modules.free_module_integer import IntegerLattice
sage: L = sage.crypto.gen_lattice(type='random', n=1, m=40, q=2^60, seed=1337,
↪ lattice=True)
sage: L.HKZ()
40 x 40 dense matrix over Integer Ring (use the '.str()' method to see the␣
↪entries)

sage: L.reduced_basis[0]
(0, 0, -1, -1, 0, 0, -1, 1, 0, 0, -1, 1, 1, 0, 0, 1, 1, 1, -1, 0, 0, 1, -1, 0,
↪ 0, -1, 0, 0, 1, 0, 0, -1, 0, 0, 0, 1, 1, 0, 0, -2)
```

```
>>> from sage.all import *
>>> from sage.modules.free_module_integer import IntegerLattice
>>> L = sage.crypto.gen_lattice(type='random', n=Integer(1), m=Integer(40),␣
↪q=Integer(2)**Integer(60), seed=Integer(1337), lattice=True)
>>> L.HKZ()
40 x 40 dense matrix over Integer Ring (use the '.str()' method to see the␣
↪entries)

>>> L.reduced_basis[Integer(0)]
(0, 0, -1, -1, 0, 0, -1, 1, 0, 0, -1, 1, 1, 0, 0, 1, 1, 1, -1, 0, 0, 1, -1, 0,
↪ 0, -1, 0, 0, 1, 0, 0, -1, 0, 0, 0, 1, 1, 0, 0, -2)
```

LLL(**args*, ***kwds*)

Return an LLL reduced basis for `self`.

A lattice basis $(b_1, b_2, ..., b_d)$ is (δ, η)-LLL-reduced if the two following conditions hold:

- For any $i > j$, we have $|\mu_{i,j}| \leq \eta$.
- For any $i < d$, we have $\delta |b_i^*|^2 \leq |b_{i+1}^* + \mu_{i+1,i} b_i^*|^2$,

where $\mu_{i,j} = \langle b_i, b_j^* \rangle / \langle b_j^*, b_j^* \rangle$ and b_i^* is the i-th vector of the Gram-Schmidt orthogonalisation of $(b_1, b_2, \ldots, b_d)$.

The default reduction parameters are $\delta = 0.99$ and $\eta = 0.501$.

The parameters δ and η must satisfy: $0.25 < \delta \leq 1.0$ and $0.5 \leq \eta < \sqrt{\delta}$. Polynomial time complexity is only guaranteed for $\delta < 1$. Not every algorithm admits the case $\delta = 1$.

INPUT:

- `*args` – passed through to `sage.matrix.matrix_integer_dense.Matrix_integer_dense.LLL()`
- `**kwds` – passed through to `sage.matrix.matrix_integer_dense.Matrix_integer_dense.LLL()`

OUTPUT: integer matrix which is an LLL-reduced basis for this lattice

EXAMPLES:

```
sage: from sage.modules.free_module_integer import IntegerLattice
sage: A = random_matrix(ZZ, 10, 10, x=-2000, y=2000)
sage: while A.rank() < 10:
....:     A = random_matrix(ZZ, 10, 10)
sage: L = IntegerLattice(A, lll_reduce=False); L
Free module of degree 10 and rank 10 over Integer Ring
```

(continues on next page)

(continued from previous page)

```
User basis matrix:
...
sage: L.reduced_basis == A
True
sage: old = L.reduced_basis[0].norm().n()                                   #␣
↪needs sage.symbolic
sage: _ = L.LLL()
sage: new = L.reduced_basis[0].norm().n()                                   #␣
↪needs sage.symbolic
sage: new <= old                                                            #␣
↪needs sage.symbolic
True
```

```
>>> from sage.all import *
>>> from sage.modules.free_module_integer import IntegerLattice
>>> A = random_matrix(ZZ, Integer(10), Integer(10), x=-Integer(2000),␣
↪y=Integer(2000))
>>> while A.rank() < Integer(10):
...     A = random_matrix(ZZ, Integer(10), Integer(10))
>>> L = IntegerLattice(A, lll_reduce=False); L
Free module of degree 10 and rank 10 over Integer Ring
User basis matrix:
...
>>> L.reduced_basis == A
True
>>> old = L.reduced_basis[Integer(0)].norm().n()                                ␣
↪    # needs sage.symbolic
>>> _ = L.LLL()
>>> new = L.reduced_basis[Integer(0)].norm().n()                                ␣
↪    # needs sage.symbolic
>>> new <= old                                                               #␣
↪needs sage.symbolic
True
```

approximate_closest_vector(*t*, *delta=None*, *algorithm='embedding'*, **args*, ***kwargs*)

Compute a vector w in this lattice which is close to the target vector t. The ratio $\frac{|t-w|}{|t-u|}$, where u is the closest lattice vector to t, is exponential in the dimension of the lattice.

This will check whether the basis is already δ-LLL-reduced and otherwise it will run LLL to make sure that it is. For more information about `delta` see *LLL()*.

INPUT:

- `t` – the target vector to compute a close vector to
- `delta` – (default: `0.99`) the LLL reduction parameter
- `algorithm` – string (default: 'embedding'):
  - `'embedding'` – embeds the lattice in a d+1 dimensional space and seeks short vectors using LLL. This calls LLL twice but is usually still quick.
  - `'nearest_plane'` – uses the "NEAREST PLANE" algorithm from [Bab86]
  - `'rounding_off'` – uses the "ROUNDING OFF" algorithm from [Bab86]. This yields slightly worse results than the other algorithms but is at least faster than `'nearest_plane'`.

- `*args` – passed through to *LLL()*
- `**kwds` – passed through to *LLL()*

OUTPUT: the vector w described above

EXAMPLES:

```
sage: from sage.modules.free_module_integer import IntegerLattice
sage: L = IntegerLattice([[1, 0], [0, 1]])
sage: L.approximate_closest_vector((-6, 5/3))
(-6, 2)
```

```
>>> from sage.all import *
>>> from sage.modules.free_module_integer import IntegerLattice
>>> L = IntegerLattice([[Integer(1), Integer(0)], [Integer(0), Integer(1)]])
>>> L.approximate_closest_vector((-Integer(6), Integer(5)/Integer(3)))
(-6, 2)
```

The quality of the approximation depends on `delta`:

```
sage: from sage.modules.free_module_integer import IntegerLattice
sage: L = IntegerLattice([[101, 0, 0, 0], [0, 101, 0, 0],
....:                     [0, 0, 101, 0], [-28, 39, 45, 1]], lll_reduce=False)
sage: t = vector([1337]*4)
sage: L.approximate_closest_vector(t, delta=0.26)
(1331, 1324, 1349, 1334)
sage: L.approximate_closest_vector(t, delta=0.99)
(1326, 1349, 1339, 1345)
sage: L.closest_vector(t)
(1326, 1349, 1339, 1345)

sage: # Checking that the other algorithms work
sage: L.approximate_closest_vector(t, algorithm='nearest_plane')
(1326, 1349, 1339, 1345)
sage: L.approximate_closest_vector(t, algorithm='rounding_off')
(1331, 1324, 1349, 1334)
```

```
>>> from sage.all import *
>>> from sage.modules.free_module_integer import IntegerLattice
>>> L = IntegerLattice([[Integer(101), Integer(0), Integer(0), Integer(0)],␣
↪[Integer(0), Integer(101), Integer(0), Integer(0)],
...                     [Integer(0), Integer(0), Integer(101), Integer(0)], [-
↪Integer(28), Integer(39), Integer(45), Integer(1)]], lll_reduce=False)
>>> t = vector([Integer(1337)]*Integer(4))
>>> L.approximate_closest_vector(t, delta=RealNumber('0.26'))
(1331, 1324, 1349, 1334)
>>> L.approximate_closest_vector(t, delta=RealNumber('0.99'))
(1326, 1349, 1339, 1345)
>>> L.closest_vector(t)
(1326, 1349, 1339, 1345)

>>> # Checking that the other algorithms work
>>> L.approximate_closest_vector(t, algorithm='nearest_plane')
(1326, 1349, 1339, 1345)
```

(continues on next page)

(continued from previous page)

```
>>> L.approximate_closest_vector(t, algorithm='rounding_off')
(1331, 1324, 1349, 1334)
```

babai(**args*, ***kwargs*)

Alias for *approximate_closest_vector()*.

closest_vector(*t*)

Compute the closest vector in the embedded lattice to a given vector.

INPUT:

- `t` – the target vector to compute the closest vector to

OUTPUT: the vector in the lattice closest to `t`

EXAMPLES:

```
sage: from sage.modules.free_module_integer import IntegerLattice
sage: L = IntegerLattice([[1, 0], [0, 1]])
sage: L.closest_vector((-6, 5/3))
(-6, 2)
```

```
>>> from sage.all import *
>>> from sage.modules.free_module_integer import IntegerLattice
>>> L = IntegerLattice([[Integer(1), Integer(0)], [Integer(0), Integer(1)]])
>>> L.closest_vector((-Integer(6), Integer(5)/Integer(3)))
(-6, 2)
```

ALGORITHM:

Uses the algorithm from [MV2010].

discriminant()

Return $|\det(G)|$, i.e. the absolute value of the determinant of the Gram matrix $B \cdot B^T$ for any basis B.

OUTPUT: integer

EXAMPLES:

```
sage: L = sage.crypto.gen_lattice(m=10, seed=1337, lattice=True)
sage: L.discriminant()
214358881
```

```
>>> from sage.all import *
>>> L = sage.crypto.gen_lattice(m=Integer(10), seed=Integer(1337),␣
↪lattice=True)
>>> L.discriminant()
214358881
```

is_unimodular()

Return `True` if this lattice is unimodular.

OUTPUT: boolean

EXAMPLES:

```
sage: from sage.modules.free_module_integer import IntegerLattice
sage: L = IntegerLattice([[1, 0], [0, 1]])
sage: L.is_unimodular()
True
sage: IntegerLattice([[2, 0], [0, 3]]).is_unimodular()
False
```

```
>>> from sage.all import *
>>> from sage.modules.free_module_integer import IntegerLattice
>>> L = IntegerLattice([[Integer(1), Integer(0)], [Integer(0), Integer(1)]])
>>> L.is_unimodular()
True
>>> IntegerLattice([[Integer(2), Integer(0)], [Integer(0), Integer(3)]]).is_
↪unimodular()
False
```

property reduced_basis

This attribute caches the currently best known reduced basis for `self`, where "best" is defined by the Euclidean norm of the first row vector.

EXAMPLES:

```
sage: from sage.modules.free_module_integer import IntegerLattice
sage: M = random_matrix(ZZ, 10, 10)
sage: while M.rank() < 10:
....:     M = random_matrix(ZZ, 10, 10)
sage: L = IntegerLattice(M, lll_reduce=False)
sage: L.reduced_basis == M
True

sage: LLL = L.LLL()
sage: LLL == L.reduced_basis or bool(LLL[0].norm() >= M[0].norm())
True
```

```
>>> from sage.all import *
>>> from sage.modules.free_module_integer import IntegerLattice
>>> M = random_matrix(ZZ, Integer(10), Integer(10))
>>> while M.rank() < Integer(10):
...     M = random_matrix(ZZ, Integer(10), Integer(10))
>>> L = IntegerLattice(M, lll_reduce=False)
>>> L.reduced_basis == M
True

>>> LLL = L.LLL()
>>> LLL == L.reduced_basis or bool(LLL[Integer(0)].norm() >= M[Integer(0)].
↪norm())
True
```

shortest_vector(*update_reduced_basis=True*, *algorithm='fplll'*, **args*, ***kwds*)

Return a shortest vector.

INPUT:

- `update_reduced_basis` – boolean (default: `True`); set this flag if the found vector should be used to

improve the basis

- `algorithm` – (default: `'fplll'`) either `'fplll'` or `'pari'`
- `*args` – passed through to underlying implementation
- `**kwds` – passed through to underlying implementation

OUTPUT: a shortest nonzero vector for this lattice

EXAMPLES:

```
sage: from sage.modules.free_module_integer import IntegerLattice
sage: A = sage.crypto.gen_lattice(type='random', n=1, m=30, q=2^40, seed=42)
sage: L = IntegerLattice(A, lll_reduce=False)
sage: min(v.norm().n() for v in L.reduced_basis)                                #␣
↪needs sage.symbolic
6.03890756700000e10

sage: L.shortest_vector().norm().n()                                            #␣
↪needs sage.symbolic
3.74165738677394

sage: L = IntegerLattice(A, lll_reduce=False)
sage: min(v.norm().n() for v in L.reduced_basis)                                #␣
↪needs sage.symbolic
6.03890756700000e10

sage: L.shortest_vector(algorithm='pari').norm().n()                            #␣
↪needs sage.symbolic
3.74165738677394

sage: L = IntegerLattice(A, lll_reduce=True)
sage: L.shortest_vector(algorithm='pari').norm().n()                            #␣
↪needs sage.symbolic
3.74165738677394
```

```
>>> from sage.all import *
>>> from sage.modules.free_module_integer import IntegerLattice
>>> A = sage.crypto.gen_lattice(type='random', n=Integer(1), m=Integer(30),␣
↪q=Integer(2)**Integer(40), seed=Integer(42))
>>> L = IntegerLattice(A, lll_reduce=False)
>>> min(v.norm().n() for v in L.reduced_basis)                                  #␣
↪needs sage.symbolic
6.03890756700000e10

>>> L.shortest_vector().norm().n()                                              #␣
↪needs sage.symbolic
3.74165738677394

>>> L = IntegerLattice(A, lll_reduce=False)
>>> min(v.norm().n() for v in L.reduced_basis)                                  #␣
↪needs sage.symbolic
6.03890756700000e10

>>> L.shortest_vector(algorithm='pari').norm().n()                              #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.symbolic
3.74165738677394

>>> L = IntegerLattice(A, lll_reduce=True)
>>> L.shortest_vector(algorithm='pari').norm().n()                    #␣
↪needs sage.symbolic
3.74165738677394
```

update_reduced_basis(*w*)

Inject the vector `w` and run LLL to update the basis.

INPUT:

- `w` – a vector

OUTPUT: nothing is returned but the internal state is modified

EXAMPLES:

```
sage: from sage.modules.free_module_integer import IntegerLattice
sage: A = sage.crypto.gen_lattice(type='random', n=1, m=30, q=2^40, seed=42)
sage: L = IntegerLattice(A)
sage: B = L.reduced_basis
sage: v = L.shortest_vector(update_reduced_basis=False)
sage: L.update_reduced_basis(v)
sage: bool(L.reduced_basis[0].norm() < B[0].norm())
True
```

```
>>> from sage.all import *
>>> from sage.modules.free_module_integer import IntegerLattice
>>> A = sage.crypto.gen_lattice(type='random', n=Integer(1), m=Integer(30),␣
↪q=Integer(2)**Integer(40), seed=Integer(42))
>>> L = IntegerLattice(A)
>>> B = L.reduced_basis
>>> v = L.shortest_vector(update_reduced_basis=False)
>>> L.update_reduced_basis(v)
>>> bool(L.reduced_basis[Integer(0)].norm() < B[Integer(0)].norm())
True
```

volume()

Return $vol(L)$ which is $\sqrt{\det(B \cdot B^T)}$ for any basis B.

OUTPUT: integer

EXAMPLES:

```
sage: L = sage.crypto.gen_lattice(m=10, seed=1337, lattice=True)
sage: L.volume()
14641
```

```
>>> from sage.all import *
>>> L = sage.crypto.gen_lattice(m=Integer(10), seed=Integer(1337),␣
↪lattice=True)
>>> L.volume()
14641
```

voronoi_cell(*radius=None*)

Compute the Voronoi cell of a lattice, returning a Polyhedron.

INPUT:

- `radius` – (default: automatic determination) radius of ball containing considered vertices

OUTPUT: the Voronoi cell as a Polyhedron instance

The result is cached so that subsequent calls to this function return instantly.

EXAMPLES:

```
sage: from sage.modules.free_module_integer import IntegerLattice
sage: L = IntegerLattice([[1, 0], [0, 1]])
sage: V = L.voronoi_cell()
sage: V.Vrepresentation()
(A vertex at (1/2, -1/2),
 A vertex at (1/2, 1/2),
 A vertex at (-1/2, 1/2),
 A vertex at (-1/2, -1/2))
```

```
>>> from sage.all import *
>>> from sage.modules.free_module_integer import IntegerLattice
>>> L = IntegerLattice([[Integer(1), Integer(0)], [Integer(0), Integer(1)]])
>>> V = L.voronoi_cell()
>>> V.Vrepresentation()
(A vertex at (1/2, -1/2),
 A vertex at (1/2, 1/2),
 A vertex at (-1/2, 1/2),
 A vertex at (-1/2, -1/2))
```

The volume of the Voronoi cell is the square root of the discriminant of the lattice:

```
sage: L = IntegerLattice(Matrix(ZZ, 4, 4, [[0,0,1,-1], [1,-1,2,1],
....:                                      [-6,0,3,3,], [-6,-24,-6,-5]])); L
Free module of degree 4 and rank 4 over Integer Ring
User basis matrix:
[  0   0   1  -1]
[  1  -1   2   1]
[ -6   0   3   3]
[ -6 -24  -6  -5]
sage: V = L.voronoi_cell()  # long time
sage: V.volume()            # long time
678
sage: sqrt(L.discriminant())
678
```

```
>>> from sage.all import *
>>> L = IntegerLattice(Matrix(ZZ, Integer(4), Integer(4), [[Integer(0),
↪Integer(0),Integer(1),-Integer(1)], [Integer(1),-Integer(1),Integer(2),
↪Integer(1)],
...                                      [-Integer(6),Integer(0),Integer(3),
↪Integer(3),], [-Integer(6),-Integer(24),-Integer(6),-Integer(5)]])); L
Free module of degree 4 and rank 4 over Integer Ring
```

(continues on next page)

(continued from previous page)

```
User basis matrix:
[  0   0   1  -1]
[  1  -1   2   1]
[ -6   0   3   3]
[ -6 -24  -6  -5]
>>> V = L.voronoi_cell()  # long time
>>> V.volume()            # long time
678
>>> sqrt(L.discriminant())
678
```

Lattices not having full dimension are handled as well:

```
sage: L = IntegerLattice([[2, 0, 0], [0, 2, 0]])
sage: V = L.voronoi_cell()
sage: V.Hrepresentation()
(An inequality (-1, 0, 0) x + 1 >= 0,
 An inequality (0, -1, 0) x + 1 >= 0,
 An inequality (1, 0, 0) x + 1 >= 0,
 An inequality (0, 1, 0) x + 1 >= 0)
```

```
>>> from sage.all import *
>>> L = IntegerLattice([[Integer(2), Integer(0), Integer(0)], [Integer(0),␣
↪Integer(2), Integer(0)]])
>>> V = L.voronoi_cell()
>>> V.Hrepresentation()
(An inequality (-1, 0, 0) x + 1 >= 0,
 An inequality (0, -1, 0) x + 1 >= 0,
 An inequality (1, 0, 0) x + 1 >= 0,
 An inequality (0, 1, 0) x + 1 >= 0)
```

ALGORITHM:

Uses parts of the algorithm from [VB1996].

voronoi_relevant_vectors()

Compute the embedded vectors inducing the Voronoi cell.

OUTPUT: the list of Voronoi relevant vectors

EXAMPLES:

```
sage: from sage.modules.free_module_integer import IntegerLattice
sage: L = IntegerLattice([[3, 0], [4, 0]])
sage: L.voronoi_relevant_vectors()
[(-1, 0), (1, 0)]
```

```
>>> from sage.all import *
>>> from sage.modules.free_module_integer import IntegerLattice
>>> L = IntegerLattice([[Integer(3), Integer(0)], [Integer(4), Integer(0)]])
>>> L.voronoi_relevant_vectors()
[(-1, 0), (1, 0)]
```

sage.modules.free_module_integer.**IntegerLattice**(*basis*, *lll_reduce=True*)

Construct a new integer lattice from `basis`.

INPUT:

- `basis` – can be one of the following:
  - a list of vectors
  - a matrix over the integers
  - an element of an absolute order
- `lll_reduce` – boolean (default: `True`); run LLL reduction on the basis on construction

EXAMPLES:

We construct a lattice from a list of rows:

```
sage: from sage.modules.free_module_integer import IntegerLattice
sage: IntegerLattice([[1,0,3], [0,2,1], [0,2,7]])
Free module of degree 3 and rank 3 over Integer Ring
User basis matrix:
[-2  0  0]
[ 0  2  1]
[ 1 -2  2]
```

```
>>> from sage.all import *
>>> from sage.modules.free_module_integer import IntegerLattice
>>> IntegerLattice([[Integer(1),Integer(0),Integer(3)], [Integer(0),Integer(2),
↪Integer(1)], [Integer(0),Integer(2),Integer(7)]])
Free module of degree 3 and rank 3 over Integer Ring
User basis matrix:
[-2  0  0]
[ 0  2  1]
[ 1 -2  2]
```

Sage includes a generator for hard lattices from cryptography:

```
sage: from sage.modules.free_module_integer import IntegerLattice
sage: A = sage.crypto.gen_lattice(type='modular', m=10, seed=1337, dual=True)
sage: IntegerLattice(A)
Free module of degree 10 and rank 10 over Integer Ring
User basis matrix:
[-1  1  2 -2  0  1  0 -1  2  1]
[ 1  0  0 -1 -2  1 -2  3 -1  0]
[ 1  2  0  2 -1  1 -2  2  2  0]
[ 1  0 -1  0  2  3  0  0 -1 -2]
[ 1 -3  0  0  2  1 -2 -1  0  0]
[-3  0 -1  0 -1  2 -2  0  0  2]
[ 0  0  0  1  0  2 -3 -3 -2 -1]
[ 0 -1 -4 -1 -1  1  2 -1  0  1]
[ 1  1 -2  1  1  2  1  1 -2  3]
[ 2 -1  1  2 -3  2  2  1  0  1]
```

```
>>> from sage.all import *
>>> from sage.modules.free_module_integer import IntegerLattice
```

(continues on next page)

(continued from previous page)

```
>>> A = sage.crypto.gen_lattice(type='modular', m=Integer(10), seed=Integer(1337),
↪ dual=True)
>>> IntegerLattice(A)
Free module of degree 10 and rank 10 over Integer Ring
User basis matrix:
[-1  1  2 -2  0  1  0 -1  2  1]
[ 1  0  0 -1 -2  1 -2  3 -1  0]
[ 1  2  0  2 -1  1 -2  2  2  0]
[ 1  0 -1  0  2  3  0  0 -1 -2]
[ 1 -3  0  0  2  1 -2 -1  0  0]
[-3  0 -1  0 -1  2 -2  0  0  2]
[ 0  0  0  1  0  2 -3 -3 -2 -1]
[ 0 -1 -4 -1 -1  1  2 -1  0  1]
[ 1  1 -2  1  1  2  1  1 -2  3]
[ 2 -1  1  2 -3  2  2  1  0  1]
```

You can also construct the lattice directly:

```
sage: from sage.modules.free_module_integer import IntegerLattice
sage: sage.crypto.gen_lattice(type='modular', m=10, seed=1337, dual=True,␣
↪lattice=True)
Free module of degree 10 and rank 10 over Integer Ring
User basis matrix:
[-1  1  2 -2  0  1  0 -1  2  1]
[ 1  0  0 -1 -2  1 -2  3 -1  0]
[ 1  2  0  2 -1  1 -2  2  2  0]
[ 1  0 -1  0  2  3  0  0 -1 -2]
[ 1 -3  0  0  2  1 -2 -1  0  0]
[-3  0 -1  0 -1  2 -2  0  0  2]
[ 0  0  0  1  0  2 -3 -3 -2 -1]
[ 0 -1 -4 -1 -1  1  2 -1  0  1]
[ 1  1 -2  1  1  2  1  1 -2  3]
[ 2 -1  1  2 -3  2  2  1  0  1]
```

```
>>> from sage.all import *
>>> from sage.modules.free_module_integer import IntegerLattice
>>> sage.crypto.gen_lattice(type='modular', m=Integer(10), seed=Integer(1337),␣
↪dual=True, lattice=True)
Free module of degree 10 and rank 10 over Integer Ring
User basis matrix:
[-1  1  2 -2  0  1  0 -1  2  1]
[ 1  0  0 -1 -2  1 -2  3 -1  0]
[ 1  2  0  2 -1  1 -2  2  2  0]
[ 1  0 -1  0  2  3  0  0 -1 -2]
[ 1 -3  0  0  2  1 -2 -1  0  0]
[-3  0 -1  0 -1  2 -2  0  0  2]
[ 0  0  0  1  0  2 -3 -3 -2 -1]
[ 0 -1 -4 -1 -1  1  2 -1  0  1]
[ 1  1 -2  1  1  2  1  1 -2  3]
[ 2 -1  1  2 -3  2  2  1  0  1]
```

We construct an ideal lattice from an element of an absolute order:

```
sage: # needs sage.rings.number_field
sage: K.<a>  = CyclotomicField(17)
sage: O = K.ring_of_integers()
sage: f = O(-a^15 + a^13 + 4*a^12 - 12*a^11 - 256*a^10 + a^9 - a^7
....:       - 4*a^6 + a^5 + 210*a^4 + 2*a^3 - 2*a^2 + 2*a - 2)
sage: from sage.modules.free_module_integer import IntegerLattice
sage: IntegerLattice(f)
Free module of degree 16 and rank 16 over Integer Ring
User basis matrix:
[  -2    2   -2    2  210    1   -4   -1    0    1 -256  -12    4    1    0   -1]
[  33   48   44   48  256 -209   28   51   45   49   -1   35   44   48   44   48]
[   1   -1    3   -1    3  211    2   -3    0    1    2 -255  -11    5    2    1]
[-223   34   50   47  258    0   29   45   46   47    2  -11   33   48   44   48]
[ -13   31   46   42   46   -2 -225   32   48   45  256   -2   27   43   44   45]
[ -16   33   42   46  254    1  -19   32   44   45    0  -13 -225   32   48   45]
[ -15 -223   30   50  255    1  -20   32   42   47   -2  -11  -15   33   44   44]
[ -11  -11   33   48  256    3  -17 -222   32   53    1   -9  -14   35   44   48]
[ -12  -13   32   45  257    0  -16  -13   32   48   -1  -10  -14 -222   31   51]
[  -9  -13 -221   32   52    1  -11  -12   33   46  258    1  -15  -12   33   49]
[  -5   -2   -1    0 -257  -13    3    0   -1   -2   -1   -3    1   -3    1  209]
[ -15  -11  -15   33  256   -1  -17  -14 -225   33    4  -12  -13  -14   31   44]
[  11   11   11   11 -245   -3   17   10   13  220   12    5   12    9   14  -35]
[ -18  -15  -20   29  250   -3  -23  -16  -19   30   -4  -17  -17  -17 -229   28]
[ -15  -11  -15 -223  242    5  -18  -12  -16   34   -2  -11  -15  -11  -15   33]
[ 378  120   92  147  152  462  136   96   99  144  -52  412  133   91 -107  138]
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> K = CyclotomicField(Integer(17), names=('a',)); (a,) = K._first_ngens(1)
>>> O = K.ring_of_integers()
>>> f = O(-a**Integer(15) + a**Integer(13) + Integer(4)*a**Integer(12) -␣
↪Integer(12)*a**Integer(11) - Integer(256)*a**Integer(10) + a**Integer(9) -␣
↪a**Integer(7)
...       - Integer(4)*a**Integer(6) + a**Integer(5) + Integer(210)*a**Integer(4)␣
↪+ Integer(2)*a**Integer(3) - Integer(2)*a**Integer(2) + Integer(2)*a -␣
↪Integer(2))
>>> from sage.modules.free_module_integer import IntegerLattice
>>> IntegerLattice(f)
Free module of degree 16 and rank 16 over Integer Ring
User basis matrix:
[  -2    2   -2    2  210    1   -4   -1    0    1 -256  -12    4    1    0   -1]
[  33   48   44   48  256 -209   28   51   45   49   -1   35   44   48   44   48]
[   1   -1    3   -1    3  211    2   -3    0    1    2 -255  -11    5    2    1]
[-223   34   50   47  258    0   29   45   46   47    2  -11   33   48   44   48]
[ -13   31   46   42   46   -2 -225   32   48   45  256   -2   27   43   44   45]
[ -16   33   42   46  254    1  -19   32   44   45    0  -13 -225   32   48   45]
[ -15 -223   30   50  255    1  -20   32   42   47   -2  -11  -15   33   44   44]
[ -11  -11   33   48  256    3  -17 -222   32   53    1   -9  -14   35   44   48]
[ -12  -13   32   45  257    0  -16  -13   32   48   -1  -10  -14 -222   31   51]
[  -9  -13 -221   32   52    1  -11  -12   33   46  258    1  -15  -12   33   49]
[  -5   -2   -1    0 -257  -13    3    0   -1   -2   -1   -3    1   -3    1  209]
[ -15  -11  -15   33  256   -1  -17  -14 -225   33    4  -12  -13  -14   31   44]
```

(continues on next page)

(continued from previous page)

```
[  11   11   11   11 -245   -3   17   10   13  220   12    5   12    9   14  -35]
[ -18  -15  -20   29  250   -3  -23  -16  -19   30   -4  -17  -17  -17 -229   28]
[ -15  -11  -15 -223  242    5  -18  -12  -16   34   -2  -11  -15  -11  -15   33]
[ 378  120   92  147  152  462  136   96   99  144  -52  412  133   91 -107  138]
```

We construct $\mathbf{Z}^n$:

```
sage: from sage.modules.free_module_integer import IntegerLattice
sage: IntegerLattice(ZZ^10)
Free module of degree 10 and rank 10 over Integer Ring
User 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 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]
```

```
>>> from sage.all import *
>>> from sage.modules.free_module_integer import IntegerLattice
>>> IntegerLattice(ZZ**Integer(10))
Free module of degree 10 and rank 10 over Integer Ring
User 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 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]
```

Sage also interfaces with fpylll's lattice generator:

```
sage: # needs fpylll
sage: from sage.modules.free_module_integer import IntegerLattice
sage: from fpylll import IntegerMatrix
sage: A = IntegerMatrix.random(8, "simdioph", bits=20, bits2=10)
sage: A = A.to_matrix(matrix(ZZ, 8, 8))
sage: IntegerLattice(A, lll_reduce=False)
Free module of degree 8 and rank 8 over Integer Ring
User basis matrix:
[   1024  829556  161099   11567  521155  769480  639201  689979]
[      0 1048576       0       0       0       0       0       0]
[      0       0 1048576       0       0       0       0       0]
[      0       0       0 1048576       0       0       0       0]
```

(continues on next page)

(continued from previous page)

```
[      0       0       0       0 1048576       0       0       0]
[      0       0       0       0       0 1048576       0       0]
[      0       0       0       0       0       0 1048576       0]
[      0       0       0       0       0       0       0 1048576]
```

```
>>> from sage.all import *
>>> # needs fpylll
>>> from sage.modules.free_module_integer import IntegerLattice
>>> from fpylll import IntegerMatrix
>>> A = IntegerMatrix.random(Integer(8), "simdioph", bits=Integer(20),␣
↪bits2=Integer(10))
>>> A = A.to_matrix(matrix(ZZ, Integer(8), Integer(8)))
>>> IntegerLattice(A, lll_reduce=False)
Free module of degree 8 and rank 8 over Integer Ring
User basis matrix:
[   1024  829556  161099   11567  521155  769480  639201  689979]
[      0 1048576       0       0       0       0       0       0]
[      0       0 1048576       0       0       0       0       0]
[      0       0       0 1048576       0       0       0       0]
[      0       0       0       0 1048576       0       0       0]
[      0       0       0       0       0 1048576       0       0]
[      0       0       0       0       0       0 1048576       0]
[      0       0       0       0       0       0       0 1048576]
```

6.2 Free quadratic modules

Sage supports computation with free quadratic modules over an arbitrary commutative ring. Nontrivial functionality is available over $\mathbf{Z}$ and fields. All free modules over an integral domain are equipped with an embedding in an ambient vector space and an inner product, which you can specify and change.

Create the free module of rank n over an arbitrary commutative ring R using the command `FreeModule(R,n)` with a given `inner_product_matrix`.

The following example illustrates the creation of both a vector space and a free module over the integers and a submodule of it. Use the functions `FreeModule`, `span` and member functions of free modules to create free modules. "Do not use the `FreeModule_xxx` constructors directly."

EXAMPLES:

```
sage: M = Matrix(QQ, [[2,1,0], [1,2,1], [0,1,2]])
sage: V = VectorSpace(QQ, 3, inner_product_matrix=M)
sage: type(V)
<class 'sage.modules.free_quadratic_module.FreeQuadraticModule_ambient_field_with_
↪category'>
sage: V.inner_product_matrix()
[2 1 0]
[1 2 1]
[0 1 2]
sage: W = V.subspace([[1,2,7], [1,1,0]])
sage: type(W)
<class 'sage.modules.free_quadratic_module.FreeQuadraticModule_submodule_field_with_
↪category'>
```

(continues on next page)

(continued from previous page)

```
sage: W
Quadratic space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -7]
[ 0  1  7]
Inner product matrix:
[2 1 0]
[1 2 1]
[0 1 2]
sage: W.gram_matrix()
[ 100 -104]
[-104  114]
```

```
>>> from sage.all import *
>>> M = Matrix(QQ, [[Integer(2),Integer(1),Integer(0)], [Integer(1),Integer(2),
↪Integer(1)], [Integer(0),Integer(1),Integer(2)]])
>>> V = VectorSpace(QQ, Integer(3), inner_product_matrix=M)
>>> type(V)
<class 'sage.modules.free_quadratic_module.FreeQuadraticModule_ambient_field_with_
↪category'>
>>> V.inner_product_matrix()
[2 1 0]
[1 2 1]
[0 1 2]
>>> W = V.subspace([[Integer(1),Integer(2),Integer(7)], [Integer(1),Integer(1),
↪Integer(0)]])
>>> type(W)
<class 'sage.modules.free_quadratic_module.FreeQuadraticModule_submodule_field_with_
↪category'>
>>> W
Quadratic space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -7]
[ 0  1  7]
Inner product matrix:
[2 1 0]
[1 2 1]
[0 1 2]
>>> W.gram_matrix()
[ 100 -104]
[-104  114]
```

AUTHORS:

- David Kohel (2008-06): First created (based on free_module.py)

sage.modules.free_quadratic_module.**FreeQuadraticModule**(*base_ring*, *rank*, *inner_product_matrix*, *sparse=False*, *inner_product_ring=None*)

Create the free quadratic module over the given commutative ring of the given rank.

INPUT:

- `base_ring` – a commutative ring
- `rank` – nonnegative integer

- `inner_product_matrix` – the inner product matrix
- `sparse` – boolean (default: `False`)
- `inner_product_ring` – the inner product codomain ring (default: `None`)

OUTPUT:

A free quadratic module (with given inner product matrix).

Note

In Sage, it is the case that there is only one dense and one sparse free ambient quadratic module of rank n over R and given inner product matrix.

EXAMPLES:

```
sage: M2 = FreeQuadraticModule(ZZ, 2, inner_product_matrix=[1,2,3,4])
sage: M2 is FreeQuadraticModule(ZZ, 2, inner_product_matrix=[1,2,3,4])
True
sage: M2.inner_product_matrix()
[1 2]
[3 4]
sage: M3 = FreeModule(ZZ, 2, inner_product_matrix=[[1,2],[3,4]])
sage: M3 is M2
True
```

```
>>> from sage.all import *
>>> M2 = FreeQuadraticModule(ZZ, Integer(2), inner_product_matrix=[Integer(1),
↪Integer(2),Integer(3),Integer(4)])
>>> M2 is FreeQuadraticModule(ZZ, Integer(2), inner_product_matrix=[Integer(1),
↪Integer(2),Integer(3),Integer(4)])
True
>>> M2.inner_product_matrix()
[1 2]
[3 4]
>>> M3 = FreeModule(ZZ, Integer(2), inner_product_matrix=[[Integer(1),Integer(2)],
↪[Integer(3),Integer(4)]])
>>> M3 is M2
True
```

class sage.modules.free_quadratic_module.**FreeQuadraticModule_ambient**(*base_ring*, *rank*, *inner_product_matrix*, *sparse=False*)

Bases: *FreeModule_ambient*, *FreeQuadraticModule_generic*

Ambient free module over a commutative ring.

class sage.modules.free_quadratic_module.**FreeQuadraticModule_ambient_domain**(*base_ring*, *rank*, *inner_product_matrix*, *sparse=False*)

Bases: *FreeModule_ambient_domain*, *FreeQuadraticModule_ambient*

Ambient free quadratic module over an integral domain.

ambient_vector_space()

Return the ambient vector space, which is this free module tensored with its fraction field.

EXAMPLES:

```
sage: M = ZZ^3;  M.ambient_vector_space()
Vector space of dimension 3 over Rational Field
```

```
>>> from sage.all import *
>>> M = ZZ**Integer(3);  M.ambient_vector_space()
Vector space of dimension 3 over Rational Field
```

class sage.modules.free_quadratic_module.**FreeQuadraticModule_ambient_field**(*base_field*, *dimension*, *inner_product_matrix*, *sparse=False*)

Bases: *FreeModule_ambient_field*, *FreeQuadraticModule_generic_field*, *FreeQuadraticModule_ambient_pid*

Create the ambient vector space of given dimension over the given field.

INPUT:

- `base_field` – a field
- `dimension` – nonnegative integer
- `sparse` – boolean (default: `False`)

EXAMPLES:

```
sage: VectorSpace(QQ,3,inner_product_matrix=[[2,1,0],[1,2,0],[0,1,2]])
Ambient quadratic space of dimension 3 over Rational Field
Inner product matrix:
[2 1 0]
[1 2 0]
[0 1 2]
```

```
>>> from sage.all import *
>>> VectorSpace(QQ,Integer(3),inner_product_matrix=[[Integer(2),Integer(1),
↪Integer(0)],[Integer(1),Integer(2),Integer(0)],[Integer(0),Integer(1),
↪Integer(2)]])
Ambient quadratic space of dimension 3 over Rational Field
Inner product matrix:
[2 1 0]
[1 2 0]
[0 1 2]
```

class sage.modules.free_quadratic_module.**FreeQuadraticModule_ambient_pid**(*base_ring*, *rank*, *inner_product_matrix*, *sparse=False*)

Bases: *FreeModule_ambient_pid*, *FreeQuadraticModule_generic_pid*, *FreeQuadraticModule_ambient_domain*

Ambient free quadratic module over a principal ideal domain.

class sage.modules.free_quadratic_module.**FreeQuadraticModule_generic**(*base_ring*, *rank*, *degree*, *inner_product_matrix*, *sparse=False*)

Bases: *FreeModule_generic*

Base class for all free quadratic modules.

Modules are ordered by inclusion in the same ambient space.

ambient_module()

Return the ambient module associated to this module.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: M = FreeModule(R,2)
sage: M.ambient_module()
Ambient free module of rank 2 over the integral domain Multivariate␣
↪Polynomial Ring in x, y over Rational Field

sage: V = FreeModule(QQ, 4).span([[1,2,3,4], [1,0,0,0]]); V
Vector space of degree 4 and dimension 2 over Rational Field
Basis matrix:
[  1   0   0   0]
[  0   1 3/2   2]
sage: V.ambient_module()
Vector space of dimension 4 over Rational Field
```

```
>>> from sage.all import *
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> M = FreeModule(R,Integer(2))
>>> M.ambient_module()
Ambient free module of rank 2 over the integral domain Multivariate␣
↪Polynomial Ring in x, y over Rational Field

>>> V = FreeModule(QQ, Integer(4)).span([[Integer(1),Integer(2),Integer(3),
↪Integer(4)], [Integer(1),Integer(0),Integer(0),Integer(0)]]); V
Vector space of degree 4 and dimension 2 over Rational Field
Basis matrix:
[  1   0   0   0]
[  0   1 3/2   2]
>>> V.ambient_module()
Vector space of dimension 4 over Rational Field
```

determinant()

Return the determinant of this free module.

EXAMPLES:

```
sage: M = FreeModule(ZZ, 3, inner_product_matrix=1)
sage: M.determinant()
1
sage: N = M.span([[1,2,3]])
sage: N.determinant()
14
```

(continues on next page)

(continued from previous page)

```
sage: P = M.span([[1,2,3], [1,1,1]])
sage: P.determinant()
6
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ, Integer(3), inner_product_matrix=Integer(1))
>>> M.determinant()
1
>>> N = M.span([[Integer(1),Integer(2),Integer(3)]])
>>> N.determinant()
14
>>> P = M.span([[Integer(1),Integer(2),Integer(3)], [Integer(1),Integer(1),
↪Integer(1)]])
>>> P.determinant()
6
```

discriminant()

Return the discriminant of this free module.

This is defined to be $(-1)^r$ of the determinant, where $r = n/2$ (n even) or $(n-1)/2$ (n odd) for a module of rank n.

EXAMPLES:

```
sage: M = FreeModule(ZZ, 3)
sage: M.discriminant()
1
sage: N = M.span([[1,2,3]])
sage: N.discriminant()
14
sage: P = M.span([[1,2,3], [1,1,1]])
sage: P.discriminant()
6
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ, Integer(3))
>>> M.discriminant()
1
>>> N = M.span([[Integer(1),Integer(2),Integer(3)]])
>>> N.discriminant()
14
>>> P = M.span([[Integer(1),Integer(2),Integer(3)], [Integer(1),Integer(1),
↪Integer(1)]])
>>> P.discriminant()
6
```

gram_matrix()

Return the Gram matrix associated to this free module.

This is defined to be `B*A*B.transpose()`, where `A` is the inner product matrix (induced from the ambient space), and `B` the basis matrix.

EXAMPLES:

```
sage: V = VectorSpace(QQ,4)
sage: u = V([1/2,1/2,1/2,1/2])
sage: v = V([0,1,1,0])
sage: w = V([0,0,1,1])
sage: M = span([u,v,w], ZZ)
sage: M.inner_product_matrix() == V.inner_product_matrix()
True
sage: L = M.submodule_with_basis([u,v,w])
sage: L.inner_product_matrix() == M.inner_product_matrix()
True
sage: L.gram_matrix()
[1 1 1]
[1 2 1]
[1 1 2]
```

```
>>> from sage.all import *
>>> V = VectorSpace(QQ,Integer(4))
>>> u = V([Integer(1)/Integer(2),Integer(1)/Integer(2),Integer(1)/Integer(2),
↪Integer(1)/Integer(2)])
>>> v = V([Integer(0),Integer(1),Integer(1),Integer(0)])
>>> w = V([Integer(0),Integer(0),Integer(1),Integer(1)])
>>> M = span([u,v,w], ZZ)
>>> M.inner_product_matrix() == V.inner_product_matrix()
True
>>> L = M.submodule_with_basis([u,v,w])
>>> L.inner_product_matrix() == M.inner_product_matrix()
True
>>> L.gram_matrix()
[1 1 1]
[1 2 1]
[1 1 2]
```

inner_product_matrix()

Return the inner product matrix associated to this module.

By definition, this is the inner product matrix of the ambient space, hence may be of degree greater than the rank of the module.

Note

The inner product does not have to be symmetric (see examples).

Todo

Differentiate the image ring of the inner product from the base ring of the module and/or ambient space. E.g. On an integral module over ZZ the inner product pairing could naturally take values in ZZ, QQ, RR, or CC.

EXAMPLES:

```
sage: M = FreeModule(ZZ, 3)
sage: M.inner_product_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ, Integer(3))
>>> M.inner_product_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
```

The inner product does not have to be symmetric or definite:

```
sage: N = FreeModule(ZZ,2,inner_product_matrix=[[1,-1],[2,5]])
sage: N.inner_product_matrix()
[ 1 -1]
[ 2  5]
sage: u, v = N.basis()
sage: u.inner_product(v)
-1
sage: v.inner_product(u)
2
```

```
>>> from sage.all import *
>>> N = FreeModule(ZZ,Integer(2),inner_product_matrix=[[Integer(1),-
↪Integer(1)],[Integer(2),Integer(5)]])
>>> N.inner_product_matrix()
[ 1 -1]
[ 2  5]
>>> u, v = N.basis()
>>> u.inner_product(v)
-1
>>> v.inner_product(u)
2
```

The inner product matrix is defined with respect to the ambient space:

```
sage: V = QQ^3
sage: u = V([1/2,1,1])
sage: v = V([1,1,1/2])
sage: M = span([u,v], ZZ)
sage: M.inner_product_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
sage: M.inner_product_matrix() == V.inner_product_matrix()
True
sage: M.gram_matrix()
[ 1/2 -3/4]
[-3/4 13/4]
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3)
>>> u = V([Integer(1)/Integer(2),Integer(1),Integer(1)])
>>> v = V([Integer(1),Integer(1),Integer(1)/Integer(2)])
>>> M = span([u,v], ZZ)
>>> M.inner_product_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
>>> M.inner_product_matrix() == V.inner_product_matrix()
True
>>> M.gram_matrix()
[ 1/2 -3/4]
[-3/4 13/4]
```

class sage.modules.free_quadratic_module.**FreeQuadraticModule_generic_field**(*base_field*, *dimension*, *degree*, *inner_product_matrix*, *sparse=False*)

Bases: *FreeModule_generic_field*, *FreeQuadraticModule_generic_pid*

Base class for all free modules over fields.

span(*gens*, *check=True*, *already_echelonized=False*)

Return the K-span of the given list of gens, where K is the base field of `self`.

Note that this span is a subspace of the ambient vector space, but need not be a subspace of `self`.

INPUT:

- `gens` – list of vectors
- `check` – boolean (default: `True`); whether or not to coerce entries of gens into base field
- `already_echelonized` – boolean (default: `False`); set this if you know the gens are already in echelon form

EXAMPLES:

```
sage: V = VectorSpace(GF(7), 3)
sage: W = V.subspace([[2,3,4]]); W
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 5 2]
sage: W.span([[1,1,1]])
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 1 1]
```

```
>>> from sage.all import *
>>> V = VectorSpace(GF(Integer(7)), Integer(3))
>>> W = V.subspace([[Integer(2),Integer(3),Integer(4)]]); W
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
```

(continues on next page)

(continued from previous page)

```
[1 5 2]
>>> W.span([[Integer(1),Integer(1),Integer(1)]])
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 1 1]
```

span_of_basis(*basis*, *check=True*, *already_echelonized=False*)

Return the free K-module with the given basis, where K is the base field of `self`.

Note that this span is a subspace of the ambient vector space, but need not be a subspace of `self`.

INPUT:

- `basis` – list of vectors
- `check` – boolean (default: `True`); whether or not to coerce entries of gens into base field
- `already_echelonized` – boolean (default: `False`); set this if you know the gens are already in echelon form

EXAMPLES:

```
sage: V = VectorSpace(GF(7), 3)
sage: W = V.subspace([[2,3,4]]); W
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 5 2]
sage: W.span_of_basis([[2,2,2], [3,3,0]])
Vector space of degree 3 and dimension 2 over Finite Field of size 7
User basis matrix:
[2 2 2]
[3 3 0]
```

```
>>> from sage.all import *
>>> V = VectorSpace(GF(Integer(7)), Integer(3))
>>> W = V.subspace([[Integer(2),Integer(3),Integer(4)]]); W
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 5 2]
>>> W.span_of_basis([[Integer(2),Integer(2),Integer(2)], [Integer(3),
↪Integer(3),Integer(0)]])
Vector space of degree 3 and dimension 2 over Finite Field of size 7
User basis matrix:
[2 2 2]
[3 3 0]
```

The basis vectors must be linearly independent or a `ValueError` exception is raised:

```
sage: W.span_of_basis([[2,2,2], [3,3,3]])
Traceback (most recent call last):
...
ValueError: The given basis vectors must be linearly independent.
```

```
>>> from sage.all import *
>>> W.span_of_basis([[Integer(2),Integer(2),Integer(2)], [Integer(3),
```

(continues on next page)

(continued from previous page)

```
↪Integer(3),Integer(3)]])
Traceback (most recent call last):
...
ValueError: The given basis vectors must be linearly independent.
```

class sage.modules.free_quadratic_module.**FreeQuadraticModule_generic_pid**(*base_ring*, *rank*, *degree*, *inner_product_matrix*, *sparse=False*)

Bases: *FreeModule_generic_pid*, *FreeQuadraticModule_generic*

Class of all free modules over a PID.

span(*gens*, *check=True*, *already_echelonized=False*)

Return the R-span of the given list of gens, where R is the base ring of `self`.

Note that this span need not be a submodule of `self`, nor even of the ambient space. It must, however, be contained in the ambient vector space, i.e., the ambient space tensored with the fraction field of R.

EXAMPLES:

```
sage: V = FreeModule(ZZ,3)
sage: W = V.submodule([V.gen(0)])
sage: W.span([V.gen(1)])
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[0 1 0]
sage: W.submodule([V.gen(1)])
Traceback (most recent call last):
...
ArithmeticError: argument gens (= [(0, 1, 0)]) does not generate a submodule␣
↪of self
```

```
>>> from sage.all import *
>>> V = FreeModule(ZZ,Integer(3))
>>> W = V.submodule([V.gen(Integer(0))])
>>> W.span([V.gen(Integer(1))])
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[0 1 0]
>>> W.submodule([V.gen(Integer(1))])
Traceback (most recent call last):
...
ArithmeticError: argument gens (= [(0, 1, 0)]) does not generate a submodule␣
↪of self
```

span_of_basis(*basis*, *check=True*, *already_echelonized=False*)

Return the free R-module with the given basis, where R is the base ring of `self`.

Note that this R-module need not be a submodule of `self`, nor even of the ambient space. It must, however, be contained in the ambient vector space, i.e., the ambient space tensored with the fraction field of R.

EXAMPLES:

```
sage: M = FreeModule(ZZ,3)
sage: W = M.span_of_basis([M([1,2,3])])
```

```
>>> from sage.all import *
>>> M = FreeModule(ZZ,Integer(3))
>>> W = M.span_of_basis([M([Integer(1),Integer(2),Integer(3)])])
```

Next we create two free $\mathbf{Z}$-modules, neither of which is a submodule of W:

```
sage: W.span_of_basis([M([2,4,0])])
Free module of degree 3 and rank 1 over Integer Ring
User basis matrix:
[2 4 0]
```

```
>>> from sage.all import *
>>> W.span_of_basis([M([Integer(2),Integer(4),Integer(0)])])
Free module of degree 3 and rank 1 over Integer Ring
User basis matrix:
[2 4 0]
```

The following module is not even in the ambient space:

```
sage: Q = QQ
sage: W.span_of_basis([ Q('1/5')*M([1,2,0]), Q('1/7')*M([1,1,0]) ])
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1/5 2/5   0]
[1/7 1/7   0]
```

```
>>> from sage.all import *
>>> Q = QQ
>>> W.span_of_basis([ Q('1/5')*M([Integer(1),Integer(2),Integer(0)]), Q('1/7
↪')*M([Integer(1),Integer(1),Integer(0)]) ])
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1/5 2/5   0]
[1/7 1/7   0]
```

Of course the input basis vectors must be linearly independent:

```
sage: W.span_of_basis([ [1,2,0], [2,4,0] ])
Traceback (most recent call last):
...
ValueError: The given basis vectors must be linearly independent.
```

```
>>> from sage.all import *
>>> W.span_of_basis([ [Integer(1),Integer(2),Integer(0)], [Integer(2),
↪Integer(4),Integer(0)] ])
Traceback (most recent call last):
...
ValueError: The given basis vectors must be linearly independent.
```

zero_submodule()

Return the zero submodule of this module.

EXAMPLES:

```
sage: V = FreeModule(ZZ,2)
sage: V.zero_submodule()
Free module of degree 2 and rank 0 over Integer Ring
Echelon basis matrix:
[]
```

```
>>> from sage.all import *
>>> V = FreeModule(ZZ,Integer(2))
>>> V.zero_submodule()
Free module of degree 2 and rank 0 over Integer Ring
Echelon basis matrix:
[]
```

class sage.modules.free_quadratic_module.**FreeQuadraticModule_submodule_field**(*ambient*, *gens*, *inner_product_matrix*, *check=True*, *already_echelonized=False*)

Bases: *FreeModule_submodule_field*, *FreeQuadraticModule_submodule_with_basis_field*

An embedded vector subspace with echelonized basis.

EXAMPLES:

Since this is an embedded vector subspace with echelonized basis, the methods `echelon_coordinates()` and `coordinates()` return the same coordinates:

```
sage: V = QQ^3
sage: W = V.span([[1,2,3], [4,5,6]])
sage: W
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]

sage: v = V([1,5,9])
sage: W.echelon_coordinates(v)
[1, 5]
sage: vector(QQ, W.echelon_coordinates(v)) * W.basis_matrix()
(1, 5, 9)

sage: v = V([1,5,9])
sage: W.coordinates(v)
[1, 5]
sage: vector(QQ, W.coordinates(v)) * W.basis_matrix()
(1, 5, 9)
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3)
>>> W = V.span([[Integer(1),Integer(2),Integer(3)], [Integer(4),Integer(5),
↪Integer(6)]])
>>> W
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]

>>> v = V([Integer(1),Integer(5),Integer(9)])
>>> W.echelon_coordinates(v)
[1, 5]
>>> vector(QQ, W.echelon_coordinates(v)) * W.basis_matrix()
(1, 5, 9)

>>> v = V([Integer(1),Integer(5),Integer(9)])
>>> W.coordinates(v)
[1, 5]
>>> vector(QQ, W.coordinates(v)) * W.basis_matrix()
(1, 5, 9)
```

class sage.modules.free_quadratic_module.**FreeQuadraticModule_submodule_pid**(*ambient*, *gens*, *inner_product_matrix*, *check=True*, *already_echelonized=False*)

Bases: *FreeModule_submodule_pid*, *FreeQuadraticModule_submodule_with_basis_pid*

An R-submodule of K^n where K is the fraction field of a principal ideal domain R.

EXAMPLES:

```
sage: M = ZZ^3
sage: W = M.span_of_basis([[1,2,3], [4,5,19]]); W
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[ 1  2  3]
[ 4  5 19]
```

```
>>> from sage.all import *
>>> M = ZZ**Integer(3)
>>> W = M.span_of_basis([[Integer(1),Integer(2),Integer(3)], [Integer(4),
↪Integer(5),Integer(19)]]); W
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[ 1  2  3]
[ 4  5 19]
```

We can save and load submodules and elements:

```
sage: loads(W.dumps()) == W
True
```

(continues on next page)

(continued from previous page)

```
sage: v = W.0 + W.1
sage: loads(v.dumps()) == v
True
```

```
>>> from sage.all import *
>>> loads(W.dumps()) == W
True
>>> v = W.gen(0) + W.gen(1)
>>> loads(v.dumps()) == v
True
```

class sage.modules.free_quadratic_module.**FreeQuadraticModule_submodule_with_basis_field**(*ambient*, *basis*, *inner_product_matrix*, *check=True*, *echelonize=False*, *echelonized_basis=None*, *already_echelonized=False*)

Bases: *FreeModule_submodule_with_basis_field*, *FreeQuadraticModule_generic_field*, *FreeQuadraticModule_submodule_with_basis_pid*

An embedded vector subspace with a distinguished user basis.

EXAMPLES:

```
sage: M = QQ^3; W = M.submodule_with_basis([[1,2,3], [4,5,19]]); W
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[ 1  2  3]
[ 4  5 19]
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> M = QQ**Integer(3); W = M.submodule_with_basis([[Integer(1),Integer(2),
↪Integer(3)], [Integer(4),Integer(5),Integer(19)]]); W
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[ 1  2  3]
[ 4  5 19]
```

Since this is an embedded vector subspace with a distinguished user basis possibly different than the echelonized basis, the `echelon_coordinates()` and user `coordinates()` do not agree:

```
sage: V = QQ^3
sage: W = V.submodule_with_basis([[1,2,3], [4,5,6]])
sage: W
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[1 2 3]
[4 5 6]

sage: v = V([1,5,9])
sage: W.echelon_coordinates(v)
[1, 5]
sage: vector(QQ, W.echelon_coordinates(v)) * W.echelonized_basis_matrix()
(1, 5, 9)

sage: v = V([1,5,9])
sage: W.coordinates(v)
[5, -1]
sage: vector(QQ, W.coordinates(v)) * W.basis_matrix()
(1, 5, 9)
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3)
>>> W = V.submodule_with_basis([[Integer(1),Integer(2),Integer(3)], [Integer(4),
↪Integer(5),Integer(6)]])
>>> W
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[1 2 3]
[4 5 6]

>>> v = V([Integer(1),Integer(5),Integer(9)])
>>> W.echelon_coordinates(v)
[1, 5]
>>> vector(QQ, W.echelon_coordinates(v)) * W.echelonized_basis_matrix()
(1, 5, 9)

>>> v = V([Integer(1),Integer(5),Integer(9)])
>>> W.coordinates(v)
[5, -1]
>>> vector(QQ, W.coordinates(v)) * W.basis_matrix()
(1, 5, 9)
```

We can load and save submodules:

```
sage: loads(W.dumps()) == W
True

sage: K.<x> = FractionField(PolynomialRing(QQ,'x'))
sage: M = K^3; W = M.span_of_basis([[1,1,x]])
sage: loads(W.dumps()) == W
True
```

```
>>> from sage.all import *
>>> loads(W.dumps()) == W
True

>>> K = FractionField(PolynomialRing(QQ,'x'), names=('x',)); (x,) = K._first_
↪ngens(1)
>>> M = K**Integer(3); W = M.span_of_basis([[Integer(1),Integer(1),x]])
>>> loads(W.dumps()) == W
True
```

class sage.modules.free_quadratic_module.**FreeQuadraticModule_submodule_with_basis_pid**(*ambient*, *basis*, *inner_product_matrix*, *check=True*, *echelonize=False*, *echelonized_basis=None*, *already_echelonized=False*)

Bases: *FreeModule_submodule_with_basis_pid*, *FreeQuadraticModule_generic_pid*

An R-submodule of K^n with distinguished basis, where K is the fraction field of a principal ideal domain R.

Modules are ordered by inclusion.

EXAMPLES:

First we compare two equal vector spaces:

```
sage: A = FreeQuadraticModule(QQ,3,2*matrix.identity(3))
sage: V = A.span([[1,2,3], [5,6,7], [8,9,10]])
sage: W = A.span([[5,6,7], [8,9,10]])
sage: V == W
True
```

```
>>> from sage.all import *
>>> A = FreeQuadraticModule(QQ,Integer(3),Integer(2)*matrix.identity(Integer(3)))
>>> V = A.span([[Integer(1),Integer(2),Integer(3)], [Integer(5),Integer(6),
↪Integer(7)], [Integer(8),Integer(9),Integer(10)]])
>>> W = A.span([[Integer(5),Integer(6),Integer(7)], [Integer(8),Integer(9),
↪Integer(10)]])
>>> V == W
True
```

Next we compare a one dimensional space to the two dimensional space defined above:

```
sage: M = A.span([[5,6,7]])
sage: V == M
False
sage: M < V
True
sage: V < M
False
```

```
>>> from sage.all import *
>>> M = A.span([[Integer(5),Integer(6),Integer(7)]])
>>> V == M
False
>>> M < V
True
>>> V < M
False
```

We compare a $\mathbf{Z}$-module to the one-dimensional space above:

```
sage: V = A.span([[5,6,7]])
sage: V = V.change_ring(ZZ).scale(1/11)
sage: V < M
True
sage: M < V
False
```

```
>>> from sage.all import *
>>> V = A.span([[Integer(5),Integer(6),Integer(7)]])
>>> V = V.change_ring(ZZ).scale(Integer(1)/Integer(11))
>>> V < M
True
>>> M < V
False
```

change_ring(R)

Return the free module over R obtained by coercing each element of `self` into a vector over the fraction

field of R, then taking the resulting R-module.

This raises a `TypeError` if coercion is not possible.

INPUT:

- `R` – a principal ideal domain

EXAMPLES:

Changing rings preserves the inner product and the user basis:

```
sage: V = QQ^3
sage: W = V.subspace([[2, '1/2', 1]]); W
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[  1 1/4 1/2]
sage: W.change_ring(GF(7))
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 2 4]

sage: N = FreeModule(ZZ, 2, inner_product_matrix=[[1,-1], [2,5]])
sage: N.inner_product_matrix()
[ 1 -1]
[ 2  5]
sage: Np = N.change_ring(RDF)
sage: Np.inner_product_matrix()
[ 1.0 -1.0]
[ 2.0  5.0]
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3)
>>> W = V.subspace([[Integer(2), '1/2', Integer(1)]]); W
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[  1 1/4 1/2]
>>> W.change_ring(GF(Integer(7)))
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 2 4]

>>> N = FreeModule(ZZ, Integer(2), inner_product_matrix=[[Integer(1),-
↪Integer(1)], [Integer(2),Integer(5)]])
>>> N.inner_product_matrix()
[ 1 -1]
[ 2  5]
>>> Np = N.change_ring(RDF)
>>> Np.inner_product_matrix()
[ 1.0 -1.0]
[ 2.0  5.0]
```

`sage.modules.free_quadratic_module.`**`InnerProductSpace`**(*K*, *dimension*, *inner_product_matrix*, *sparse=False*)

EXAMPLES:

The base can be complicated, as long as it is a field:

```
sage: F.<x> = FractionField(PolynomialRing(ZZ,'x'))
sage: D = diagonal_matrix([x, x - 1, x + 1])
sage: V = QuadraticSpace(F, 3, D)
sage: V
Ambient quadratic space of dimension 3 over
 Fraction Field of Univariate Polynomial Ring in x over Integer Ring
Inner product matrix:
[    x     0     0]
[    0 x - 1     0]
[    0     0 x + 1]
sage: V.basis()
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
```

```
>>> from sage.all import *
>>> F = FractionField(PolynomialRing(ZZ,'x'), names=('x',)); (x,) = F._first_
↪ngens(1)
>>> D = diagonal_matrix([x, x - Integer(1), x + Integer(1)])
>>> V = QuadraticSpace(F, Integer(3), D)
>>> V
Ambient quadratic space of dimension 3 over
 Fraction Field of Univariate Polynomial Ring in x over Integer Ring
Inner product matrix:
[    x     0     0]
[    0 x - 1     0]
[    0     0 x + 1]
>>> V.basis()
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
```

The base must be a field or a `TypeError` is raised:

```
sage: QuadraticSpace(ZZ, 5, identity_matrix(ZZ,2))
Traceback (most recent call last):
...
TypeError: argument K (= Integer Ring) must be a field
```

```
>>> from sage.all import *
>>> QuadraticSpace(ZZ, Integer(5), identity_matrix(ZZ,Integer(2)))
Traceback (most recent call last):
...
TypeError: argument K (= Integer Ring) must be a field
```

sage.modules.free_quadratic_module.**QuadraticSpace**(*K*, *dimension*, *inner_product_matrix*, *sparse=False*)

EXAMPLES:

The base can be complicated, as long as it is a field:

```
sage: F.<x> = FractionField(PolynomialRing(ZZ,'x'))
sage: D = diagonal_matrix([x, x - 1, x + 1])
sage: V = QuadraticSpace(F, 3, D)
sage: V
Ambient quadratic space of dimension 3 over
 Fraction Field of Univariate Polynomial Ring in x over Integer Ring
```

(continues on next page)

(continued from previous page)

```
Inner product matrix:
[    x     0     0]
[    0 x - 1     0]
[    0     0 x + 1]
sage: V.basis()
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
```

```
>>> from sage.all import *
>>> F = FractionField(PolynomialRing(ZZ,'x'), names=('x',)); (x,) = F._first_
↪ngens(1)
>>> D = diagonal_matrix([x, x - Integer(1), x + Integer(1)])
>>> V = QuadraticSpace(F, Integer(3), D)
>>> V
Ambient quadratic space of dimension 3 over
 Fraction Field of Univariate Polynomial Ring in x over Integer Ring
Inner product matrix:
[    x     0     0]
[    0 x - 1     0]
[    0     0 x + 1]
>>> V.basis()
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
```

The base must be a field or a `TypeError` is raised:

```
sage: QuadraticSpace(ZZ, 5, identity_matrix(ZZ,2))
Traceback (most recent call last):
...
TypeError: argument K (= Integer Ring) must be a field
```

```
>>> from sage.all import *
>>> QuadraticSpace(ZZ, Integer(5), identity_matrix(ZZ,Integer(2)))
Traceback (most recent call last):
...
TypeError: argument K (= Integer Ring) must be a field
```

sage.modules.free_quadratic_module.**is_FreeQuadraticModule**(M)

Return `True` if M is a free quadratic module.

EXAMPLES:

```
sage: from sage.modules.free_quadratic_module import is_FreeQuadraticModule
sage: U = FreeModule(QQ,3)
sage: is_FreeQuadraticModule(U)
doctest:warning...
DeprecationWarning: the function is_FreeQuadraticModule is deprecated;
use 'isinstance(..., FreeQuadraticModule_generic)' instead
See https://github.com/sagemath/sage/issues/37924 for details.
False
sage: V = FreeModule(QQ,3,inner_product_matrix=diagonal_matrix([1,1,1]))
sage: is_FreeQuadraticModule(V)
True
sage: W = FreeModule(QQ,3,inner_product_matrix=diagonal_matrix([2,3,3]))
```

(continues on next page)

(continued from previous page)

```
sage: is_FreeQuadraticModule(W)
True
```

```
>>> from sage.all import *
>>> from sage.modules.free_quadratic_module import is_FreeQuadraticModule
>>> U = FreeModule(QQ,Integer(3))
>>> is_FreeQuadraticModule(U)
doctest:warning...
DeprecationWarning: the function is_FreeQuadraticModule is deprecated;
use 'isinstance(..., FreeQuadraticModule_generic)' instead
See https://github.com/sagemath/sage/issues/37924 for details.
False
>>> V = FreeModule(QQ,Integer(3),inner_product_matrix=diagonal_matrix([Integer(1),
↪Integer(1),Integer(1)]))
>>> is_FreeQuadraticModule(V)
True
>>> W = FreeModule(QQ,Integer(3),inner_product_matrix=diagonal_matrix([Integer(2),
↪Integer(3),Integer(3)]))
>>> is_FreeQuadraticModule(W)
True
```

6.3 Integral lattices

An integral lattice is a finitely generated free abelian group $L \cong \mathbf{Z}^r$ equipped with a non-degenerate, symmetric bilinear form $L \times L : \to \mathbf{Z}$.

Here, lattices have an ambient quadratic space $\mathbf{Q}^n$ and a distinguished basis.

EXAMPLES:

```
sage: M = Matrix(ZZ, [[0,1], [1,0]])
sage: IntegralLattice(M)
Lattice of degree 2 and rank 2 over Integer Ring
Standard basis
Inner product matrix:
[0 1]
[1 0]
```

```
>>> from sage.all import *
>>> M = Matrix(ZZ, [[Integer(0),Integer(1)], [Integer(1),Integer(0)]])
>>> IntegralLattice(M)
Lattice of degree 2 and rank 2 over Integer Ring
Standard basis
Inner product matrix:
[0 1]
[1 0]
```

A lattice can be defined by an inner product matrix of the ambient space and a basis:

```
sage: G = matrix.identity(3)
sage: basis = [[1,-1,0], [0,1,-1]]
sage: L = IntegralLattice(G, basis)
```

(continues on next page)

(continued from previous page)

```
sage: L
Lattice of degree 3 and rank 2 over Integer Ring
Basis matrix:
[ 1 -1  0]
[ 0  1 -1]
Standard scalar product

sage: L.gram_matrix()
[ 2 -1]
[-1  2]
```

```
>>> from sage.all import *
>>> G = matrix.identity(Integer(3))
>>> basis = [[Integer(1),-Integer(1),Integer(0)], [Integer(0),Integer(1),-Integer(1)]]
>>> L = IntegralLattice(G, basis)
>>> L
Lattice of degree 3 and rank 2 over Integer Ring
Basis matrix:
[ 1 -1  0]
[ 0  1 -1]
Standard scalar product

>>> L.gram_matrix()
[ 2 -1]
[-1  2]
```

AUTHORS:

- Simon Brandhorst (2017-09): First created
- Paolo Menegatti (2018-03): Added IntegralLatticeDirectSum, IntegralLatticeGluing
- Lorenz Panny (2024): enumeration routines for short and close vectors

class sage.modules.free_quadratic_module_integer_symmetric.**FreeQuadraticModule_integer_symmetric**(*ambient*, *basis*, *inner_product_matrix*, *check*, *already_echelonized*)

Bases: *FreeQuadraticModule_submodule_with_basis_pid*

This class represents non-degenerate, integral, symmetric free quadratic **Z**-modules.

INPUT:

- `ambient` – an ambient free quadratic module
- `basis` – list of elements of ambient or a matrix
- `inner_product_matrix` – a symmetric matrix over the rationals

EXAMPLES:

```
sage: IntegralLattice("U", basis=[vector([1,1])])
Lattice of degree 2 and rank 1 over Integer Ring
Basis matrix:
[1 1]
Inner product matrix:
[0 1]
[1 0]
```

```
>>> from sage.all import *
>>> IntegralLattice("U", basis=[vector([Integer(1),Integer(1)])])
Lattice of degree 2 and rank 1 over Integer Ring
Basis matrix:
[1 1]
Inner product matrix:
[0 1]
[1 0]
```

LLL()

Return this lattice with an LLL reduced basis.

EXAMPLES:

```
sage: L = IntegralLattice('A2')                                             #␣
↪needs sage.graphs
sage: L.lll() == L                                                          #␣
↪needs sage.graphs sage.libs.pari
True

sage: G = matrix(ZZ, 3, [0,1,0, 1,0,0, 0,0,7])
sage: V = matrix(ZZ, 3, [-14,-15,-15, -4,1,16, -5,-5,-4])
sage: L = IntegralLattice(V * G * V.T)
sage: L.lll().gram_matrix()                                                 #␣
↪needs sage.libs.gap
[0 0 1]
[0 7 0]
[1 0 0]
```

```
>>> from sage.all import *
>>> L = IntegralLattice('A2')                                               #␣
↪needs sage.graphs
>>> L.lll() == L                                                            #␣
↪needs sage.graphs sage.libs.pari
True

>>> G = matrix(ZZ, Integer(3), [Integer(0),Integer(1),Integer(0), Integer(1),
↪Integer(0),Integer(0), Integer(0),Integer(0),Integer(7)])
>>> V = matrix(ZZ, Integer(3), [-Integer(14),-Integer(15),-Integer(15), -
```

(continues on next page)

(continued from previous page)

```
↪Integer(4),Integer(1),Integer(16), -Integer(5),-Integer(5),-Integer(4)])
>>> L = IntegralLattice(V * G * V.T)
>>> L.lll().gram_matrix()                                                    #␣
↪needs sage.libs.gap
[0 0 1]
[0 7 0]
[1 0 0]
```

automorphisms(*gens=None*, *is_finite=None*)

Return the orthogonal group of this lattice as a matrix group.

The elements are isometries of the ambient vector space which preserve this lattice. They are represented by matrices with respect to the standard basis.

INPUT:

- `gens` – list of matrices (default: `None`)
- `is_finite` – boolean (default: `None`); if set to `True`, then the group is placed in the category of finite groups. Sage does not check this.

OUTPUT:

The matrix group generated by `gens`. If `gens` is not specified, then generators of the full orthogonal group of this lattice are computed. They are continued as the identity on the orthogonal complement of the lattice in its ambient space. Currently, we can only compute the orthogonal group for positive definite lattices.

EXAMPLES:

```
sage: A4 = IntegralLattice("A4")                                            #␣
↪needs sage.graphs
sage: Aut = A4.orthogonal_group(); Aut                                      #␣
↪needs sage.graphs sage.libs.gap
Group of isometries with 4 generators (
[0 0 0 1]  [-1 -1 -1  0]  [ 1  0  0  0]  [ 1  0  0  0]
[0 0 1 0]  [ 0  0  0 -1]  [-1 -1 -1 -1]  [ 0  1  0  0]
[0 1 0 0]  [ 0  0  1  1]  [ 0  0  0  1]  [ 0  0  1  1]
[1 0 0 0], [ 0  1  0  0], [ 0  0  1  0], [ 0  0  0 -1]
)
```

```
>>> from sage.all import *
>>> A4 = IntegralLattice("A4")                                              #␣
↪needs sage.graphs
>>> Aut = A4.orthogonal_group(); Aut                                        #␣
↪needs sage.graphs sage.libs.gap
Group of isometries with 4 generators (
[0 0 0 1]  [-1 -1 -1  0]  [ 1  0  0  0]  [ 1  0  0  0]
[0 0 1 0]  [ 0  0  0 -1]  [-1 -1 -1 -1]  [ 0  1  0  0]
[0 1 0 0]  [ 0  0  1  1]  [ 0  0  0  1]  [ 0  0  1  1]
[1 0 0 0], [ 0  1  0  0], [ 0  0  1  0], [ 0  0  0 -1]
)
```

The group acts from the right on the lattice and its discriminant group:

```
sage: # needs sage.graphs sage.libs.gap
sage: x = A4.an_element()
sage: g = Aut.an_element(); g
[-1 -1 -1  0]
[ 0  0  1  0]
[ 0  0 -1 -1]
[ 0  1  1  1]
sage: x*g
(-1, -1, -1, 0)
sage: (x*g).parent() == A4
True
sage: (g*x).parent()
Vector space of dimension 4 over Rational Field
sage: y = A4.discriminant_group().an_element()
sage: y*g
(4)
```

```
>>> from sage.all import *
>>> # needs sage.graphs sage.libs.gap
>>> x = A4.an_element()
>>> g = Aut.an_element(); g
[-1 -1 -1  0]
[ 0  0  1  0]
[ 0  0 -1 -1]
[ 0  1  1  1]
>>> x*g
(-1, -1, -1, 0)
>>> (x*g).parent() == A4
True
>>> (g*x).parent()
Vector space of dimension 4 over Rational Field
>>> y = A4.discriminant_group().an_element()
>>> y*g
(4)
```

If the group is finite we can compute the usual things:

```
sage: # needs sage.graphs sage.libs.gap
sage: Aut.order()
240
sage: conj = Aut.conjugacy_classes_representatives()
sage: len(conj)
14
sage: Aut.structure_description()
'C2 x S5'
```

```
>>> from sage.all import *
>>> # needs sage.graphs sage.libs.gap
>>> Aut.order()
240
>>> conj = Aut.conjugacy_classes_representatives()
>>> len(conj)
```

(continues on next page)

(continued from previous page)

```
14
>>> Aut.structure_description()
'C2 x S5'
```

The lattice can live in a larger ambient space:

```
sage: A2 = IntegralLattice(matrix.identity(3),
....:                      Matrix(ZZ, 2, 3, [1,-1,0,0,1,-1]))
sage: A2.orthogonal_group()                                                    #␣
↪needs sage.libs.gap
Group of isometries with 2 generators (
[ 2/3  2/3 -1/3]  [1 0 0]
[ 2/3 -1/3  2/3]  [0 0 1]
[-1/3  2/3  2/3], [0 1 0]
)
```

```
>>> from sage.all import *
>>> A2 = IntegralLattice(matrix.identity(Integer(3)),
...                      Matrix(ZZ, Integer(2), Integer(3), [Integer(1),-
↪Integer(1),Integer(0),Integer(0),Integer(1),-Integer(1)]))
>>> A2.orthogonal_group()                                                      #␣
↪needs sage.libs.gap
Group of isometries with 2 generators (
[ 2/3  2/3 -1/3]  [1 0 0]
[ 2/3 -1/3  2/3]  [0 0 1]
[-1/3  2/3  2/3], [0 1 0]
)
```

It can be negative definite as well:

```
sage: A2m = IntegralLattice(-Matrix(ZZ, 2, [2,1,1,2]))
sage: G = A2m.orthogonal_group()                                               #␣
↪needs sage.libs.gap
sage: G.order()                                                                #␣
↪needs sage.libs.gap
12
```

```
>>> from sage.all import *
>>> A2m = IntegralLattice(-Matrix(ZZ, Integer(2), [Integer(2),Integer(1),
↪Integer(1),Integer(2)]))
>>> G = A2m.orthogonal_group()                                                 #␣
↪needs sage.libs.gap
>>> G.order()                                                                  #␣
↪needs sage.libs.gap
12
```

If the lattice is indefinite, sage does not know how to compute generators. Can you teach it?:

```
sage: U = IntegralLattice(Matrix(ZZ, 2, [0,1,1,0]))
sage: U.orthogonal_group()                                                     #␣
↪needs sage.libs.gap
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
NotImplementedError: currently, we can only compute generators
for orthogonal groups over definite lattices.
```

```
>>> from sage.all import *
>>> U = IntegralLattice(Matrix(ZZ, Integer(2), [Integer(0),Integer(1),
↪Integer(1),Integer(0)]))
>>> U.orthogonal_group()                                                    #␣
↪needs sage.libs.gap
Traceback (most recent call last):
...
NotImplementedError: currently, we can only compute generators
for orthogonal groups over definite lattices.
```

But we can define subgroups:

```
sage: S = IntegralLattice(Matrix(ZZ, 2, [2, 3, 3, 2]))
sage: f = Matrix(ZZ, 2, [0,1,-1,3])
sage: S.orthogonal_group([f])                                              #␣
↪needs sage.libs.gap
Group of isometries with 1 generator (
[ 0  1]
[-1  3]
)
```

```
>>> from sage.all import *
>>> S = IntegralLattice(Matrix(ZZ, Integer(2), [Integer(2), Integer(3),␣
↪Integer(3), Integer(2)]))
>>> f = Matrix(ZZ, Integer(2), [Integer(0),Integer(1),-Integer(1),Integer(3)])
>>> S.orthogonal_group([f])                                                #␣
↪needs sage.libs.gap
Group of isometries with 1 generator (
[ 0  1]
[-1  3]
)
```

direct_sum(*M*)

Return the direct sum of this lattice with `M`.

INPUT:

- `M` – a module over **Z**

EXAMPLES:

```
sage: A = IntegralLattice(1)
sage: A.direct_sum(A)
Lattice of degree 2 and rank 2 over Integer Ring
Standard basis
Standard scalar product
```

```
>>> from sage.all import *
>>> A = IntegralLattice(Integer(1))
```

(continues on next page)

(continued from previous page)

```
>>> A.direct_sum(A)
Lattice of degree 2 and rank 2 over Integer Ring
Standard basis
Standard scalar product
```

discriminant_group(*s=0*)

Return the discriminant group $L^\vee/L$ of this lattice.

INPUT:

- s – integer (default: 0)

OUTPUT:

The s primary part of the discriminant group. If $s = 0$, returns the whole discriminant group.

EXAMPLES:

```
sage: L = IntegralLattice(Matrix(ZZ, 2, 2, [2,1,1,-2]) * 2)
sage: L.discriminant_group()
Finite quadratic module over Integer Ring with invariants (2, 10)
Gram matrix of the quadratic form with values in Q/2Z:
[  1 1/2]
[1/2 1/5]
sage: L.discriminant_group(2)
Finite quadratic module over Integer Ring with invariants (2, 2)
Gram matrix of the quadratic form with values in Q/2Z:
[  1 1/2]
[1/2   1]
sage: L.discriminant_group(5)
Finite quadratic module over Integer Ring with invariants (5,)
Gram matrix of the quadratic form with values in Q/2Z:
[4/5]
```

```
>>> from sage.all import *
>>> L = IntegralLattice(Matrix(ZZ, Integer(2), Integer(2), [Integer(2),
↪Integer(1),Integer(1),-Integer(2)]) * Integer(2))
>>> L.discriminant_group()
Finite quadratic module over Integer Ring with invariants (2, 10)
Gram matrix of the quadratic form with values in Q/2Z:
[  1 1/2]
[1/2 1/5]
>>> L.discriminant_group(Integer(2))
Finite quadratic module over Integer Ring with invariants (2, 2)
Gram matrix of the quadratic form with values in Q/2Z:
[  1 1/2]
[1/2   1]
>>> L.discriminant_group(Integer(5))
Finite quadratic module over Integer Ring with invariants (5,)
Gram matrix of the quadratic form with values in Q/2Z:
[4/5]
```

dual_lattice()

Return the dual lattice as a *FreeQuadraticModule*.

Let L be a lattice. Its dual lattice is

$$L^\vee = \{x \in L \otimes \mathbf{Q} : (x, l) \in \mathbf{Z}\ \forall l \in L\}.$$

EXAMPLES:

```
sage: L = IntegralLattice("A2")                                                #␣
↪needs sage.graphs
sage: Ldual = L.dual_lattice(); Ldual                                          #␣
↪needs sage.graphs
Free module of degree 2 and rank 2 over Integer Ring
Echelon basis matrix:
[1/3 2/3]
[  0   1]
```

```
>>> from sage.all import *
>>> L = IntegralLattice("A2")                                                  #␣
↪needs sage.graphs
>>> Ldual = L.dual_lattice(); Ldual                                            #␣
↪needs sage.graphs
Free module of degree 2 and rank 2 over Integer Ring
Echelon basis matrix:
[1/3 2/3]
[  0   1]
```

Since our lattices are always integral, a lattice is contained in its dual:

```
sage: L.is_submodule(Ldual)                                                    #␣
↪needs sage.graphs
True
```

```
>>> from sage.all import *
>>> L.is_submodule(Ldual)                                                      #␣
↪needs sage.graphs
True
```

enumerate_close_vectors(*target*)

Return an iterator over all the vectors in this lattice, starting from vectors relatively close to the given `target` vector.

Warning

The returned vectors are not necessarily ordered strictly by their distance to the target.

EXAMPLES:

```
sage: L = IntegralLattice(4, [[1,2,3,4], [7,7,8,8], [1,-1,1,0]])
sage: t = vector([1/2, -133/7, 123.44, -11])
sage: close = L.enumerate_close_vectors(t)
sage: vecs = [next(close) for _ in range(10)]
sage: sorted(vecs, key=lambda v: (L(v).inner_product(L(v)), v))
[(-1, -16, 121, 148), (0, -17, 122, 148), (-3, -22, 122, 148), (1, -18, 123,␣
```

(continues on next page)

(continued from previous page)

```
↪148), (-2, -23, 123, 148),
 (2, -19, 124, 148), (3, -20, 125, 148), (4, -21, 126, 148), (-3, -19, 124,␣
↪152), (-2, -20, 125, 152)]
```

```
>>> from sage.all import *
>>> L = IntegralLattice(Integer(4), [[Integer(1),Integer(2),Integer(3),
↪Integer(4)], [Integer(7),Integer(7),Integer(8),Integer(8)], [Integer(1),-
↪Integer(1),Integer(1),Integer(0)]])
>>> t = vector([Integer(1)/Integer(2), -Integer(133)/Integer(7), RealNumber(
↪'123.44'), -Integer(11)])
>>> close = L.enumerate_close_vectors(t)
>>> vecs = [next(close) for _ in range(Integer(10))]
>>> sorted(vecs, key=lambda v: (L(v).inner_product(L(v)), v))
[(-1, -16, 121, 148), (0, -17, 122, 148), (-3, -22, 122, 148), (1, -18, 123,␣
↪148), (-2, -23, 123, 148),
 (2, -19, 124, 148), (3, -20, 125, 148), (4, -21, 126, 148), (-3, -19, 124,␣
↪152), (-2, -20, 125, 152)]
```

enumerate_short_vectors()

Return an iterator over all the vectors in this lattice (modulo sign), starting from shorter vectors.

Warning

The returned vectors are not necessarily ordered strictly by length.

EXAMPLES:

```
sage: L = IntegralLattice(4, [[1,2,3,4], [7,7,8,8], [1,-1,1,0]])
sage: short = L.enumerate_short_vectors()
sage: vecs = [next(short) for _ in range(20)]
sage: sorted(vecs, key=lambda v: (L(v).inner_product(L(v)), v))
[(1, -1, 1, 0), (2, -2, 2, 0), (3, -3, 3, 0), (0, 3, 2, 4), (1, 2, 3, 4),
 (3, 2, -2, -4), (4, 4, 1, 0), (-1, 4, 1, 4), (3, 5, 0, 0), (4, 1, -1, -4),
 (2, 1, 4, 4), (2, 3, -3, -4), (5, 3, 2, 0), (2, 6, -1, 0), (5, 0, 0, -4),
 (-2, 5, 0, 4), (4, -4, 4, 0), (1, 4, -4, -4), (6, 2, 3, 0), (3, 0, 5, 4)]
```

```
>>> from sage.all import *
>>> L = IntegralLattice(Integer(4), [[Integer(1),Integer(2),Integer(3),
↪Integer(4)], [Integer(7),Integer(7),Integer(8),Integer(8)], [Integer(1),-
↪Integer(1),Integer(1),Integer(0)]])
>>> short = L.enumerate_short_vectors()
>>> vecs = [next(short) for _ in range(Integer(20))]
>>> sorted(vecs, key=lambda v: (L(v).inner_product(L(v)), v))
[(1, -1, 1, 0), (2, -2, 2, 0), (3, -3, 3, 0), (0, 3, 2, 4), (1, 2, 3, 4),
 (3, 2, -2, -4), (4, 4, 1, 0), (-1, 4, 1, 4), (3, 5, 0, 0), (4, 1, -1, -4),
 (2, 1, 4, 4), (2, 3, -3, -4), (5, 3, 2, 0), (2, 6, -1, 0), (5, 0, 0, -4),
 (-2, 5, 0, 4), (4, -4, 4, 0), (1, 4, -4, -4), (6, 2, 3, 0), (3, 0, 5, 4)]
```

This example demonstrates that the lattice inner product is used for the norm:

```
sage: Q = Matrix(QQ, [[1000, 0], [0, 1]])
sage: B = [[1, 1], [1, -1]]
sage: L = IntegralLattice(Q, basis=B)
sage: short = L.enumerate_short_vectors()
sage: vecs = [next(short) for _ in range(20)]
sage: sorted(vecs, key=lambda v: (L(v).inner_product(L(v)), v))
[(0, -2), (0, -4), (0, -6), (0, -8), (0, -10), (0, -12), (0, -14), (0, -16),
(0, -18), (0, -20), (0, -22), (0, -24), (0, -26), (0, -28), (0, -30), (-1, -
↪1),
(-1, 1), (-1, -3), (-1, 3), (0, -32)]
```

```
>>> from sage.all import *
>>> Q = Matrix(QQ, [[Integer(1000), Integer(0)], [Integer(0), Integer(1)]])
>>> B = [[Integer(1), Integer(1)], [Integer(1), -Integer(1)]]
>>> L = IntegralLattice(Q, basis=B)
>>> short = L.enumerate_short_vectors()
>>> vecs = [next(short) for _ in range(Integer(20))]
>>> sorted(vecs, key=lambda v: (L(v).inner_product(L(v)), v))
[(0, -2), (0, -4), (0, -6), (0, -8), (0, -10), (0, -12), (0, -14), (0, -16),
(0, -18), (0, -20), (0, -22), (0, -24), (0, -26), (0, -28), (0, -30), (-1, -
↪1),
(-1, 1), (-1, -3), (-1, 3), (0, -32)]
```

genus()

Return the genus of this lattice.

EXAMPLES:

```
sage: L = IntegralLattice("U")
sage: L.genus()                                                             #␣
↪needs sage.padics
Genus of
[0 1]
[1 0]
Signature:  (1, 1)
Genus symbol at 2:    1^2
```

```
>>> from sage.all import *
>>> L = IntegralLattice("U")
>>> L.genus()                                                               #␣
↪needs sage.padics
Genus of
[0 1]
[1 0]
Signature:  (1, 1)
Genus symbol at 2:    1^2
```

is_even()

Return whether the diagonal entries of the Gram matrix are even.

EXAMPLES:

```
sage: G = Matrix(ZZ, 2, 2, [-1,1,1,2])
sage: L = IntegralLattice(G)
sage: L.is_even()
False
sage: L = IntegralLattice("A2")                                                 #␣
↪needs sage.graphs
sage: L.is_even()                                                               #␣
↪needs sage.graphs
True
```

```
>>> from sage.all import *
>>> G = Matrix(ZZ, Integer(2), Integer(2), [-Integer(1),Integer(1),Integer(1),
↪Integer(2)])
>>> L = IntegralLattice(G)
>>> L.is_even()
False
>>> L = IntegralLattice("A2")                                                   #␣
↪needs sage.graphs
>>> L.is_even()                                                                 #␣
↪needs sage.graphs
True
```

is_primitive(*M*)

Return whether `M` is a primitive submodule of this lattice.

A **Z**-submodule `M` of a **Z**-module `L` is called primitive if the quotient `L/M` is torsion free.

INPUT:

- `M` – a submodule of this lattice

EXAMPLES:

```
sage: U = IntegralLattice("U")
sage: L1 = U.span([vector([1,1])])
sage: L2 = U.span([vector([1,-1])])
sage: U.is_primitive(L1)
True
sage: U.is_primitive(L2)
True
sage: U.is_primitive(L1 + L2)
False
```

```
>>> from sage.all import *
>>> U = IntegralLattice("U")
>>> L1 = U.span([vector([Integer(1),Integer(1)])])
>>> L2 = U.span([vector([Integer(1),-Integer(1)])])
>>> U.is_primitive(L1)
True
>>> U.is_primitive(L2)
True
>>> U.is_primitive(L1 + L2)
False
```

We can also compute the index:

```
sage: (L1 + L2).index_in(U)
2
```

```
>>> from sage.all import *
>>> (L1 + L2).index_in(U)
2
```

lll()

Return this lattice with an LLL reduced basis.

EXAMPLES:

```
sage: L = IntegralLattice('A2')                                                 #␣
↪needs sage.graphs
sage: L.lll() == L                                                              #␣
↪needs sage.graphs sage.libs.pari
True

sage: G = matrix(ZZ, 3, [0,1,0, 1,0,0, 0,0,7])
sage: V = matrix(ZZ, 3, [-14,-15,-15, -4,1,16, -5,-5,-4])
sage: L = IntegralLattice(V * G * V.T)
sage: L.lll().gram_matrix()                                                     #␣
↪needs sage.libs.gap
[0 0 1]
[0 7 0]
[1 0 0]
```

```
>>> from sage.all import *
>>> L = IntegralLattice('A2')                                                 #␣
↪needs sage.graphs
>>> L.lll() == L                                                              #␣
↪needs sage.graphs sage.libs.pari
True

>>> G = matrix(ZZ, Integer(3), [Integer(0),Integer(1),Integer(0), Integer(1),
↪Integer(0),Integer(0), Integer(0),Integer(0),Integer(7)])
>>> V = matrix(ZZ, Integer(3), [-Integer(14),-Integer(15),-Integer(15), -
↪Integer(4),Integer(1),Integer(16), -Integer(5),-Integer(5),-Integer(4)])
>>> L = IntegralLattice(V * G * V.T)
>>> L.lll().gram_matrix()                                                     #␣
↪needs sage.libs.gap
[0 0 1]
[0 7 0]
[1 0 0]
```

max()

Return the maximum of this lattice.

$$\max\{x^2 | x \in L \setminus \{0\}\}$$

EXAMPLES:

```
sage: L = IntegralLattice('A2')                                                #␣
↪needs sage.graphs
sage: L.maximum()                                                              #␣
↪needs sage.graphs
+Infinity
sage: L.twist(-1).maximum()                                                    #␣
↪needs sage.graphs sage.libs.pari
-2
```

```
>>> from sage.all import *
>>> L = IntegralLattice('A2')                                                  #␣
↪needs sage.graphs
>>> L.maximum()                                                               #␣
↪needs sage.graphs
+Infinity
>>> L.twist(-Integer(1)).maximum()                                             ␣
↪     # needs sage.graphs sage.libs.pari
-2
```

maximal_overlattice(*p=None*)

Return a maximal even integral overlattice of this lattice.

INPUT:

- `p` – (default: `None`) if given return an overlattice M of this lattice L that is maximal at p and the completions $M_q = L_q$ are equal for all primes $q \neq p$

If p is 2 or `None`, then the lattice must be even.

EXAMPLES:

```
sage: # needs sage.graphs sage.libs.pari
sage: L = IntegralLattice("A4").twist(25*89)
sage: L.maximal_overlattice().determinant()
5
sage: L.maximal_overlattice(89).determinant().factor()
5^9
sage: L.maximal_overlattice(5).determinant().factor()
5 * 89^4
```

```
>>> from sage.all import *
>>> # needs sage.graphs sage.libs.pari
>>> L = IntegralLattice("A4").twist(Integer(25)*Integer(89))
>>> L.maximal_overlattice().determinant()
5
>>> L.maximal_overlattice(Integer(89)).determinant().factor()
5^9
>>> L.maximal_overlattice(Integer(5)).determinant().factor()
5 * 89^4
```

maximum()

Return the maximum of this lattice.

$$\max\{x^2 | x \in L \setminus \{0\}\}$$

EXAMPLES:

```
sage: L = IntegralLattice('A2')                                                 #␣
↪needs sage.graphs
sage: L.maximum()                                                               #␣
↪needs sage.graphs
+Infinity
sage: L.twist(-1).maximum()                                                     #␣
↪needs sage.graphs sage.libs.pari
-2
```

```
>>> from sage.all import *
>>> L = IntegralLattice('A2')                                                   #␣
↪needs sage.graphs
>>> L.maximum()                                                                 #␣
↪needs sage.graphs
+Infinity
>>> L.twist(-Integer(1)).maximum()                                              ␣
↪      # needs sage.graphs sage.libs.pari
-2
```

min()

Return the minimum of this lattice.

$$\min\{x^2 | x \in L \setminus \{0\}\}$$

EXAMPLES:

```
sage: L = IntegralLattice('A2')                                                 #␣
↪needs sage.graphs
sage: L.minimum()                                                               #␣
↪needs sage.graphs
2
sage: L.twist(-1).minimum()                                                     #␣
↪needs sage.graphs
-Infinity
```

```
>>> from sage.all import *
>>> L = IntegralLattice('A2')                                                   #␣
↪needs sage.graphs
>>> L.minimum()                                                                 #␣
↪needs sage.graphs
2
>>> L.twist(-Integer(1)).minimum()                                              ␣
↪      # needs sage.graphs
-Infinity
```

minimum()

Return the minimum of this lattice.

$$\min\{x^2 | x \in L \setminus \{0\}\}$$

EXAMPLES:

```
sage: L = IntegralLattice('A2')                                                #␣
↪needs sage.graphs
sage: L.minimum()                                                              #␣
↪needs sage.graphs
2
sage: L.twist(-1).minimum()                                                    #␣
↪needs sage.graphs
-Infinity
```

```
>>> from sage.all import *
>>> L = IntegralLattice('A2')                                                  #␣
↪needs sage.graphs
>>> L.minimum()                                                               #␣
↪needs sage.graphs
2
>>> L.twist(-Integer(1)).minimum()                                             ␣
↪     # needs sage.graphs
-Infinity
```

orthogonal_complement(*M*)

Return the orthogonal complement of `M` in this lattice.

INPUT:

- `M` – a module in the same ambient space or a list of elements of the ambient space

EXAMPLES:

```
sage: H5 = Matrix(ZZ, 2, [2,1,1,-2])
sage: L = IntegralLattice(H5)
sage: S = L.span([vector([1,1])])
sage: L.orthogonal_complement(S)
Lattice of degree 2 and rank 1 over Integer Ring
Basis matrix:
[1 3]
Inner product matrix:
[ 2  1]
[ 1 -2]

sage: L = IntegralLattice(2)
sage: L.orthogonal_complement([vector(ZZ, [1,0])])
Lattice of degree 2 and rank 1 over Integer Ring
Basis matrix:
[0 1]
Standard scalar product
```

```
>>> from sage.all import *
>>> H5 = Matrix(ZZ, Integer(2), [Integer(2),Integer(1),Integer(1),-
↪Integer(2)])
>>> L = IntegralLattice(H5)
>>> S = L.span([vector([Integer(1),Integer(1)])])
>>> L.orthogonal_complement(S)
Lattice of degree 2 and rank 1 over Integer Ring
Basis matrix:
```

(continues on next page)

(continued from previous page)

```
[1 3]
Inner product matrix:
[ 2  1]
[ 1 -2]

>>> L = IntegralLattice(Integer(2))
>>> L.orthogonal_complement([vector(ZZ, [Integer(1),Integer(0)])])
Lattice of degree 2 and rank 1 over Integer Ring
Basis matrix:
[0 1]
Standard scalar product
```

orthogonal_group(*gens=None*, *is_finite=None*)

Return the orthogonal group of this lattice as a matrix group.

The elements are isometries of the ambient vector space which preserve this lattice. They are represented by matrices with respect to the standard basis.

INPUT:

- `gens` – list of matrices (default: `None`)
- `is_finite` – boolean (default: `None`); if set to `True`, then the group is placed in the category of finite groups. Sage does not check this.

OUTPUT:

The matrix group generated by `gens`. If `gens` is not specified, then generators of the full orthogonal group of this lattice are computed. They are continued as the identity on the orthogonal complement of the lattice in its ambient space. Currently, we can only compute the orthogonal group for positive definite lattices.

EXAMPLES:

```
sage: A4 = IntegralLattice("A4")                                                #␣
↪needs sage.graphs
sage: Aut = A4.orthogonal_group(); Aut                                          #␣
↪needs sage.graphs sage.libs.gap
Group of isometries with 4 generators (
[0 0 0 1]  [-1 -1 -1  0]  [ 1  0  0  0]  [ 1  0  0  0]
[0 0 1 0]  [ 0  0  0 -1]  [-1 -1 -1 -1]  [ 0  1  0  0]
[0 1 0 0]  [ 0  0  1  1]  [ 0  0  0  1]  [ 0  0  1  1]
[1 0 0 0], [ 0  1  0  0], [ 0  0  1  0], [ 0  0  0 -1]
)
```

```
>>> from sage.all import *
>>> A4 = IntegralLattice("A4")                                                  #␣
↪needs sage.graphs
>>> Aut = A4.orthogonal_group(); Aut                                            #␣
↪needs sage.graphs sage.libs.gap
Group of isometries with 4 generators (
[0 0 0 1]  [-1 -1 -1  0]  [ 1  0  0  0]  [ 1  0  0  0]
[0 0 1 0]  [ 0  0  0 -1]  [-1 -1 -1 -1]  [ 0  1  0  0]
[0 1 0 0]  [ 0  0  1  1]  [ 0  0  0  1]  [ 0  0  1  1]
[1 0 0 0], [ 0  1  0  0], [ 0  0  1  0], [ 0  0  0 -1]
)
```

The group acts from the right on the lattice and its discriminant group:

```
sage: # needs sage.graphs sage.libs.gap
sage: x = A4.an_element()
sage: g = Aut.an_element(); g
[-1 -1 -1  0]
[ 0  0  1  0]
[ 0  0 -1 -1]
[ 0  1  1  1]
sage: x*g
(-1, -1, -1, 0)
sage: (x*g).parent() == A4
True
sage: (g*x).parent()
Vector space of dimension 4 over Rational Field
sage: y = A4.discriminant_group().an_element()
sage: y*g
(4)
```

```
>>> from sage.all import *
>>> # needs sage.graphs sage.libs.gap
>>> x = A4.an_element()
>>> g = Aut.an_element(); g
[-1 -1 -1  0]
[ 0  0  1  0]
[ 0  0 -1 -1]
[ 0  1  1  1]
>>> x*g
(-1, -1, -1, 0)
>>> (x*g).parent() == A4
True
>>> (g*x).parent()
Vector space of dimension 4 over Rational Field
>>> y = A4.discriminant_group().an_element()
>>> y*g
(4)
```

If the group is finite we can compute the usual things:

```
sage: # needs sage.graphs sage.libs.gap
sage: Aut.order()
240
sage: conj = Aut.conjugacy_classes_representatives()
sage: len(conj)
14
sage: Aut.structure_description()
'C2 x S5'
```

```
>>> from sage.all import *
>>> # needs sage.graphs sage.libs.gap
>>> Aut.order()
240
>>> conj = Aut.conjugacy_classes_representatives()
```

(continues on next page)

(continued from previous page)

```
>>> len(conj)
14
>>> Aut.structure_description()
'C2 x S5'
```

The lattice can live in a larger ambient space:

```
sage: A2 = IntegralLattice(matrix.identity(3),
....:                      Matrix(ZZ, 2, 3, [1,-1,0,0,1,-1]))
sage: A2.orthogonal_group()                                                    #␣
↪needs sage.libs.gap
Group of isometries with 2 generators (
[ 2/3  2/3 -1/3]  [1 0 0]
[ 2/3 -1/3  2/3]  [0 0 1]
[-1/3  2/3  2/3], [0 1 0]
)
```

```
>>> from sage.all import *
>>> A2 = IntegralLattice(matrix.identity(Integer(3)),
...                      Matrix(ZZ, Integer(2), Integer(3), [Integer(1),-
↪Integer(1),Integer(0),Integer(0),Integer(1),-Integer(1)]))
>>> A2.orthogonal_group()                                                      #␣
↪needs sage.libs.gap
Group of isometries with 2 generators (
[ 2/3  2/3 -1/3]  [1 0 0]
[ 2/3 -1/3  2/3]  [0 0 1]
[-1/3  2/3  2/3], [0 1 0]
)
```

It can be negative definite as well:

```
sage: A2m = IntegralLattice(-Matrix(ZZ, 2, [2,1,1,2]))
sage: G = A2m.orthogonal_group()                                               #␣
↪needs sage.libs.gap
sage: G.order()                                                                #␣
↪needs sage.libs.gap
12
```

```
>>> from sage.all import *
>>> A2m = IntegralLattice(-Matrix(ZZ, Integer(2), [Integer(2),Integer(1),
↪Integer(1),Integer(2)]))
>>> G = A2m.orthogonal_group()                                                 #␣
↪needs sage.libs.gap
>>> G.order()                                                                  #␣
↪needs sage.libs.gap
12
```

If the lattice is indefinite, sage does not know how to compute generators. Can you teach it?:

```
sage: U = IntegralLattice(Matrix(ZZ, 2, [0,1,1,0]))
sage: U.orthogonal_group()                                                     #␣
↪needs sage.libs.gap
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
NotImplementedError: currently, we can only compute generators
for orthogonal groups over definite lattices.
```

```
>>> from sage.all import *
>>> U = IntegralLattice(Matrix(ZZ, Integer(2), [Integer(0),Integer(1),
↪Integer(1),Integer(0)]))
>>> U.orthogonal_group()                                                #␣
↪needs sage.libs.gap
Traceback (most recent call last):
...
NotImplementedError: currently, we can only compute generators
for orthogonal groups over definite lattices.
```

But we can define subgroups:

```
sage: S = IntegralLattice(Matrix(ZZ, 2, [2, 3, 3, 2]))
sage: f = Matrix(ZZ, 2, [0,1,-1,3])
sage: S.orthogonal_group([f])                                          #␣
↪needs sage.libs.gap
Group of isometries with 1 generator (
[ 0  1]
[-1  3]
)
```

```
>>> from sage.all import *
>>> S = IntegralLattice(Matrix(ZZ, Integer(2), [Integer(2), Integer(3),␣
↪Integer(3), Integer(2)]))
>>> f = Matrix(ZZ, Integer(2), [Integer(0),Integer(1),-Integer(1),Integer(3)])
>>> S.orthogonal_group([f])                                            #␣
↪needs sage.libs.gap
Group of isometries with 1 generator (
[ 0  1]
[-1  3]
)
```

overlattice(*gens*)

Return the lattice spanned by this lattice and `gens`.

INPUT:

- `gens` – list of elements or a rational matrix

EXAMPLES:

```
sage: L = IntegralLattice(Matrix(ZZ, 2, 2, [2,0,0,2]))
sage: M = L.overlattice([vector([1,1])/2])
sage: M.gram_matrix()
[1 1]
[1 2]
```

```
>>> from sage.all import *
>>> L = IntegralLattice(Matrix(ZZ, Integer(2), Integer(2), [Integer(2),
↪Integer(0),Integer(0),Integer(2)]))
>>> M = L.overlattice([vector([Integer(1),Integer(1)])/Integer(2)])
>>> M.gram_matrix()
[1 1]
[1 2]
```

quadratic_form()

Return the quadratic form given by $q(x) = (x, x)$.

EXAMPLES:

```
sage: L = IntegralLattice("A2")                                                 #␣
↪needs sage.graphs
sage: q = L.quadratic_form(); q                                                 #␣
↪needs sage.graphs
Quadratic form in 2 variables over Integer Ring with coefficients:
[ 2 -2 ]
[ * 2 ]
```

```
>>> from sage.all import *
>>> L = IntegralLattice("A2")                                                 #␣
↪needs sage.graphs
>>> q = L.quadratic_form(); q                                                 #␣
↪needs sage.graphs
Quadratic form in 2 variables over Integer Ring with coefficients:
[ 2 -2 ]
[ * 2 ]
```

short_vectors(*n*, ***kwargs*)

Return the short vectors of length $< n$.

INPUT:

- `n` – integer
- further keyword arguments are passed on to `sage.quadratic_forms.short_vector_list_up_to_length()`

OUTPUT: list L where `L[k]` is the list of vectors of lengths k

EXAMPLES:

```
sage: A2 = IntegralLattice('A2')                                                #␣
↪needs sage.graphs
sage: A2.short_vectors(3)                                                       #␣
↪needs sage.graphs sage.libs.pari
[[(0, 0)], [], [(1, 1), (-1, -1), (0, 1), (0, -1), (1, 0), (-1, 0)]]
sage: A2.short_vectors(3, up_to_sign_flag=True)                                 #␣
↪needs sage.graphs sage.libs.pari
[[(0, 0)], [], [(1, 1), (0, 1), (1, 0)]]
```

```
>>> from sage.all import *
>>> A2 = IntegralLattice('A2')                                                #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
>>> A2.short_vectors(Integer(3))                                              ␣
↪      # needs sage.graphs sage.libs.pari
[[(0, 0)], [], [(1, 1), (-1, -1), (0, 1), (0, -1), (1, 0), (-1, 0)]]
>>> A2.short_vectors(Integer(3), up_to_sign_flag=True)                        ␣
↪      # needs sage.graphs sage.libs.pari
[[(0, 0)], [], [(1, 1), (0, 1), (1, 0)]]
```

signature()

Return the signature of this lattice, which is defined as the difference between the number of positive eigenvalues and the number of negative eigenvalues in the Gram matrix.

EXAMPLES:

```
sage: U = IntegralLattice("U")
sage: U.signature()
0
```

```
>>> from sage.all import *
>>> U = IntegralLattice("U")
>>> U.signature()
0
```

signature_pair()

Return the signature tuple (n_+, n_-) of this lattice.

Here n_+ (resp. n_-) is the number of positive (resp. negative) eigenvalues of the Gram matrix.

EXAMPLES:

```
sage: A2 = IntegralLattice("A2")                                              #␣
↪needs sage.graphs
sage: A2.signature_pair()                                                     #␣
↪needs sage.graphs
(2, 0)
```

```
>>> from sage.all import *
>>> A2 = IntegralLattice("A2")                                              #␣
↪needs sage.graphs
>>> A2.signature_pair()                                                     #␣
↪needs sage.graphs
(2, 0)
```

sublattice(*basis*)

Return the sublattice spanned by `basis`.

INPUT:

- `basis` – list of elements of this lattice

EXAMPLES:

```
sage: U = IntegralLattice("U")
sage: S = U.sublattice([vector([1,1])]); S
```

(continues on next page)

(continued from previous page)

```
Lattice of degree 2 and rank 1 over Integer Ring
Basis matrix:
[1 1]
Inner product matrix:
[0 1]
[1 0]
sage: U.sublattice([vector([1,-1])/2])
Traceback (most recent call last):
...
ValueError: lattices must be integral; use FreeQuadraticModule instead
sage: S.sublattice([vector([1,-1])])
Traceback (most recent call last):
...
ValueError: the basis (= [(1, -1)]) does not span a submodule
```

```
>>> from sage.all import *
>>> U = IntegralLattice("U")
>>> S = U.sublattice([vector([Integer(1),Integer(1)])]); S
Lattice of degree 2 and rank 1 over Integer Ring
Basis matrix:
[1 1]
Inner product matrix:
[0 1]
[1 0]
>>> U.sublattice([vector([Integer(1),-Integer(1)])/Integer(2)])
Traceback (most recent call last):
...
ValueError: lattices must be integral; use FreeQuadraticModule instead
>>> S.sublattice([vector([Integer(1),-Integer(1)])])
Traceback (most recent call last):
...
ValueError: the basis (= [(1, -1)]) does not span a submodule
```

tensor_product (*other*, *discard_basis=False*)

Return the tensor product of `self` and `other`.

INPUT:

- `other` – an integral lattice
- `discard_basis` – boolean (default: `False`); if `True`, then the lattice returned is equipped with the standard basis

EXAMPLES:

```
sage: # needs sage.graphs
sage: L = IntegralLattice("D3", [[1,-1,0], [0,1,-1]])
sage: L1 = L.tensor_product(L); L1
Lattice of degree 9 and rank 4 over Integer Ring
Basis matrix:
[ 1 -1  0 -1  1  0  0  0  0]
[ 0  1 -1  0 -1  1  0  0  0]
[ 0  0  0  1 -1  0 -1  1  0]
[ 0  0  0  0  1 -1  0 -1  1]
```

(continues on next page)

(continued from previous page)

```
Inner product matrix:
[ 4 -2 -2 -2  1  1 -2  1  1]
[-2  4  0  1 -2  0  1 -2  0]
[-2  0  4  1  0 -2  1  0 -2]
[-2  1  1  4 -2 -2  0  0  0]
[ 1 -2  0 -2  4  0  0  0  0]
[ 1  0 -2 -2  0  4  0  0  0]
[-2  1  1  0  0  0  4 -2 -2]
[ 1 -2  0  0  0  0 -2  4  0]
[ 1  0 -2  0  0  0 -2  0  4]
sage: L1.gram_matrix()
[ 36 -12 -12   4]
[-12  24   4  -8]
[-12   4  24  -8]
[  4  -8  -8  16]
sage: L2 = L.tensor_product(L, True); L2
Lattice of degree 4 and rank 4 over Integer Ring
Standard basis
Inner product matrix:
[ 36 -12 -12   4]
[-12  24   4  -8]
[-12   4  24  -8]
[  4  -8  -8  16]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> L = IntegralLattice("D3", [[Integer(1),-Integer(1),Integer(0)],␣
↪[Integer(0),Integer(1),-Integer(1)]])
>>> L1 = L.tensor_product(L); L1
Lattice of degree 9 and rank 4 over Integer Ring
Basis matrix:
[ 1 -1  0 -1  1  0  0  0  0]
[ 0  1 -1  0 -1  1  0  0  0]
[ 0  0  0  1 -1  0 -1  1  0]
[ 0  0  0  0  1 -1  0 -1  1]
Inner product matrix:
[ 4 -2 -2 -2  1  1 -2  1  1]
[-2  4  0  1 -2  0  1 -2  0]
[-2  0  4  1  0 -2  1  0 -2]
[-2  1  1  4 -2 -2  0  0  0]
[ 1 -2  0 -2  4  0  0  0  0]
[ 1  0 -2 -2  0  4  0  0  0]
[-2  1  1  0  0  0  4 -2 -2]
[ 1 -2  0  0  0  0 -2  4  0]
[ 1  0 -2  0  0  0 -2  0  4]
>>> L1.gram_matrix()
[ 36 -12 -12   4]
[-12  24   4  -8]
[-12   4  24  -8]
[  4  -8  -8  16]
>>> L2 = L.tensor_product(L, True); L2
Lattice of degree 4 and rank 4 over Integer Ring
```

(continues on next page)

(continued from previous page)

```
Standard basis
Inner product matrix:
[ 36 -12 -12   4]
[-12  24   4  -8]
[-12   4  24  -8]
[  4  -8  -8  16]
```

twist(*s*, *discard_basis=False*)

Return the lattice with inner product matrix scaled by `s`.

INPUT:

- `s` – nonzero integer
- `discard_basis` – boolean (default: `False`); if `True`, then the lattice returned is equipped with the standard basis

EXAMPLES:

```
sage: L = IntegralLattice("A4")                                                 #␣
↪needs sage.graphs
sage: L.twist(3)                                                                #␣
↪needs sage.graphs
Lattice of degree 4 and rank 4 over Integer Ring
Standard basis
Inner product matrix:
[ 6 -3  0  0]
[-3  6 -3  0]
[ 0 -3  6 -3]
[ 0  0 -3  6]
sage: L = IntegralLattice(3, [[2,1,0], [0,1,1]]); L
Lattice of degree 3 and rank 2 over Integer Ring
Basis matrix:
[2 1 0]
[0 1 1]
Standard scalar product
sage: L.twist(1)
Lattice of degree 3 and rank 2 over Integer Ring
Basis matrix:
[2 1 0]
[0 1 1]
Standard scalar product
sage: L.twist(1, True)
Lattice of degree 2 and rank 2 over Integer Ring
Standard basis
Inner product matrix:
[5 1]
[1 2]
```

```
>>> from sage.all import *
>>> L = IntegralLattice("A4")                                                   #␣
↪needs sage.graphs
>>> L.twist(Integer(3))                                                         ␣
↪     # needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
Lattice of degree 4 and rank 4 over Integer Ring
Standard basis
Inner product matrix:
[ 6 -3  0  0]
[-3  6 -3  0]
[ 0 -3  6 -3]
[ 0  0 -3  6]
>>> L = IntegralLattice(Integer(3), [[Integer(2),Integer(1),Integer(0)],␣
↪[Integer(0),Integer(1),Integer(1)]]); L
Lattice of degree 3 and rank 2 over Integer Ring
Basis matrix:
[2 1 0]
[0 1 1]
Standard scalar product
>>> L.twist(Integer(1))
Lattice of degree 3 and rank 2 over Integer Ring
Basis matrix:
[2 1 0]
[0 1 1]
Standard scalar product
>>> L.twist(Integer(1), True)
Lattice of degree 2 and rank 2 over Integer Ring
Standard basis
Inner product matrix:
[5 1]
[1 2]
```

sage.modules.free_quadratic_module_integer_symmetric.**IntegralLattice**(*data*, *basis=None*)

Return the integral lattice spanned by `basis` in the ambient space.

A lattice is a finitely generated free abelian group $L \cong \mathbf{Z}^r$ equipped with a non-degenerate, symmetric bilinear form $L \times L : \to \mathbf{Z}$. Here, lattices have an ambient quadratic space $\mathbf{Q}^n$ and a distinguished basis.

INPUT:

The input is a descriptor of the lattice and a (optional) basis. - `data` – can be one of the following:

- a symmetric matrix over the rationals – the inner product matrix
- an integer – the dimension for a Euclidean lattice
- a symmetric Cartan type or anything recognized by `CartanMatrix` (see also `Cartan types`) – for a root lattice
- the string `'U'` or `'H'` – for hyperbolic lattices
- `basis` – (optional) a matrix whose rows form a basis of the lattice, or a list of module elements forming a basis

OUTPUT:

A lattice in the ambient space defined by the inner_product_matrix. Unless specified, the basis of the lattice is the standard basis.

EXAMPLES:

```
sage: H5 = Matrix(ZZ, 2, [2,1,1,-2])
sage: IntegralLattice(H5)
Lattice of degree 2 and rank 2 over Integer Ring
Standard basis
Inner product matrix:
[ 2  1]
[ 1 -2]
```

```
>>> from sage.all import *
>>> H5 = Matrix(ZZ, Integer(2), [Integer(2),Integer(1),Integer(1),-Integer(2)])
>>> IntegralLattice(H5)
Lattice of degree 2 and rank 2 over Integer Ring
Standard basis
Inner product matrix:
[ 2  1]
[ 1 -2]
```

A basis can be specified too:

```
sage: IntegralLattice(H5, Matrix([1,1]))
Lattice of degree 2 and rank 1 over Integer Ring
Basis matrix:
[1 1]
Inner product matrix:
[ 2  1]
[ 1 -2]
```

```
>>> from sage.all import *
>>> IntegralLattice(H5, Matrix([Integer(1),Integer(1)]))
Lattice of degree 2 and rank 1 over Integer Ring
Basis matrix:
[1 1]
Inner product matrix:
[ 2  1]
[ 1 -2]
```

We can define a Euclidean lattice just by its dimension:

```
sage: IntegralLattice(3)
Lattice of degree 3 and rank 3 over Integer Ring
Standard basis
Standard scalar product
```

```
>>> from sage.all import *
>>> IntegralLattice(Integer(3))
Lattice of degree 3 and rank 3 over Integer Ring
Standard basis
Standard scalar product
```

Here is an example of the A_2 root lattice in Euclidean space:

```
sage: basis = Matrix([[1,-1,0], [0,1,-1]])
sage: A2 = IntegralLattice(3, basis)
```

(continues on next page)

(continued from previous page)

```
sage: A2
Lattice of degree 3 and rank 2 over Integer Ring
Basis matrix:
[ 1 -1  0]
[ 0  1 -1]
Standard scalar product
sage: A2.gram_matrix()
[ 2 -1]
[-1  2]
```

```
>>> from sage.all import *
>>> basis = Matrix([[Integer(1),-Integer(1),Integer(0)], [Integer(0),Integer(1),-
↪Integer(1)]])
>>> A2 = IntegralLattice(Integer(3), basis)
>>> A2
Lattice of degree 3 and rank 2 over Integer Ring
Basis matrix:
[ 1 -1  0]
[ 0  1 -1]
Standard scalar product
>>> A2.gram_matrix()
[ 2 -1]
[-1  2]
```

We use `'U'` or `'H'` for defining a hyperbolic lattice:

```
sage: L1 = IntegralLattice('U')
sage: L1
Lattice of degree 2 and rank 2 over Integer Ring
Standard basis
Inner product matrix:
[0 1]
[1 0]
sage: L1 == IntegralLattice('H')
True
```

```
>>> from sage.all import *
>>> L1 = IntegralLattice('U')
>>> L1
Lattice of degree 2 and rank 2 over Integer Ring
Standard basis
Inner product matrix:
[0 1]
[1 0]
>>> L1 == IntegralLattice('H')
True
```

We can construct root lattices by specifying their type (see `Cartan types` and `CartanMatrix`):

```
sage: # needs sage.graphs
sage: IntegralLattice(["E", 7])
Lattice of degree 7 and rank 7 over Integer Ring
```

(continues on next page)

(continued from previous page)

```
Standard basis
Inner product matrix:
[ 2  0 -1  0  0  0  0]
[ 0  2  0 -1  0  0  0]
[-1  0  2 -1  0  0  0]
[ 0 -1 -1  2 -1  0  0]
[ 0  0  0 -1  2 -1  0]
[ 0  0  0  0 -1  2 -1]
[ 0  0  0  0  0 -1  2]
sage: IntegralLattice(["A", 2])
Lattice of degree 2 and rank 2 over Integer Ring
Standard basis
Inner product matrix:
[ 2 -1]
[-1  2]
sage: IntegralLattice("D3")
Lattice of degree 3 and rank 3 over Integer Ring
Standard basis
Inner product matrix:
[ 2 -1 -1]
[-1  2  0]
[-1  0  2]
sage: IntegralLattice(["D", 4])
Lattice of degree 4 and rank 4 over Integer Ring
Standard basis
Inner product matrix:
[ 2 -1  0  0]
[-1  2 -1 -1]
[ 0 -1  2  0]
[ 0 -1  0  2]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> IntegralLattice(["E", Integer(7)])
Lattice of degree 7 and rank 7 over Integer Ring
Standard basis
Inner product matrix:
[ 2  0 -1  0  0  0  0]
[ 0  2  0 -1  0  0  0]
[-1  0  2 -1  0  0  0]
[ 0 -1 -1  2 -1  0  0]
[ 0  0  0 -1  2 -1  0]
[ 0  0  0  0 -1  2 -1]
[ 0  0  0  0  0 -1  2]
>>> IntegralLattice(["A", Integer(2)])
Lattice of degree 2 and rank 2 over Integer Ring
Standard basis
Inner product matrix:
[ 2 -1]
[-1  2]
>>> IntegralLattice("D3")
Lattice of degree 3 and rank 3 over Integer Ring
```

(continues on next page)

(continued from previous page)

```
Standard basis
Inner product matrix:
[ 2 -1 -1]
[-1  2  0]
[-1  0  2]
>>> IntegralLattice(["D", Integer(4)])
Lattice of degree 4 and rank 4 over Integer Ring
Standard basis
Inner product matrix:
[ 2 -1  0  0]
[-1  2 -1 -1]
[ 0 -1  2  0]
[ 0 -1  0  2]
```

We can specify a basis as well:

```
sage: G = Matrix(ZZ, 2, [0,1,1,0])
sage: B = [vector([1,1])]
sage: IntegralLattice(G, basis=B)
Lattice of degree 2 and rank 1 over Integer Ring
Basis matrix:
[1 1]
Inner product matrix:
[0 1]
[1 0]
sage: IntegralLattice(["A", 3], [[1,1,1]])                                  #␣
↪needs sage.graphs
Lattice of degree 3 and rank 1 over Integer Ring
Basis matrix:
[1 1 1]
Inner product matrix:
[ 2 -1  0]
[-1  2 -1]
[ 0 -1  2]
sage: IntegralLattice(4, [[1,1,1,1]])
Lattice of degree 4 and rank 1 over Integer Ring
Basis matrix:
[1 1 1 1]
Standard scalar product
sage: IntegralLattice("A2", [[1,1]])                                        #␣
↪needs sage.graphs
Lattice of degree 2 and rank 1 over Integer Ring
Basis matrix:
[1 1]
Inner product matrix:
[ 2 -1]
[-1  2]
```

```
>>> from sage.all import *
>>> G = Matrix(ZZ, Integer(2), [Integer(0),Integer(1),Integer(1),Integer(0)])
>>> B = [vector([Integer(1),Integer(1)])]
>>> IntegralLattice(G, basis=B)
```

(continues on next page)

(continued from previous page)

```
Lattice of degree 2 and rank 1 over Integer Ring
Basis matrix:
[1 1]
Inner product matrix:
[0 1]
[1 0]
>>> IntegralLattice(["A", Integer(3)], [[Integer(1),Integer(1),Integer(1)]])
↪                                  # needs sage.graphs
Lattice of degree 3 and rank 1 over Integer Ring
Basis matrix:
[1 1 1]
Inner product matrix:
[ 2 -1  0]
[-1  2 -1]
[ 0 -1  2]
>>> IntegralLattice(Integer(4), [[Integer(1),Integer(1),Integer(1),Integer(1)]])
Lattice of degree 4 and rank 1 over Integer Ring
Basis matrix:
[1 1 1 1]
Standard scalar product
>>> IntegralLattice("A2", [[Integer(1),Integer(1)]])
↪               # needs sage.graphs
Lattice of degree 2 and rank 1 over Integer Ring
Basis matrix:
[1 1]
Inner product matrix:
[ 2 -1]
[-1  2]
```

sage.modules.free_quadratic_module_integer_symmetric.**IntegralLatticeDirectSum**(*Lattices*, *return_embeddings=False*)

Return the direct sum of the lattices contained in the list `Lattices`.

INPUT:

- `Lattices` – list of lattices `[L_1,...,L_n]`
- `return_embeddings` – boolean (default: `False`)

OUTPUT:

The direct sum of the L_i if `return_embeddings` is `False` or the tuple `[L, phi]` where L is the direct sum of L_i and `phi` is the list of embeddings from L_i to L.

EXAMPLES:

```
sage: # needs sage.graphs
sage: from sage.modules.free_quadratic_module_integer_symmetric import␣
↪IntegralLatticeDirectSum
sage: L1 = IntegralLattice("D4")
sage: L2 = IntegralLattice("A3", [[1, 1, 2]])
sage: L3 = IntegralLattice("A4", [[0, 1, 1, 2], [1, 2, 3, 1]])
sage: Lattices = [L1, L2, L3]
```

(continues on next page)

(continued from previous page)

```
sage: IntegralLatticeDirectSum([L1, L2, L3])
Lattice of degree 11 and rank 7 over Integer Ring
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 1 2 0 0 0 0]
[0 0 0 0 0 0 0 0 1 1 2]
[0 0 0 0 0 0 0 1 2 3 1]
Inner product matrix:
[ 2 -1  0  0  0  0  0  0  0  0  0]
[-1  2 -1 -1  0  0  0  0  0  0  0]
[ 0 -1  2  0  0  0  0  0  0  0  0]
[ 0 -1  0  2  0  0  0  0  0  0  0]
[ 0  0  0  0  2 -1  0  0  0  0  0]
[ 0  0  0  0 -1  2 -1  0  0  0  0]
[ 0  0  0  0  0 -1  2  0  0  0  0]
[ 0  0  0  0  0  0  0  2 -1  0  0]
[ 0  0  0  0  0  0  0 -1  2 -1  0]
[ 0  0  0  0  0  0  0  0 -1  2 -1]
[ 0  0  0  0  0  0  0  0  0 -1  2]
sage: [L, phi] = IntegralLatticeDirectSum([L1, L2, L3], True)
sage: LL3 = L.sublattice(phi[2].image().basis_matrix())
sage: L3.discriminant() == LL3.discriminant()
True
sage: x = L3([1, 2, 3, 1])
sage: phi[2](x).inner_product(phi[2](x)) == x.inner_product(x)
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> from sage.modules.free_quadratic_module_integer_symmetric import␣
↪IntegralLatticeDirectSum
>>> L1 = IntegralLattice("D4")
>>> L2 = IntegralLattice("A3", [[Integer(1), Integer(1), Integer(2)]])
>>> L3 = IntegralLattice("A4", [[Integer(0), Integer(1), Integer(1), Integer(2)],␣
↪[Integer(1), Integer(2), Integer(3), Integer(1)]])
>>> Lattices = [L1, L2, L3]
>>> IntegralLatticeDirectSum([L1, L2, L3])
Lattice of degree 11 and rank 7 over Integer Ring
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 1 2 0 0 0 0]
[0 0 0 0 0 0 0 0 1 1 2]
[0 0 0 0 0 0 0 1 2 3 1]
Inner product matrix:
[ 2 -1  0  0  0  0  0  0  0  0  0]
[-1  2 -1 -1  0  0  0  0  0  0  0]
```

(continues on next page)

(continued from previous page)

```
[ 0 -1  2  0  0  0  0  0  0  0  0]
[ 0 -1  0  2  0  0  0  0  0  0  0]
[ 0  0  0  0  2 -1  0  0  0  0  0]
[ 0  0  0  0 -1  2 -1  0  0  0  0]
[ 0  0  0  0  0 -1  2  0  0  0  0]
[ 0  0  0  0  0  0  0  2 -1  0  0]
[ 0  0  0  0  0  0  0 -1  2 -1  0]
[ 0  0  0  0  0  0  0  0 -1  2 -1]
[ 0  0  0  0  0  0  0  0  0 -1  2]
>>> [L, phi] = IntegralLatticeDirectSum([L1, L2, L3], True)
>>> LL3 = L.sublattice(phi[Integer(2)].image().basis_matrix())
>>> L3.discriminant() == LL3.discriminant()
True
>>> x = L3([Integer(1), Integer(2), Integer(3), Integer(1)])
>>> phi[Integer(2)](x).inner_product(phi[Integer(2)](x)) == x.inner_product(x)
True
```

sage.modules.free_quadratic_module_integer_symmetric.**IntegralLatticeGluing**(*Lattices*, *glue*, *return_embeddings=False*)

Return an overlattice of the direct sum as defined by `glue`.

INPUT:

- `Lattices` – list of lattices $[L_1, ..., L_n]$
- `glue` – list where the elements are lists in the form $[g_1, ..., g_n]$; here g_i is an element of the discriminant group of L_i`$and the overlattice is spanned by the additional$`$[sum(g) for g in glue]$`
- `return_embeddings` – boolean (default: `False`)

OUTPUT:

The glued lattice given by L_i and `glue` if `return_embeddings` is `False` or the tuple `[L, phi]` where L is the glued lattice and `phi` the list of embeddings from L_i to L

EXAMPLES:

A single lattice can be glued. This is the same as taking an overlattice:

```
sage: from sage.modules.free_quadratic_module_integer_symmetric import␣
↪IntegralLatticeGluing
sage: L1 = IntegralLattice(matrix([[4]]))
sage: g1 = L1.discriminant_group().gens()[0]
sage: glue = [[2 * g1]]
sage: L = IntegralLatticeGluing([L1], glue)
sage: L
Lattice of degree 1 and rank 1 over Integer Ring
Basis matrix:
[1/2]
Inner product matrix:
[4]
sage: L.gram_matrix()
[1]
sage: IntegralLatticeGluing([L1], glue, return_embeddings=True)
[Lattice of degree 1 and rank 1 over Integer Ring
```

(continues on next page)

(continued from previous page)

```
 Basis matrix:
 [1/2]
 Inner product matrix:
 [4], [Free module morphism defined by the matrix
  [2]
  Domain: Lattice of degree 1 and rank 1 over Integer Ring
  Standard basis
  Inner product matrix:
  [4]
  Codomain: Lattice of degree 1 and rank 1 over Integer Ring
  Basis matrix:
  [1/2]
  Inner product matrix:
  [4]]]

sage: # needs sage.graphs
sage: L1 = IntegralLattice([[2]])
sage: L2 = IntegralLattice([[2]])
sage: AL1 = L1.discriminant_group()
sage: AL2 = L2.discriminant_group()
sage: AL1
Finite quadratic module over Integer Ring with invariants (2,)
Gram matrix of the quadratic form with values in Q/2Z:
[1/2]
sage: g1 = L1.discriminant_group().gens()[0]
sage: g2 = L2.discriminant_group().gens()[0]
sage: glue = [[g1, g2]]
sage: IntegralLatticeGluing([L1, L2], glue)
Lattice of degree 2 and rank 2 over Integer Ring
Basis matrix:
[1/2 1/2]
[  0   1]
Inner product matrix:
[2 0]
[0 2]

sage: # needs sage.graphs
sage: L1 = IntegralLattice("A4")
sage: L2 = IntegralLattice("A4")
sage: g1 = L1.discriminant_group().gens()[0]
sage: g2 = L2.discriminant_group().gens()[0]
sage: glue = [[g1, 2 * g2]]
sage: [V, phi] = IntegralLatticeGluing([L1, L2], glue, True)
sage: V
Lattice of degree 8 and rank 8 over Integer Ring
Basis matrix:
[1/5 2/5 3/5 4/5 2/5 4/5 1/5 3/5]
[  0   1   0   0   0   0   0   0]
[  0   0   1   0   0   0   0   0]
[  0   0   0   1   0   0   0   0]
[  0   0   0   0   1   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   1   0]
[  0   0   0   0   0   0   0   1]
Inner product matrix:
[ 2 -1  0  0  0  0  0  0]
[-1  2 -1  0  0  0  0  0]
[ 0 -1  2 -1  0  0  0  0]
[ 0  0 -1  2  0  0  0  0]
[ 0  0  0  0  2 -1  0  0]
[ 0  0  0  0 -1  2 -1  0]
[ 0  0  0  0  0 -1  2 -1]
[ 0  0  0  0  0  0 -1  2]
sage: V.sublattice(phi[0].image().basis_matrix())
Lattice of degree 8 and rank 4 over Integer Ring
Basis matrix:
[1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0]
Inner product matrix:
[ 2 -1  0  0  0  0  0  0]
[-1  2 -1  0  0  0  0  0]
[ 0 -1  2 -1  0  0  0  0]
[ 0  0 -1  2  0  0  0  0]
[ 0  0  0  0  2 -1  0  0]
[ 0  0  0  0 -1  2 -1  0]
[ 0  0  0  0  0 -1  2 -1]
[ 0  0  0  0  0  0 -1  2]
```

```
>>> from sage.all import *
>>> from sage.modules.free_quadratic_module_integer_symmetric import␣
↪IntegralLatticeGluing
>>> L1 = IntegralLattice(matrix([[Integer(4)]]))
>>> g1 = L1.discriminant_group().gens()[Integer(0)]
>>> glue = [[Integer(2) * g1]]
>>> L = IntegralLatticeGluing([L1], glue)
>>> L
Lattice of degree 1 and rank 1 over Integer Ring
Basis matrix:
[1/2]
Inner product matrix:
[4]
>>> L.gram_matrix()
[1]
>>> IntegralLatticeGluing([L1], glue, return_embeddings=True)
[Lattice of degree 1 and rank 1 over Integer Ring
 Basis matrix:
 [1/2]
 Inner product matrix:
 [4], [Free module morphism defined by the matrix
  [2]
  Domain: Lattice of degree 1 and rank 1 over Integer Ring
  Standard basis
```

(continues on next page)

(continued from previous page)

```
  Inner product matrix:
  [4]
  Codomain: Lattice of degree 1 and rank 1 over Integer Ring
  Basis matrix:
  [1/2]
  Inner product matrix:
  [4]]]

>>> # needs sage.graphs
>>> L1 = IntegralLattice([[Integer(2)]])
>>> L2 = IntegralLattice([[Integer(2)]])
>>> AL1 = L1.discriminant_group()
>>> AL2 = L2.discriminant_group()
>>> AL1
Finite quadratic module over Integer Ring with invariants (2,)
Gram matrix of the quadratic form with values in Q/2Z:
[1/2]
>>> g1 = L1.discriminant_group().gens()[Integer(0)]
>>> g2 = L2.discriminant_group().gens()[Integer(0)]
>>> glue = [[g1, g2]]
>>> IntegralLatticeGluing([L1, L2], glue)
Lattice of degree 2 and rank 2 over Integer Ring
Basis matrix:
[1/2 1/2]
[  0   1]
Inner product matrix:
[2 0]
[0 2]

>>> # needs sage.graphs
>>> L1 = IntegralLattice("A4")
>>> L2 = IntegralLattice("A4")
>>> g1 = L1.discriminant_group().gens()[Integer(0)]
>>> g2 = L2.discriminant_group().gens()[Integer(0)]
>>> glue = [[g1, Integer(2) * g2]]
>>> [V, phi] = IntegralLatticeGluing([L1, L2], glue, True)
>>> V
Lattice of degree 8 and rank 8 over Integer Ring
Basis matrix:
[1/5 2/5 3/5 4/5 2/5 4/5 1/5 3/5]
[  0   1   0   0   0   0   0   0]
[  0   0   1   0   0   0   0   0]
[  0   0   0   1   0   0   0   0]
[  0   0   0   0   1   0   0   0]
[  0   0   0   0   0   1   0   0]
[  0   0   0   0   0   0   1   0]
[  0   0   0   0   0   0   0   1]
Inner product matrix:
[ 2 -1  0  0  0  0  0  0]
[-1  2 -1  0  0  0  0  0]
[ 0 -1  2 -1  0  0  0  0]
[ 0  0 -1  2  0  0  0  0]
```

(continues on next page)

(continued from previous page)

```
[ 0  0  0  0  2 -1  0  0]
[ 0  0  0  0 -1  2 -1  0]
[ 0  0  0  0  0 -1  2 -1]
[ 0  0  0  0  0  0 -1  2]
>>> V.sublattice(phi[Integer(0)].image().basis_matrix())
Lattice of degree 8 and rank 4 over Integer Ring
Basis matrix:
[1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0]
Inner product matrix:
[ 2 -1  0  0  0  0  0  0]
[-1  2 -1  0  0  0  0  0]
[ 0 -1  2 -1  0  0  0  0]
[ 0  0 -1  2  0  0  0  0]
[ 0  0  0  0  2 -1  0  0]
[ 0  0  0  0 -1  2 -1  0]
[ 0  0  0  0  0 -1  2 -1]
[ 0  0  0  0  0  0 -1  2]
```

Different gluings can be composed:

```
sage: # needs sage.graphs
sage: D4 = IntegralLattice("D4")
sage: D4.discriminant_group()
Finite quadratic module over Integer Ring with invariants (2, 2)
Gram matrix of the quadratic form with values in Q/2Z:
[  1 1/2]
[1/2   1]
sage: L2 = IntegralLattice(2 * matrix.identity(2))
sage: L2.discriminant_group()
Finite quadratic module over Integer Ring with invariants (2, 2)
Gram matrix of the quadratic form with values in Q/2Z:
[1/2   0]
[  0 1/2]
sage: g1 = D4.discriminant_group().gens()[0]
sage: g2 = L2.discriminant_group().gens()[0] + L2.discriminant_group().gens()[1]
sage: D6, phi = IntegralLatticeGluing([D4, L2], [[g1, g2]], True)
sage: AD6 = D6.discriminant_group()
sage: AD6.normal_form()
Finite quadratic module over Integer Ring with invariants (2, 2)
Gram matrix of the quadratic form with values in Q/2Z:
[3/2   0]
[  0 3/2]
sage: f1, g1 = AD6.normal_form().gens()
sage: f2, g2 = L2.discriminant_group().gens()
sage: E8, psi = IntegralLatticeGluing([D6, L2], [[f1, f2], [g1, g2]], True)
sage: D4embed = E8.sublattice(psi[0](phi[0].image()).basis_matrix())
sage: x = D4([1, 0, 0, 0])
sage: psi[0](phi[0](x)).inner_product(psi[0](phi[0](x))) == x.inner_product(x)
True
```

(continues on next page)

(continued from previous page)

```
sage: D4embed
Lattice of degree 8 and rank 4 over Integer Ring
Basis matrix:
[1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0]
Inner product matrix:
[ 2 -1  0  0  0  0  0  0]
[-1  2 -1 -1  0  0  0  0]
[ 0 -1  2  0  0  0  0  0]
[ 0 -1  0  2  0  0  0  0]
[ 0  0  0  0  2  0  0  0]
[ 0  0  0  0  0  2  0  0]
[ 0  0  0  0  0  0  2  0]
[ 0  0  0  0  0  0  0  2]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> D4 = IntegralLattice("D4")
>>> D4.discriminant_group()
Finite quadratic module over Integer Ring with invariants (2, 2)
Gram matrix of the quadratic form with values in Q/2Z:
[  1 1/2]
[1/2   1]
>>> L2 = IntegralLattice(Integer(2) * matrix.identity(Integer(2)))
>>> L2.discriminant_group()
Finite quadratic module over Integer Ring with invariants (2, 2)
Gram matrix of the quadratic form with values in Q/2Z:
[1/2   0]
[  0 1/2]
>>> g1 = D4.discriminant_group().gens()[Integer(0)]
>>> g2 = L2.discriminant_group().gens()[Integer(0)] + L2.discriminant_group().
↪gens()[Integer(1)]
>>> D6, phi = IntegralLatticeGluing([D4, L2], [[g1, g2]], True)
>>> AD6 = D6.discriminant_group()
>>> AD6.normal_form()
Finite quadratic module over Integer Ring with invariants (2, 2)
Gram matrix of the quadratic form with values in Q/2Z:
[3/2   0]
[  0 3/2]
>>> f1, g1 = AD6.normal_form().gens()
>>> f2, g2 = L2.discriminant_group().gens()
>>> E8, psi = IntegralLatticeGluing([D6, L2], [[f1, f2], [g1, g2]], True)
>>> D4embed = E8.sublattice(psi[Integer(0)](phi[Integer(0)].image()).basis_
↪matrix())
>>> x = D4([Integer(1), Integer(0), Integer(0), Integer(0)])
>>> psi[Integer(0)](phi[Integer(0)](x)).inner_
↪product(psi[Integer(0)](phi[Integer(0)](x))) == x.inner_product(x)
True
>>> D4embed
Lattice of degree 8 and rank 4 over Integer Ring
```

(continues on next page)

(continued from previous page)

```
Basis matrix:
[1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0]
Inner product matrix:
[ 2 -1  0  0  0  0  0  0]
[-1  2 -1 -1  0  0  0  0]
[ 0 -1  2  0  0  0  0  0]
[ 0 -1  0  2  0  0  0  0]
[ 0  0  0  0  2  0  0  0]
[ 0  0  0  0  0  2  0  0]
[ 0  0  0  0  0  0  2  0]
[ 0  0  0  0  0  0  0  2]
```

The input may be a list of three or more lattices:

```
sage: # needs sage.graphs
sage: A7 = IntegralLattice("A7")
sage: D5 = IntegralLattice("D5")
sage: gA7 = A7.discriminant_group().gens()[0]
sage: gD5 = D5.discriminant_group().gens()[0]
sage: [L, phi] = IntegralLatticeGluing([A7, A7, D5, D5],
....:                           [[gA7, gA7, gD5, 2 * gD5],
....:                           [gA7, 7 * gA7, 2 * gD5, gD5]], True)
sage: L.determinant()
1
sage: B = phi[0].matrix()
sage: B*L.gram_matrix()*B.transpose() == A7.gram_matrix()
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> A7 = IntegralLattice("A7")
>>> D5 = IntegralLattice("D5")
>>> gA7 = A7.discriminant_group().gens()[Integer(0)]
>>> gD5 = D5.discriminant_group().gens()[Integer(0)]
>>> [L, phi] = IntegralLatticeGluing([A7, A7, D5, D5],
...                           [[gA7, gA7, gD5, Integer(2) * gD5],
...                           [gA7, Integer(7) * gA7, Integer(2) * gD5, gD5]],␣
↪True)
>>> L.determinant()
1
>>> B = phi[Integer(0)].matrix()
>>> B*L.gram_matrix()*B.transpose() == A7.gram_matrix()
True
```

The gluing takes place in the direct sum of the respective ambient spaces:

```
sage: # needs sage.graphs
sage: L1 = IntegralLattice("D4", [[1, 1, 0, 0], [0, 1, 1, 0]])
sage: L2 = IntegralLattice("E6", [[0, 2, 0, 0, 0, 0], [0, 0, 0, 0, 1, 1]])
```

(continues on next page)

(continued from previous page)

```
sage: [f1, f2] = L1.discriminant_group().gens()
sage: [g1, g2] = L2.discriminant_group().gens()
sage: [L, phi] = IntegralLatticeGluing([L1, L2],
....:                                  [[f1, g1], [f2, 2 * g2]], True)
sage: phi[0]
Free module morphism defined by the matrix
[ 2  2 -2 -1]
[ 0  2 -1  0]
Domain: Lattice of degree 4 and rank 2 over Integer Ring
Basis matrix:
[1 1 0 0]
[0 1 1 0]
Inner product matrix:
[ 2 -1  0  0]
[-1  2 -1 -1]
[ 0 -1  2  0]
[ 0 -1  0  2]
Codomain: Lattice of degree 10 and rank 4 over Integer Ring
Basis matrix:
[ 1/2    0 -1/2    0    0  1/2    0    0  1/2  1/2]
[   0  1/2  1/2    0    0  1/2    0    0    0    0]
[   0    0    0    0    0    1    0    0    0    0]
[   0    0    0    0    0    0    0    0    1    1]
Inner product matrix:
[ 2 -1  0  0  0  0  0  0  0  0]
[-1  2 -1 -1  0  0  0  0  0  0]
[ 0 -1  2  0  0  0  0  0  0  0]
[ 0 -1  0  2  0  0  0  0  0  0]
[ 0  0  0  0  2  0 -1  0  0  0]
[ 0  0  0  0  0  2  0 -1  0  0]
[ 0  0  0  0 -1  0  2 -1  0  0]
[ 0  0  0  0  0 -1 -1  2 -1  0]
[ 0  0  0  0  0  0  0 -1  2 -1]
[ 0  0  0  0  0  0  0  0 -1  2]
sage: B = phi[0].matrix()
sage: B * L.gram_matrix() * B.transpose() == L1.gram_matrix()
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> L1 = IntegralLattice("D4", [[Integer(1), Integer(1), Integer(0), Integer(0)],␣
↪[Integer(0), Integer(1), Integer(1), Integer(0)]])
>>> L2 = IntegralLattice("E6", [[Integer(0), Integer(2), Integer(0), Integer(0),␣
↪Integer(0), Integer(0)], [Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(1), Integer(1)]])
>>> [f1, f2] = L1.discriminant_group().gens()
>>> [g1, g2] = L2.discriminant_group().gens()
>>> [L, phi] = IntegralLatticeGluing([L1, L2],
...                                  [[f1, g1], [f2, Integer(2) * g2]], True)
>>> phi[Integer(0)]
Free module morphism defined by the matrix
[ 2  2 -2 -1]
```

(continues on next page)

(continued from previous page)

```
[ 0  2 -1  0]
Domain: Lattice of degree 4 and rank 2 over Integer Ring
Basis matrix:
[1 1 0 0]
[0 1 1 0]
Inner product matrix:
[ 2 -1  0  0]
[-1  2 -1 -1]
[ 0 -1  2  0]
[ 0 -1  0  2]
Codomain: Lattice of degree 10 and rank 4 over Integer Ring
Basis matrix:
[ 1/2    0 -1/2    0    0  1/2    0    0  1/2  1/2]
[   0  1/2  1/2    0    0  1/2    0    0    0    0]
[   0    0    0    0    0    1    0    0    0    0]
[   0    0    0    0    0    0    0    0    1    1]
Inner product matrix:
[ 2 -1  0  0  0  0  0  0  0  0]
[-1  2 -1 -1  0  0  0  0  0  0]
[ 0 -1  2  0  0  0  0  0  0  0]
[ 0 -1  0  2  0  0  0  0  0  0]
[ 0  0  0  0  2  0 -1  0  0  0]
[ 0  0  0  0  0  2  0 -1  0  0]
[ 0  0  0  0 -1  0  2 -1  0  0]
[ 0  0  0  0  0 -1 -1  2 -1  0]
[ 0  0  0  0  0  0  0 -1  2 -1]
[ 0  0  0  0  0  0  0  0 -1  2]
>>> B = phi[Integer(0)].matrix()
>>> B * L.gram_matrix() * B.transpose() == L1.gram_matrix()
True
```

sage.modules.free_quadratic_module_integer_symmetric.**local_modification**(*M*, *G*, *p*, *check=True*)

Return a local modification of M that matches G at p.

INPUT:

- `M` – a $\mathbf{Z}_p$-maximal lattice
- `G` – the gram matrix of a lattice isomorphic to M over $\mathbf{Q}_p$
- `p` – a prime number

OUTPUT:

an integral lattice M' in the ambient space of M such that M and M' are locally equal at all completions except at p where M' is locally equivalent to the lattice with gram matrix G

EXAMPLES:

```
sage: # needs sage.graphs sage.libs.pari
sage: from sage.modules.free_quadratic_module_integer_symmetric import local_
↪modification
sage: L = IntegralLattice("A3").twist(15)
sage: M = L.maximal_overlattice()
```

(continues on next page)

(continued from previous page)

```
sage: for p in prime_divisors(L.determinant()):
....:     M = local_modification(M, L.gram_matrix(), p)
sage: M.genus() == L.genus()
True
sage: L = IntegralLattice("D4").twist(3*4)
sage: M = L.maximal_overlattice()
sage: local_modification(M, L.gram_matrix(), 2)
Lattice of degree 4 and rank 4 over Integer Ring
Basis matrix:
[1/3   0 2/3 2/3]
[  0 1/3   0 2/3]
[  0   0   1   0]
[  0   0   0   1]
Inner product matrix:
[ 24 -12   0   0]
[-12  24 -12 -12]
[  0 -12  24   0]
[  0 -12   0  24]
```

```
>>> from sage.all import *
>>> # needs sage.graphs sage.libs.pari
>>> from sage.modules.free_quadratic_module_integer_symmetric import local_
↪modification
>>> L = IntegralLattice("A3").twist(Integer(15))
>>> M = L.maximal_overlattice()
>>> for p in prime_divisors(L.determinant()):
...     M = local_modification(M, L.gram_matrix(), p)
>>> M.genus() == L.genus()
True
>>> L = IntegralLattice("D4").twist(Integer(3)*Integer(4))
>>> M = L.maximal_overlattice()
>>> local_modification(M, L.gram_matrix(), Integer(2))
Lattice of degree 4 and rank 4 over Integer Ring
Basis matrix:
[1/3   0 2/3 2/3]
[  0 1/3   0 2/3]
[  0   0   1   0]
[  0   0   0   1]
Inner product matrix:
[ 24 -12   0   0]
[-12  24 -12 -12]
[  0 -12  24   0]
[  0 -12   0  24]
```

6.4 Finite $\mathbb{Z}$-modules with bilinear and quadratic forms

AUTHORS:

- Simon Brandhorst (2017-09): First created

sage.modules.torsion_quadratic_module.**TorsionQuadraticForm**(*q*)

Create a torsion quadratic form module from a rational matrix.

The resulting quadratic form takes values in $\mathbf{Q}/\mathbf{Z}$ or $\mathbf{Q}/2\mathbf{Z}$ (depending on `q`). If it takes values modulo 2, then it is non-degenerate. In any case the bilinear form is non-degenerate.

INPUT:

- `q` – a symmetric rational matrix

EXAMPLES:

```
sage: q1 = Matrix(QQ, 2, [1,1/2,1/2,1])
sage: TorsionQuadraticForm(q1)
Finite quadratic module over Integer Ring with invariants (2, 2)
Gram matrix of the quadratic form with values in Q/2Z:
[  1 1/2]
[1/2   1]
```

```
>>> from sage.all import *
>>> q1 = Matrix(QQ, Integer(2), [Integer(1),Integer(1)/Integer(2),Integer(1)/
↪Integer(2),Integer(1)])
>>> TorsionQuadraticForm(q1)
Finite quadratic module over Integer Ring with invariants (2, 2)
Gram matrix of the quadratic form with values in Q/2Z:
[  1 1/2]
[1/2   1]
```

In the following example the quadratic form is degenerate. But the bilinear form is still non-degenerate:

```
sage: q2 = diagonal_matrix(QQ, [1/4,1/3])
sage: TorsionQuadraticForm(q2)
Finite quadratic module over Integer Ring with invariants (12,)
Gram matrix of the quadratic form with values in Q/Z:
[7/12]
```

```
>>> from sage.all import *
>>> q2 = diagonal_matrix(QQ, [Integer(1)/Integer(4),Integer(1)/Integer(3)])
>>> TorsionQuadraticForm(q2)
Finite quadratic module over Integer Ring with invariants (12,)
Gram matrix of the quadratic form with values in Q/Z:
[7/12]
```

class sage.modules.torsion_quadratic_module.**TorsionQuadraticModule**(*V*, *W*, *gens*, *modulus*, *modulus_qf*)

Bases: *FGP_Module_class*, CachedRepresentation

Finite quotients with a bilinear and a quadratic form.

Let V be a symmetric *FreeQuadraticModule* and $W \subseteq V$ a submodule of the same rank as V. The quotient V/W is a torsion quadratic module. It inherits a bilinear form b and a quadratic form q.

$b : V \times V \to \mathbf{Q}/m\mathbf{Z}$, where $m\mathbf{Z} = (V, W)$ and $b(x, y) = (x, y) + m\mathbf{Z}$

$q : V \to \mathbf{Q}/n\mathbf{Z}$, where $n\mathbf{Z} = 2(V, W) + \mathbf{Z}\{(w, w)|w \in W\}$

INPUT:

- `V` – a *FreeModule* with a symmetric inner product matrix
- `W` – a submodule of `V` of the same rank as `V`

- `check` – boolean (default: `True`)
- `modulus` – a rational number dividing m (default: m); the inner product b is defined in $\mathbf{Q}/$ `modulus` $\mathbf{Z}$
- `modulus_qf` – a rational number dividing n (default: n); the quadratic form q is defined in $\mathbf{Q}/$ `modulus_qf` $\mathbf{Z}$

EXAMPLES:

```
sage: from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
sage: V = FreeModule(ZZ, 3)
sage: T = TorsionQuadraticModule(V, 5*V); T
Finite quadratic module over Integer Ring with invariants (5, 5, 5)
Gram matrix of the quadratic form with values in Q/5Z:
[1 0 0]
[0 1 0]
[0 0 1]
```

```
>>> from sage.all import *
>>> from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
>>> V = FreeModule(ZZ, Integer(3))
>>> T = TorsionQuadraticModule(V, Integer(5)*V); T
Finite quadratic module over Integer Ring with invariants (5, 5, 5)
Gram matrix of the quadratic form with values in Q/5Z:
[1 0 0]
[0 1 0]
[0 0 1]
```

Element

alias of *TorsionQuadraticModuleElement*

all_submodules()

Return a list of all submodules of `self`.

Warning

This method creates all submodules in memory. The number of submodules grows rapidly with the number of generators. For example consider a vector space of dimension n over a finite field of prime order p. The number of subspaces is (very) roughly $p^{(n^2-n)/2}$.

EXAMPLES:

```
sage: D = IntegralLattice("D4").discriminant_group()                    #␣
↪needs sage.combinat
sage: D.all_submodules()                                                #␣
↪needs sage.combinat
[Finite quadratic module over Integer Ring with invariants ()
  Gram matrix of the quadratic form with values in Q/2Z:
  [],
 Finite quadratic module over Integer Ring with invariants (2,)
  Gram matrix of the quadratic form with values in Q/2Z:
  [1],
 Finite quadratic module over Integer Ring with invariants (2,)
  Gram matrix of the quadratic form with values in Q/2Z:
```

(continues on next page)

(continued from previous page)

```
 [1],
Finite quadratic module over Integer Ring with invariants (2,)
 Gram matrix of the quadratic form with values in Q/2Z:
 [1],
Finite quadratic module over Integer Ring with invariants (2, 2)
 Gram matrix of the quadratic form with values in Q/2Z:
 [  1 1/2]
 [1/2   1]]
```

```
>>> from sage.all import *
>>> D = IntegralLattice("D4").discriminant_group()                    #␣
↪needs sage.combinat
>>> D.all_submodules()                                                #␣
↪needs sage.combinat
[Finite quadratic module over Integer Ring with invariants ()
  Gram matrix of the quadratic form with values in Q/2Z:
  [],
 Finite quadratic module over Integer Ring with invariants (2,)
  Gram matrix of the quadratic form with values in Q/2Z:
  [1],
 Finite quadratic module over Integer Ring with invariants (2,)
  Gram matrix of the quadratic form with values in Q/2Z:
  [1],
 Finite quadratic module over Integer Ring with invariants (2,)
  Gram matrix of the quadratic form with values in Q/2Z:
  [1],
 Finite quadratic module over Integer Ring with invariants (2, 2)
  Gram matrix of the quadratic form with values in Q/2Z:
  [  1 1/2]
  [1/2   1]]
```

brown_invariant()

Return the Brown invariant of this torsion quadratic form.

Let (D, q) be a torsion quadratic module with values in $\mathbf{Q}/2\mathbf{Z}$. The Brown invariant $Br(D, q) \in \mathbf{Z}/8\mathbf{Z}$ is defined by the equation

$$\exp\left(\frac{2\pi i}{8} Br(q)\right) = \frac{1}{\sqrt{D}} \sum_{x \in D} \exp(i\pi q(x)).$$

The Brown invariant is additive with respect to direct sums of torsion quadratic modules.

OUTPUT: an element of $\mathbf{Z}/8\mathbf{Z}$

EXAMPLES:

```
sage: L = IntegralLattice("D4")                                       #␣
↪needs sage.combinat
sage: D = L.discriminant_group()                                      #␣
↪needs sage.combinat
sage: D.brown_invariant()                                             #␣
↪needs sage.combinat
4
```

```
>>> from sage.all import *
>>> L = IntegralLattice("D4")                                    #␣
↪needs sage.combinat
>>> D = L.discriminant_group()                                   #␣
↪needs sage.combinat
>>> D.brown_invariant()                                          #␣
↪needs sage.combinat
4
```

We require the quadratic form to be defined modulo $2\mathbf{Z}$:

```
sage: from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
sage: V = FreeQuadraticModule(ZZ, 3, matrix.identity(3))
sage: T = TorsionQuadraticModule((1/10)*V, V)
sage: T.brown_invariant()
Traceback (most recent call last):
...
ValueError: the torsion quadratic form must have values in QQ / 2 ZZ
```

```
>>> from sage.all import *
>>> from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
>>> V = FreeQuadraticModule(ZZ, Integer(3), matrix.identity(Integer(3)))
>>> T = TorsionQuadraticModule((Integer(1)/Integer(10))*V, V)
>>> T.brown_invariant()
Traceback (most recent call last):
...
ValueError: the torsion quadratic form must have values in QQ / 2 ZZ
```

gens()

Return generators of `self`.

There is no assumption on the generators except that they generate the module.

EXAMPLES:

```
sage: from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
sage: V = FreeModule(ZZ, 3)
sage: T = TorsionQuadraticModule(V, 5*V)
sage: T.gens()
((1, 0, 0), (0, 1, 0), (0, 0, 1))
```

```
>>> from sage.all import *
>>> from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
>>> V = FreeModule(ZZ, Integer(3))
>>> T = TorsionQuadraticModule(V, Integer(5)*V)
>>> T.gens()
((1, 0, 0), (0, 1, 0), (0, 0, 1))
```

genus(*signature_pair*)

Return the genus defined by `self` and the `signature_pair`.

If no such genus exists, raise a `ValueError`.

REFERENCES:

[Nik1977] Corollary 1.9.4 and 1.16.3.

EXAMPLES:

```
sage: # needs sage.combinat
sage: L = IntegralLattice("D4").direct_sum(IntegralLattice("A2"))
sage: D = L.discriminant_group()
sage: genus = D.genus(L.signature_pair())                                   #␣
↪needs sage.libs.pari
sage: genus                                                                 #␣
↪needs sage.libs.pari
Genus of
None
Signature:  (6, 0)
Genus symbol at 2:    1^4:2^-2
Genus symbol at 3:     1^-5 3^-1
sage: genus == L.genus()                                                    #␣
↪needs sage.libs.pari
True
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> L = IntegralLattice("D4").direct_sum(IntegralLattice("A2"))
>>> D = L.discriminant_group()
>>> genus = D.genus(L.signature_pair())                                     #␣
↪needs sage.libs.pari
>>> genus                                                                   #␣
↪needs sage.libs.pari
Genus of
None
Signature:  (6, 0)
Genus symbol at 2:    1^4:2^-2
Genus symbol at 3:     1^-5 3^-1
>>> genus == L.genus()                                                      #␣
↪needs sage.libs.pari
True
```

Let H be an even unimodular lattice of signature $(9, 1)$. Then $L = D_4 + A_2$ is primitively embedded in H. We compute the discriminant form of the orthogonal complement of L in H:

```
sage: DK = D.twist(-1)                                                      #␣
↪needs sage.combinat sage.libs.pari
sage: DK                                                                    #␣
↪needs sage.combinat sage.libs.pari
Finite quadratic module over Integer Ring with invariants (2, 6)
Gram matrix of the quadratic form with values in Q/2Z:
[  1 1/2]
[1/2 1/3]
```

```
>>> from sage.all import *
>>> DK = D.twist(-Integer(1))                                               ␣
↪     # needs sage.combinat sage.libs.pari
>>> DK                                                                      #␣
↪needs sage.combinat sage.libs.pari
```

(continues on next page)

(continued from previous page)

```
Finite quadratic module over Integer Ring with invariants (2, 6)
Gram matrix of the quadratic form with values in Q/2Z:
[  1 1/2]
[1/2 1/3]
```

We know that K has signature $(5, 1)$ and thus we can compute the genus of K as:

```
sage: DK.genus((3,1))                                                          #␣
↪needs sage.combinat sage.libs.pari
Genus of
None
Signature:  (3, 1)
Genus symbol at 2:    1^2:2^-2
Genus symbol at 3:     1^-3 3^1
```

```
>>> from sage.all import *
>>> DK.genus((Integer(3),Integer(1)))                                          ␣
↪              # needs sage.combinat sage.libs.pari
Genus of
None
Signature:  (3, 1)
Genus symbol at 2:    1^2:2^-2
Genus symbol at 3:     1^-3 3^1
```

We can also compute the genus of an odd lattice from its discriminant form:

```
sage: L = IntegralLattice(matrix.diagonal(range(1, 5)))
sage: D = L.discriminant_group()
sage: D.genus((4,0))                                                           #␣
↪needs sage.libs.pari
Genus of
None
Signature:  (4, 0)
Genus symbol at 2:    [1^-2 2^1 4^1]_6
Genus symbol at 3:     1^-3 3^1
```

```
>>> from sage.all import *
>>> L = IntegralLattice(matrix.diagonal(range(Integer(1), Integer(5))))
>>> D = L.discriminant_group()
>>> D.genus((Integer(4),Integer(0)))                                           ␣
↪              # needs sage.libs.pari
Genus of
None
Signature:  (4, 0)
Genus symbol at 2:    [1^-2 2^1 4^1]_6
Genus symbol at 3:     1^-3 3^1
```

gram_matrix_bilinear()

Return the Gram matrix with respect to the generators.

OUTPUT:

A rational matrix `G` with `G[i,j]` given by the inner product of the i-th and j-th generator. Its entries are only well defined mod (V, W).

EXAMPLES:

```
sage: from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
sage: V = FreeQuadraticModule(ZZ, 3, matrix.identity(3)*5)
sage: T = TorsionQuadraticModule((1/5)*V, V)
sage: T.gram_matrix_bilinear()
[1/5   0   0]
[  0 1/5   0]
[  0   0 1/5]
```

```
>>> from sage.all import *
>>> from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
>>> V = FreeQuadraticModule(ZZ, Integer(3), matrix.
...identity(Integer(3))*Integer(5))
>>> T = TorsionQuadraticModule((Integer(1)/Integer(5))*V, V)
>>> T.gram_matrix_bilinear()
[1/5   0   0]
[  0 1/5   0]
[  0   0 1/5]
```

gram_matrix_quadratic()

The Gram matrix of the quadratic form with respect to the generators.

OUTPUT:

- a rational matrix `Gq` with `Gq[i,j] = gens[i]*gens[j]` and `G[i,i] = gens[i].q()`

EXAMPLES:

```
sage: from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
sage: D4_gram = Matrix(ZZ, [[2,0,0,-1],[0,2,0,-1],[0,0,2,-1],[-1,-1,-1,2]])
sage: D4 = FreeQuadraticModule(ZZ, 4, D4_gram)
sage: D4dual = D4.span(D4_gram.inverse())
sage: discrForm = TorsionQuadraticModule(D4dual, D4)
sage: discrForm.gram_matrix_quadratic()
[  1 1/2]
[1/2   1]
sage: discrForm.gram_matrix_bilinear()
[  0 1/2]
[1/2   0]
```

```
>>> from sage.all import *
>>> from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
>>> D4_gram = Matrix(ZZ, [[Integer(2),Integer(0),Integer(0),-Integer(1)],
...[Integer(0),Integer(2),Integer(0),-Integer(1)],[Integer(0),Integer(0),
...Integer(2),-Integer(1)],[-Integer(1),-Integer(1),-Integer(1),Integer(2)]])
>>> D4 = FreeQuadraticModule(ZZ, Integer(4), D4_gram)
>>> D4dual = D4.span(D4_gram.inverse())
>>> discrForm = TorsionQuadraticModule(D4dual, D4)
>>> discrForm.gram_matrix_quadratic()
[  1 1/2]
[1/2   1]
>>> discrForm.gram_matrix_bilinear()
[  0 1/2]
[1/2   0]
```

is_genus(*signature_pair*, *even=True*)

Return `True` if there is a lattice with this signature and discriminant form.

Todo

implement the same for odd lattices

INPUT:

- `signature_pair` – tuple of nonnegative integers `(s_plus, s_minus)`
- `even` – boolean (default: `True`)

EXAMPLES:

```
sage: L3 = IntegralLattice(3 * Matrix(ZZ, 2, [2,1,1,2]))
sage: L = IntegralLattice("D4").direct_sum(L3)                              #␣
↪needs sage.combinat
sage: D = L.discriminant_group()                                            #␣
↪needs sage.combinat
sage: D.is_genus((6,0))                                                     #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> L3 = IntegralLattice(Integer(3) * Matrix(ZZ, Integer(2), [Integer(2),
↪Integer(1),Integer(1),Integer(2)]))
>>> L = IntegralLattice("D4").direct_sum(L3)                                #␣
↪needs sage.combinat
>>> D = L.discriminant_group()                                              #␣
↪needs sage.combinat
>>> D.is_genus((Integer(6),Integer(0)))                                       ␣
↪              # needs sage.combinat
True
```

Let us see if there is a lattice in the genus defined by the same discriminant form but with a different signature:

```
sage: D.is_genus((4,2))                                                     #␣
↪needs sage.combinat
False
sage: D.is_genus((16,2))                                                    #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> D.is_genus((Integer(4),Integer(2)))                                       ␣
↪              # needs sage.combinat
False
>>> D.is_genus((Integer(16),Integer(2)))                                      ␣
↪              # needs sage.combinat
True
```

normal_form(*partial=False*)

Return the normal form of this torsion quadratic module.

Two torsion quadratic modules are isomorphic if and only if they have the same value modules and the same normal form.

A torsion quadratic module (T, q) with values in $\mathbf{Q}/n\mathbf{Z}$ is in normal form if the rescaled quadratic module $(T, q/n)$ with values in $\mathbf{Q}/\mathbf{Z}$ is in normal form.

For the definition of normal form see [MirMor2009] IV Definition 4.6. Below are some of its properties. Let p be odd and u be the smallest non-square modulo p. The normal form is a diagonal matrix with diagonal entries either p^n or up^n.

If $p = 2$ is even, then the normal form consists of 1×1 blocks of the form

$$(0), \quad 2^n(1), \quad 2^n(3), \quad 2^n(5), \quad 2^n(7)$$

or of 2×2 blocks of the form

$$2^n \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}, \quad 2^n \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

The blocks are ordered by their valuation.

INPUT:

- `partial` – boolean (default: `False`); return only a partial normal form. It is not unique but still useful to extract invariants.

OUTPUT: a torsion quadratic module

EXAMPLES:

```
sage: L1 = IntegralLattice(matrix([[-2,0,0], [0,1,0], [0,0,4]]))
sage: L1.discriminant_group().normal_form()                                 #␣
↪needs sage.rings.padics
Finite quadratic module over Integer Ring with invariants (2, 4)
Gram matrix of the quadratic form with values in Q/Z:
[1/2   0]
[  0 1/4]
sage: L2 = IntegralLattice(matrix([[-2,0,0], [0,1,0], [0,0,-4]]))
sage: L2.discriminant_group().normal_form()                                 #␣
↪needs sage.rings.padics
Finite quadratic module over Integer Ring with invariants (2, 4)
Gram matrix of the quadratic form with values in Q/Z:
[1/2   0]
[  0 1/4]
```

```
>>> from sage.all import *
>>> L1 = IntegralLattice(matrix([[-Integer(2),Integer(0),Integer(0)],␣
↪[Integer(0),Integer(1),Integer(0)], [Integer(0),Integer(0),Integer(4)]]))
>>> L1.discriminant_group().normal_form()                                   #␣
↪needs sage.rings.padics
Finite quadratic module over Integer Ring with invariants (2, 4)
Gram matrix of the quadratic form with values in Q/Z:
[1/2   0]
[  0 1/4]
>>> L2 = IntegralLattice(matrix([[-Integer(2),Integer(0),Integer(0)],␣
↪[Integer(0),Integer(1),Integer(0)], [Integer(0),Integer(0),-Integer(4)]]))
>>> L2.discriminant_group().normal_form()                                   #␣
↪needs sage.rings.padics
```

(continues on next page)

(continued from previous page)

```
Finite quadratic module over Integer Ring with invariants (2, 4)
Gram matrix of the quadratic form with values in Q/Z:
[1/2   0]
[  0 1/4]
```

We check that Issue #24864 is fixed:

```
sage: L1 = IntegralLattice(matrix([[-4,0,0], [0,4,0], [0,0,-2]]))
sage: AL1 = L1.discriminant_group()
sage: L2 = IntegralLattice(matrix([[-4,0,0], [0,-4,0], [0,0,2]]))
sage: AL2 = L2.discriminant_group()
sage: AL1.normal_form()                                                     #␣
↪needs sage.rings.padics
Finite quadratic module over Integer Ring with invariants (2, 4, 4)
Gram matrix of the quadratic form with values in Q/2Z:
[1/2   0   0]
[  0 1/4   0]
[  0   0 5/4]
sage: AL2.normal_form()                                                     #␣
↪needs sage.libs.pari sage.rings.padics
Finite quadratic module over Integer Ring with invariants (2, 4, 4)
Gram matrix of the quadratic form with values in Q/2Z:
[1/2   0   0]
[  0 1/4   0]
[  0   0 5/4]
```

```
>>> from sage.all import *
>>> L1 = IntegralLattice(matrix([[-Integer(4),Integer(0),Integer(0)],␣
↪[Integer(0),Integer(4),Integer(0)], [Integer(0),Integer(0),-Integer(2)]]))
>>> AL1 = L1.discriminant_group()
>>> L2 = IntegralLattice(matrix([[-Integer(4),Integer(0),Integer(0)],␣
↪[Integer(0),-Integer(4),Integer(0)], [Integer(0),Integer(0),Integer(2)]]))
>>> AL2 = L2.discriminant_group()
>>> AL1.normal_form()                                                       #␣
↪needs sage.rings.padics
Finite quadratic module over Integer Ring with invariants (2, 4, 4)
Gram matrix of the quadratic form with values in Q/2Z:
[1/2   0   0]
[  0 1/4   0]
[  0   0 5/4]
>>> AL2.normal_form()                                                       #␣
↪needs sage.libs.pari sage.rings.padics
Finite quadratic module over Integer Ring with invariants (2, 4, 4)
Gram matrix of the quadratic form with values in Q/2Z:
[1/2   0   0]
[  0 1/4   0]
[  0   0 5/4]
```

Some exotic cases:

```
sage: from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
sage: D4_gram = Matrix(ZZ, 4, 4,[2,0,0,-1, 0,2,0,-1, 0,0,2,-1, -1,-1,-1,2])
```

(continues on next page)

(continued from previous page)

```
sage: D4 = FreeQuadraticModule(ZZ, 4, D4_gram)
sage: D4dual = D4.span(D4_gram.inverse())
sage: T = TorsionQuadraticModule((1/6)*D4dual, D4); T
Finite quadratic module over Integer Ring with invariants (6, 6, 12, 12)
Gram matrix of the quadratic form with values in Q/(1/3)Z:
[ 1/18  1/12  5/36  1/36]
[ 1/12   1/6  1/36   1/9]
[ 5/36  1/36  1/36 11/72]
[ 1/36   1/9 11/72  1/36]
sage: T.normal_form()                                                          #␣
↪needs sage.libs.pari sage.rings.padics
Finite quadratic module over Integer Ring with invariants (6, 6, 12, 12)
Gram matrix of the quadratic form with values in Q/(1/3)Z:
[ 1/6 1/12    0    0    0    0    0    0]
[1/12  1/6    0    0    0    0    0    0]
[   0    0 1/12 1/24    0    0    0    0]
[   0    0 1/24 1/12    0    0    0    0]
[   0    0    0    0  1/9    0    0    0]
[   0    0    0    0    0  1/9    0    0]
[   0    0    0    0    0    0  1/9    0]
[   0    0    0    0    0    0    0  1/9]
```

```
>>> from sage.all import *
>>> from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
>>> D4_gram = Matrix(ZZ, Integer(4), Integer(4),[Integer(2),Integer(0),
↪Integer(0),-Integer(1), Integer(0),Integer(2),Integer(0),-Integer(1),␣
↪Integer(0),Integer(0),Integer(2),-Integer(1), -Integer(1),-Integer(1),-
↪Integer(1),Integer(2)])
>>> D4 = FreeQuadraticModule(ZZ, Integer(4), D4_gram)
>>> D4dual = D4.span(D4_gram.inverse())
>>> T = TorsionQuadraticModule((Integer(1)/Integer(6))*D4dual, D4); T
Finite quadratic module over Integer Ring with invariants (6, 6, 12, 12)
Gram matrix of the quadratic form with values in Q/(1/3)Z:
[ 1/18  1/12  5/36  1/36]
[ 1/12   1/6  1/36   1/9]
[ 5/36  1/36  1/36 11/72]
[ 1/36   1/9 11/72  1/36]
>>> T.normal_form()                                                            #␣
↪needs sage.libs.pari sage.rings.padics
Finite quadratic module over Integer Ring with invariants (6, 6, 12, 12)
Gram matrix of the quadratic form with values in Q/(1/3)Z:
[ 1/6 1/12    0    0    0    0    0    0]
[1/12  1/6    0    0    0    0    0    0]
[   0    0 1/12 1/24    0    0    0    0]
[   0    0 1/24 1/12    0    0    0    0]
[   0    0    0    0  1/9    0    0    0]
[   0    0    0    0    0  1/9    0    0]
[   0    0    0    0    0    0  1/9    0]
[   0    0    0    0    0    0    0  1/9]
```

orthogonal_group(*gens=None*, *check=False*)

Orthogonal group of the associated torsion quadratic form.

Warning

This is can be smaller than the orthogonal group of the bilinear form.

INPUT:

- `gens` – a list of generators, for instance square matrices, something that acts on `self`, or an automorphism of the underlying abelian group
- `check` – perform additional checks on the generators

EXAMPLES:

You can provide generators to obtain a subgroup of the full orthogonal group:

```
sage: D = TorsionQuadraticForm(matrix.identity(2)/2)
sage: f = matrix(2, [0,1,1,0])
sage: D.orthogonal_group(gens=[f]).order()                                  #␣
↪needs sage.groups
2
```

```
>>> from sage.all import *
>>> D = TorsionQuadraticForm(matrix.identity(Integer(2))/Integer(2))
>>> f = matrix(Integer(2), [Integer(0),Integer(1),Integer(1),Integer(0)])
>>> D.orthogonal_group(gens=[f]).order()                                    #␣
↪needs sage.groups
2
```

If no generators are given a slow brute force approach is used to calculate the full orthogonal group:

```
sage: D = TorsionQuadraticForm(matrix.identity(3)/2)
sage: OD = D.orthogonal_group()                                             #␣
↪needs sage.groups
sage: OD.order()                                                            #␣
↪needs sage.groups
6
sage: fd = D.hom([D.1, D.0, D.2])                                           #␣
↪needs sage.symbolic
sage: OD(fd)                                                                #␣
↪needs sage.groups sage.symbolic
[0 1 0]
[1 0 0]
[0 0 1]
```

```
>>> from sage.all import *
>>> D = TorsionQuadraticForm(matrix.identity(Integer(3))/Integer(2))
>>> OD = D.orthogonal_group()                                               #␣
↪needs sage.groups
>>> OD.order()                                                              #␣
↪needs sage.groups
6
>>> fd = D.hom([D.gen(1), D.gen(0), D.gen(2)])                                ␣
↪           # needs sage.symbolic
>>> OD(fd)                                                                  #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.groups sage.symbolic
[0 1 0]
[1 0 0]
[0 0 1]
```

We compute the kernel of the action of the orthogonal group of L on the discriminant group:

```
sage: # needs sage.combinat sage.groups
sage: L = IntegralLattice('A4')
sage: O = L.orthogonal_group()
sage: D = L.discriminant_group()
sage: Obar = D.orthogonal_group(O.gens())
sage: O.order()
240
sage: Obar.order()
2
sage: phi = O.hom(Obar.gens())
sage: phi.kernel().order()
120
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.groups
>>> L = IntegralLattice('A4')
>>> O = L.orthogonal_group()
>>> D = L.discriminant_group()
>>> Obar = D.orthogonal_group(O.gens())
>>> O.order()
240
>>> Obar.order()
2
>>> phi = O.hom(Obar.gens())
>>> phi.kernel().order()
120
```

orthogonal_submodule_to(S)

Return the submodule orthogonal to `S`.

INPUT:

- `S` – a submodule, list, or tuple of generators

EXAMPLES:

```
sage: from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
sage: V = FreeModule(ZZ, 10)
sage: T = TorsionQuadraticModule(V, 3*V)
sage: S = T.submodule(T.gens()[:5])
sage: O = T.orthogonal_submodule_to(S)
sage: O
Finite quadratic module over Integer Ring with invariants (3, 3, 3, 3, 3)
Gram matrix of the quadratic form with values in Q/3Z:
[1 0 0 0 0]
[0 1 0 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
sage: O.V() + S.V() == T.V()
True
```

```
>>> from sage.all import *
>>> from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
>>> V = FreeModule(ZZ, Integer(10))
>>> T = TorsionQuadraticModule(V, Integer(3)*V)
>>> S = T.submodule(T.gens()[:Integer(5)])
>>> O = T.orthogonal_submodule_to(S)
>>> O
Finite quadratic module over Integer Ring with invariants (3, 3, 3, 3, 3)
Gram matrix of the quadratic form with values in Q/3Z:
[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]
>>> O.V() + S.V() == T.V()
True
```

primary_part(*m*)

Return the `m`-primary part of this torsion quadratic module as a submodule.

INPUT:

- `m` – integer

OUTPUT: a submodule

EXAMPLES:

```
sage: from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
sage: T = TorsionQuadraticModule((1/6)*ZZ^3, ZZ^3); T
Finite quadratic module over Integer Ring with invariants (6, 6, 6)
Gram matrix of the quadratic form with values in Q/(1/3)Z:
[1/36    0    0]
[   0 1/36    0]
[   0    0 1/36]
sage: T.primary_part(2)
Finite quadratic module over Integer Ring with invariants (2, 2, 2)
Gram matrix of the quadratic form with values in Q/(1/3)Z:
[1/4   0   0]
[  0 1/4   0]
[  0   0 1/4]
```

```
>>> from sage.all import *
>>> from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
>>> T = TorsionQuadraticModule((Integer(1)/Integer(6))*ZZ**Integer(3),␣
↪ZZ**Integer(3)); T
Finite quadratic module over Integer Ring with invariants (6, 6, 6)
```

(continues on next page)

(continued from previous page)

```
Gram matrix of the quadratic form with values in Q/(1/3)Z:
[1/36    0    0]
[   0 1/36    0]
[   0    0 1/36]
>>> T.primary_part(Integer(2))
Finite quadratic module over Integer Ring with invariants (2, 2, 2)
Gram matrix of the quadratic form with values in Q/(1/3)Z:
[1/4   0   0]
[  0 1/4   0]
[  0   0 1/4]
```

submodule_with_gens(*gens*)

Return a submodule with generators given by `gens`.

INPUT:

- `gens` – list of generators that convert into `self`

OUTPUT: a submodule with the specified generators

EXAMPLES:

```
sage: from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
sage: V = FreeQuadraticModule(ZZ, 3, matrix.identity(3)*10)
sage: T = TorsionQuadraticModule((1/10)*V, V)
sage: g = T.gens()
sage: new_gens = [2*g[0], 5*g[0]]
sage: T.submodule_with_gens(new_gens)
Finite quadratic module over Integer Ring with invariants (10,)
Gram matrix of the quadratic form with values in Q/2Z:
[2/5   0]
[  0 1/2]
```

```
>>> from sage.all import *
>>> from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
>>> V = FreeQuadraticModule(ZZ, Integer(3), matrix.
↪identity(Integer(3))*Integer(10))
>>> T = TorsionQuadraticModule((Integer(1)/Integer(10))*V, V)
>>> g = T.gens()
>>> new_gens = [Integer(2)*g[Integer(0)], Integer(5)*g[Integer(0)]]
>>> T.submodule_with_gens(new_gens)
Finite quadratic module over Integer Ring with invariants (10,)
Gram matrix of the quadratic form with values in Q/2Z:
[2/5   0]
[  0 1/2]
```

The generators do not need to be independent:

```
sage: new_gens = [g[0], 2*g[1], g[0], g[1]]
sage: T.submodule_with_gens(new_gens)
Finite quadratic module over Integer Ring with invariants (10, 10)
Gram matrix of the quadratic form with values in Q/2Z:
[1/10    0 1/10    0]
[   0  2/5    0  1/5]
```

(continues on next page)

(continued from previous page)

```
[1/10    0 1/10    0]
[   0  1/5    0 1/10]
```

```
>>> from sage.all import *
>>> new_gens = [g[Integer(0)], Integer(2)*g[Integer(1)], g[Integer(0)],␣
↪g[Integer(1)]]
>>> T.submodule_with_gens(new_gens)
Finite quadratic module over Integer Ring with invariants (10, 10)
Gram matrix of the quadratic form with values in Q/2Z:
[1/10    0 1/10    0]
[   0  2/5    0  1/5]
[1/10    0 1/10    0]
[   0  1/5    0 1/10]
```

twist(*s*)

Return the torsion quadratic module with quadratic form scaled by `s`.

If the old form was defined modulo n, then the new form is defined modulo ns.

INPUT:

- `s` – a rational number

EXAMPLES:

```
sage: q = TorsionQuadraticForm(matrix.diagonal([3/9, 1/9]))
sage: q.twist(-1)
Finite quadratic module over Integer Ring with invariants (3, 9)
Gram matrix of the quadratic form with values in Q/Z:
[2/3   0]
[  0 8/9]
```

```
>>> from sage.all import *
>>> q = TorsionQuadraticForm(matrix.diagonal([Integer(3)/Integer(9),␣
↪Integer(1)/Integer(9)]))
>>> q.twist(-Integer(1))
Finite quadratic module over Integer Ring with invariants (3, 9)
Gram matrix of the quadratic form with values in Q/Z:
[2/3   0]
[  0 8/9]
```

This form is defined modulo 3:

```
sage: q.twist(3)
Finite quadratic module over Integer Ring with invariants (3, 9)
Gram matrix of the quadratic form with values in Q/3Z:
[  1   0]
[  0 1/3]
```

```
>>> from sage.all import *
>>> q.twist(Integer(3))
Finite quadratic module over Integer Ring with invariants (3, 9)
Gram matrix of the quadratic form with values in Q/3Z:
```

(continues on next page)

(continued from previous page)

```
[  1   0]
[  0 1/3]
```

The next form is defined modulo 4:

```
sage: q.twist(4)
Finite quadratic module over Integer Ring with invariants (3, 9)
Gram matrix of the quadratic form with values in Q/4Z:
[4/3   0]
[  0 4/9]
```

```
>>> from sage.all import *
>>> q.twist(Integer(4))
Finite quadratic module over Integer Ring with invariants (3, 9)
Gram matrix of the quadratic form with values in Q/4Z:
[4/3   0]
[  0 4/9]
```

value_module()

Return $\mathbf{Q}/m\mathbf{Z}$ with $m = (V, W)$.

This is where the inner product takes values.

EXAMPLES:

```
sage: A2 = Matrix(ZZ, 2, 2, [2,-1,-1,2])
sage: L = IntegralLattice(2*A2)
sage: D = L.discriminant_group(); D
Finite quadratic module over Integer Ring with invariants (2, 6)
Gram matrix of the quadratic form with values in Q/2Z:
[  1 1/2]
[1/2 1/3]
sage: D.value_module()
Q/Z
```

```
>>> from sage.all import *
>>> A2 = Matrix(ZZ, Integer(2), Integer(2), [Integer(2),-Integer(1),-
↪Integer(1),Integer(2)])
>>> L = IntegralLattice(Integer(2)*A2)
>>> D = L.discriminant_group(); D
Finite quadratic module over Integer Ring with invariants (2, 6)
Gram matrix of the quadratic form with values in Q/2Z:
[  1 1/2]
[1/2 1/3]
>>> D.value_module()
Q/Z
```

value_module_qf()

Return $\mathbf{Q}/n\mathbf{Z}$ with $n\mathbf{Z} = (V, W) + \mathbf{Z}\{(w, w)|w \in W\}$.

This is where the torsion quadratic form takes values.

EXAMPLES:

```
sage: A2 = Matrix(ZZ, 2, 2, [2,-1,-1,2])
sage: L = IntegralLattice(2 * A2)
sage: D = L.discriminant_group(); D
Finite quadratic module over Integer Ring with invariants (2, 6)
Gram matrix of the quadratic form with values in Q/2Z:
[  1 1/2]
[1/2 1/3]
sage: D.value_module_qf()
Q/2Z
```

```
>>> from sage.all import *
>>> A2 = Matrix(ZZ, Integer(2), Integer(2), [Integer(2),-Integer(1),-
↪Integer(1),Integer(2)])
>>> L = IntegralLattice(Integer(2) * A2)
>>> D = L.discriminant_group(); D
Finite quadratic module over Integer Ring with invariants (2, 6)
Gram matrix of the quadratic form with values in Q/2Z:
[  1 1/2]
[1/2 1/3]
>>> D.value_module_qf()
Q/2Z
```

class sage.modules.torsion_quadratic_module.**TorsionQuadraticModuleElement**(*parent*, *x*, *check=True*)

Bases: *FGP_Element*

An element of a torsion quadratic module.

INPUT:

- `parent` – parent
- `x` – element of `parent.V()`
- `check` – boolean (default: `True`)

b(*other*)

Compute the inner product of two elements.

OUTPUT:

- an element of $\mathbf{Q}/m\mathbf{Z}$ with $m\mathbf{Z} = (V, W)$

EXAMPLES:

```
sage: from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
sage: V = (1/2)*ZZ^2; W = ZZ^2
sage: T = TorsionQuadraticModule(V, W)
sage: g = T.gens()
sage: x = g[0]; x
(1, 0)
sage: y = g[0] + g[1]
sage: x*y
1/4
```

```
>>> from sage.all import *
>>> from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
>>> V = (Integer(1)/Integer(2))*ZZ**Integer(2); W = ZZ**Integer(2)
>>> T = TorsionQuadraticModule(V, W)
>>> g = T.gens()
>>> x = g[Integer(0)]; x
(1, 0)
>>> y = g[Integer(0)] + g[Integer(1)]
>>> x*y
1/4
```

The inner product has further aliases:

```
sage: x.inner_product(y)
1/4
sage: x.b(y)
1/4
```

```
>>> from sage.all import *
>>> x.inner_product(y)
1/4
>>> x.b(y)
1/4
```

inner_product(*other*)

Compute the inner product of two elements.

OUTPUT:

- an element of $\mathbf{Q}/m\mathbf{Z}$ with $m\mathbf{Z} = (V, W)$

EXAMPLES:

```
sage: from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
sage: V = (1/2)*ZZ^2; W = ZZ^2
sage: T = TorsionQuadraticModule(V, W)
sage: g = T.gens()
sage: x = g[0]; x
(1, 0)
sage: y = g[0] + g[1]
sage: x*y
1/4
```

```
>>> from sage.all import *
>>> from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
>>> V = (Integer(1)/Integer(2))*ZZ**Integer(2); W = ZZ**Integer(2)
>>> T = TorsionQuadraticModule(V, W)
>>> g = T.gens()
>>> x = g[Integer(0)]; x
(1, 0)
>>> y = g[Integer(0)] + g[Integer(1)]
>>> x*y
1/4
```

The inner product has further aliases:

```
sage: x.inner_product(y)
1/4
sage: x.b(y)
1/4
```

```
>>> from sage.all import *
>>> x.inner_product(y)
1/4
>>> x.b(y)
1/4
```

q()

Compute the quadratic product of `self`.

OUTPUT:

- an element of $\mathbf{Q}/n\mathbf{Z}$ where $n\mathbf{Z} = 2(V, W) + \mathbf{Z}\{(w, w)|w \in W\}$

EXAMPLES:

```
sage: from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
sage: W = FreeQuadraticModule(ZZ, 2, 2*matrix.identity(2))
sage: V = (1/2) * W
sage: T = TorsionQuadraticModule(V, W)
sage: x = T.gen(0)
sage: x
(1, 0)
sage: x.quadratic_product()
1/2
sage: x.quadratic_product().parent()
Q/2Z
sage: x*x
1/2
sage: (x*x).parent()
Q/Z
```

```
>>> from sage.all import *
>>> from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
>>> W = FreeQuadraticModule(ZZ, Integer(2), Integer(2)*matrix.
↪identity(Integer(2)))
>>> V = (Integer(1)/Integer(2)) * W
>>> T = TorsionQuadraticModule(V, W)
>>> x = T.gen(Integer(0))
>>> x
(1, 0)
>>> x.quadratic_product()
1/2
>>> x.quadratic_product().parent()
Q/2Z
>>> x*x
1/2
>>> (x*x).parent()
Q/Z
```

quadratic_product()

Compute the quadratic product of `self`.

OUTPUT:

- an element of $\mathbf{Q}/n\mathbf{Z}$ where $n\mathbf{Z} = 2(V, W) + \mathbf{Z}\{(w, w)|w \in W\}$

EXAMPLES:

```
sage: from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
sage: W = FreeQuadraticModule(ZZ, 2, 2*matrix.identity(2))
sage: V = (1/2) * W
sage: T = TorsionQuadraticModule(V, W)
sage: x = T.gen(0)
sage: x
(1, 0)
sage: x.quadratic_product()
1/2
sage: x.quadratic_product().parent()
Q/2Z
sage: x*x
1/2
sage: (x*x).parent()
Q/Z
```

```
>>> from sage.all import *
>>> from sage.modules.torsion_quadratic_module import TorsionQuadraticModule
>>> W = FreeQuadraticModule(ZZ, Integer(2), Integer(2)*matrix.
↪identity(Integer(2)))
>>> V = (Integer(1)/Integer(2)) * W
>>> T = TorsionQuadraticModule(V, W)
>>> x = T.gen(Integer(0))
>>> x
(1, 0)
>>> x.quadratic_product()
1/2
>>> x.quadratic_product().parent()
Q/2Z
>>> x*x
1/2
>>> (x*x).parent()
Q/Z
```

6.5 $\mathbb{Z}$-filtered vector spaces

This module implements filtered vector spaces, that is, a descending sequence of vector spaces

$$\cdots \supset F_d \supset F_{d+1} \supset F_{d+2} \supset \cdots$$

with degrees $d \in \mathbf{Z}$. It is not required that F_d is the entire ambient space for $d \ll 0$ (see *is_exhaustive()*) nor that $F_d = 0$ for $d \gg 0$ (see *is_separating()*). To construct a filtered vector space, use the *FilteredVectorSpace()* command. It supports easy creation of simple filtrations, for example the trivial one:

```
sage: FilteredVectorSpace(2, base_ring=RDF)
RDF^2
```

```
>>> from sage.all import *
>>> FilteredVectorSpace(Integer(2), base_ring=RDF)
RDF^2
```

The next-simplest filtration has a single non-trivial inclusion between V_d and V_{d+1}:

```
sage: d = 1
sage: V = FilteredVectorSpace(2, d);  V
QQ^2 >= 0
sage: [V.get_degree(i).dimension() for i in range(0,4)]
[2, 2, 0, 0]
```

```
>>> from sage.all import *
>>> d = Integer(1)
>>> V = FilteredVectorSpace(Integer(2), d);  V
QQ^2 >= 0
>>> [V.get_degree(i).dimension() for i in range(Integer(0),Integer(4))]
[2, 2, 0, 0]
```

To construct general filtrations, you need to tell Sage about generating vectors for the nested subspaces. For example, a dictionary whose keys are the degrees and values are a list of generators:

```
sage: r1 = (1, 0, 5)
sage: r2 = (0, 1, 2)
sage: r3 = (1, 2, 1)
sage: V = FilteredVectorSpace({0:[r1, r2, r3], 1:[r1, r2], 3:[r1]});  V
QQ^3 >= QQ^2 >= QQ^1 >= QQ^1 >= 0
```

```
>>> from sage.all import *
>>> r1 = (Integer(1), Integer(0), Integer(5))
>>> r2 = (Integer(0), Integer(1), Integer(2))
>>> r3 = (Integer(1), Integer(2), Integer(1))
>>> V = FilteredVectorSpace({Integer(0):[r1, r2, r3], Integer(1):[r1, r2],␣
↪Integer(3):[r1]});  V
QQ^3 >= QQ^2 >= QQ^1 >= QQ^1 >= 0
```

For degrees d that are not specified, the associated vector subspace is the same as the next-lower degree, that is, $V_d \simeq V_{d-1}$. In the above example, this means that

- $V_d \simeq \mathbf{Q}^3$ for $d < 0$
- $V_0 = span(r_1, r_2) \simeq \mathbf{Q}^2$
- $V_1 = V_2 = span(r_3) \simeq \mathbf{Q}$
- $V_d = 0$ for $d \geq 3$

That is:

```
sage: V.get_degree(0) == V
True
sage: V.get_degree(1) == V.span([r1, r2])
```

(continues on next page)

(continued from previous page)

```
True
sage: V.get_degree(2) == V.get_degree(3) == V.span([r1])
True
sage: V.get_degree(4) == V.get_degree(5) == V.span([])
True
```

```
>>> from sage.all import *
>>> V.get_degree(Integer(0)) == V
True
>>> V.get_degree(Integer(1)) == V.span([r1, r2])
True
>>> V.get_degree(Integer(2)) == V.get_degree(Integer(3)) == V.span([r1])
True
>>> V.get_degree(Integer(4)) == V.get_degree(Integer(5)) == V.span([])
True
```

If you have many generators you can just pass the generators once and then refer to them by index:

```
sage: FilteredVectorSpace([r1, r2, r3], {0:[0,1,2], 1:[1,2], 3:[1]})
QQ^3 >= QQ^2 >= QQ^1 >= QQ^1 >= 0
```

```
>>> from sage.all import *
>>> FilteredVectorSpace([r1, r2, r3], {Integer(0):[Integer(0),Integer(1),Integer(2)],␣
↪Integer(1):[Integer(1),Integer(2)], Integer(3):[Integer(1)]})
QQ^3 >= QQ^2 >= QQ^1 >= QQ^1 >= 0
```

Note that generators for the degree-d subspace of the filtration are automatically generators for all lower degrees. For example, here we do not have to specify the ray r_2 separately in degree 1:

```
sage: FilteredVectorSpace([r1, r2, r3], {0:[0   ], 1:[1]})
QQ^2 >= QQ^1 >= 0 in QQ^3
sage: FilteredVectorSpace([r1, r2, r3], {0:[0, 1], 1:[1]})
QQ^2 >= QQ^1 >= 0 in QQ^3
```

```
>>> from sage.all import *
>>> FilteredVectorSpace([r1, r2, r3], {Integer(0):[Integer(0)   ],␣
↪Integer(1):[Integer(1)]})
QQ^2 >= QQ^1 >= 0 in QQ^3
>>> FilteredVectorSpace([r1, r2, r3], {Integer(0):[Integer(0), Integer(1)],␣
↪Integer(1):[Integer(1)]})
QQ^2 >= QQ^1 >= 0 in QQ^3
```

The degree can be infinite (plus infinity), this allows construction of filtered vector spaces that are not eventually zero in high degree:

```
sage: FilteredVectorSpace([r1, r2, r3], {0:[0,1], oo:[1]})
QQ^2 >= QQ^1 in QQ^3
```

```
>>> from sage.all import *
>>> FilteredVectorSpace([r1, r2, r3], {Integer(0):[Integer(0),Integer(1)],␣
↪oo:[Integer(1)]})
QQ^2 >= QQ^1 in QQ^3
```

Any field can be used as the vector space base. For example a finite field:

```
sage: F.<a> = GF(5^3)                                                          #␣
↪needs sage.rings.finite_rings
sage: r1 = (a, 0, F(5));  r1                                                   #␣
↪needs sage.rings.finite_rings
(a, 0, 0)
sage: FilteredVectorSpace([r1, r2, r3], {0:[0,1], oo:[1]}, base_ring=F)        #␣
↪needs sage.rings.finite_rings
GF(125)^2 >= GF(125)^1 in GF(125)^3
```

```
>>> from sage.all import *
>>> F = GF(Integer(5)**Integer(3), names=('a',)); (a,) = F._first_ngens(1)# needs␣
↪sage.rings.finite_rings
>>> r1 = (a, Integer(0), F(Integer(5)));  r1                                    ␣
↪              # needs sage.rings.finite_rings
(a, 0, 0)
>>> FilteredVectorSpace([r1, r2, r3], {Integer(0):[Integer(0),Integer(1)],␣
↪oo:[Integer(1)]}, base_ring=F)              # needs sage.rings.finite_rings
GF(125)^2 >= GF(125)^1 in GF(125)^3
```

Or the algebraic field:

```
sage: r1 = (1, 0, 1+QQbar(I));  r1                                             #␣
↪needs sage.rings.number_field
(1, 0, I + 1)
sage: FilteredVectorSpace([r1, r2, r3], {0:[0,1], oo:[1]}, base_ring=QQbar)    #␣
↪needs sage.rings.number_field
Vector space of dimension 2 over Algebraic Field
>= Vector space of dimension 1 over Algebraic Field
in Vector space of dimension 3 over Algebraic Field
```

```
>>> from sage.all import *
>>> r1 = (Integer(1), Integer(0), Integer(1)+QQbar(I));  r1                     ␣
↪                       # needs sage.rings.number_field
(1, 0, I + 1)
>>> FilteredVectorSpace([r1, r2, r3], {Integer(0):[Integer(0),Integer(1)],␣
↪oo:[Integer(1)]}, base_ring=QQbar)          # needs sage.rings.number_field
Vector space of dimension 2 over Algebraic Field
>= Vector space of dimension 1 over Algebraic Field
in Vector space of dimension 3 over Algebraic Field
```

sage.modules.filtered_vector_space.**FilteredVectorSpace**(*arg1*, *arg2=None*, *base_ring=Rational Field*, *check=True*)

Construct a filtered vector space.

INPUT:

This function accepts various input that determines the vector space and filtration.

- Just the dimensionFilteredVectorSpace(dimension): Return the trivial filtration (where all vector spaces are isomorphic).
- Dimension and maximal degree, see `constructor_from_dim_degree()` for arguments. Construct a filtration with only one non-trivial step $V \supset 0$ at the given cutoff degree.

- A dictionary containing the degrees as keys and a list of vector space generators as values, see `FilteredVectorSpace_from_generators()`
- Generators and a dictionary containing the degrees as keys and the indices of vector space generators as values, see `FilteredVectorSpace_from_generators_indices()`

In addition, the following keyword arguments are supported:

- `base_ring` – a field (default: **Q**). The base field of the vector space. Must be a field.

EXAMPLES:

Just the dimension for the trivial filtration:

```
sage: FilteredVectorSpace(2)
QQ^2
```

```
>>> from sage.all import *
>>> FilteredVectorSpace(Integer(2))
QQ^2
```

Dimension and degree:

```
sage: FilteredVectorSpace(2, 1)
QQ^2 >= 0
```

```
>>> from sage.all import *
>>> FilteredVectorSpace(Integer(2), Integer(1))
QQ^2 >= 0
```

Dictionary of generators:

```
sage: FilteredVectorSpace({1:[(1,0), (0,1)], 3:[(1,0)]})
QQ^2 >= QQ^1 >= QQ^1 >= 0
```

```
>>> from sage.all import *
>>> FilteredVectorSpace({Integer(1):[(Integer(1),Integer(0)), (Integer(0),
↪Integer(1))], Integer(3):[(Integer(1),Integer(0))]})
QQ^2 >= QQ^1 >= QQ^1 >= 0
```

Generators and a dictionary referring to them by index:

```
sage: FilteredVectorSpace([(1,0), (0,1)], {1:[0,1], 3:[0]})
QQ^2 >= QQ^1 >= QQ^1 >= 0
```

```
>>> from sage.all import *
>>> FilteredVectorSpace([(Integer(1),Integer(0)), (Integer(0),Integer(1))],
↪{Integer(1):[Integer(0),Integer(1)], Integer(3):[Integer(0)]})
QQ^2 >= QQ^1 >= QQ^1 >= 0
```

class sage.modules.filtered_vector_space.**FilteredVectorSpace_class**(*base_ring*, *dim*, *generators*, *filtration*, *check=True*)

Bases: *FreeModule_ambient_field*

A descending filtration of a vector space.

INPUT:

- `base_ring` – a field; the base field of the ambient vector space
- `dim` – integer; the dimension of the ambient vector space
- `generators` – tuple of generators for the ambient vector space. These will be used to span the subspaces of the filtration.
- `filtration` – dictionary of filtration steps in ray index notation. See *construct_from_generators_indices()* for details.
- `check` – boolean (default: `True`); whether to perform consistency checks

ambient_vector_space()

Return the ambient (unfiltered) vector space.

OUTPUT: a vector space

EXAMPLES:

```
sage: V = FilteredVectorSpace(1, 0)
sage: V.ambient_vector_space()
Vector space of dimension 1 over Rational Field
```

```
>>> from sage.all import *
>>> V = FilteredVectorSpace(Integer(1), Integer(0))
>>> V.ambient_vector_space()
Vector space of dimension 1 over Rational Field
```

change_ring(*base_ring*)

Return the same filtration over a different base ring.

INPUT:

- `base_ring` – the new base ring

OUTPUT:

This method returns a new filtered vector space whose subspaces are defined by the same generators but over a different base ring.

EXAMPLES:

```
sage: V = FilteredVectorSpace(1, 0);  V
QQ^1 >= 0
sage: V.change_ring(RDF)
RDF^1 >= 0
```

```
>>> from sage.all import *
>>> V = FilteredVectorSpace(Integer(1), Integer(0));  V
QQ^1 >= 0
>>> V.change_ring(RDF)
RDF^1 >= 0
```

direct_sum(*other*)

Return the direct sum.

INPUT:

- `other` – a filtered vector space

OUTPUT: the direct sum as a filtered vector space

EXAMPLES:

```
sage: V = FilteredVectorSpace(2, 0)
sage: W = FilteredVectorSpace({0:[(1,-1),(2,1)], 1:[(1,1)]})
sage: V.direct_sum(W)
QQ^4 >= QQ^1 >= 0
sage: V + W    # syntactic sugar
QQ^4 >= QQ^1 >= 0
sage: V + V == FilteredVectorSpace(4, 0)
True

sage: W = FilteredVectorSpace([(1,-1),(2,1)], {1:[0,1], 2:[1]})
sage: V + W
QQ^4 >= QQ^2 >= QQ^1 >= 0
```

```
>>> from sage.all import *
>>> V = FilteredVectorSpace(Integer(2), Integer(0))
>>> W = FilteredVectorSpace({Integer(0):[(Integer(1),-Integer(1)),(Integer(2),
↪Integer(1))], Integer(1):[(Integer(1),Integer(1))]})
>>> V.direct_sum(W)
QQ^4 >= QQ^1 >= 0
>>> V + W    # syntactic sugar
QQ^4 >= QQ^1 >= 0
>>> V + V == FilteredVectorSpace(Integer(4), Integer(0))
True

>>> W = FilteredVectorSpace([(Integer(1),-Integer(1)),(Integer(2),
↪Integer(1))], {Integer(1):[Integer(0),Integer(1)], Integer(2):[Integer(1)]})
>>> V + W
QQ^4 >= QQ^2 >= QQ^1 >= 0
```

A suitable base ring is chosen if they do not match:

```
sage: v = [(1,0), (0,1)]
sage: F1 = FilteredVectorSpace(v, {0:[0], 1:[1]}, base_ring=QQ)
sage: F2 = FilteredVectorSpace(v, {0:[0], 1:[1]}, base_ring=RDF)
sage: F1 + F2                                                                 #␣
↪needs scipy
RDF^4 >= RDF^2 >= 0
```

```
>>> from sage.all import *
>>> v = [(Integer(1),Integer(0)), (Integer(0),Integer(1))]
>>> F1 = FilteredVectorSpace(v, {Integer(0):[Integer(0)],␣
↪Integer(1):[Integer(1)]}, base_ring=QQ)
>>> F2 = FilteredVectorSpace(v, {Integer(0):[Integer(0)],␣
↪Integer(1):[Integer(1)]}, base_ring=RDF)
>>> F1 + F2                                                                   #␣
↪needs scipy
RDF^4 >= RDF^2 >= 0
```

dual()

Return the dual filtered vector space.

OUTPUT:

The graded dual, that is, the dual of a degree-d subspace is a set of linear constraints in degree $-d+1$. That is, the dual generators live in degree $-d$.

EXAMPLES:

```
sage: gens = identity_matrix(3).rows()
sage: F = FilteredVectorSpace(gens, {0:[0,1,2], 2:[0]});  F
QQ^3 >= QQ^1 >= QQ^1 >= 0
sage: F.support()
(0, 2)

sage: F.dual()
QQ^3 >= QQ^2 >= QQ^2 >= 0
sage: F.dual().support()
(-2, 0)
```

```
>>> from sage.all import *
>>> gens = identity_matrix(Integer(3)).rows()
>>> F = FilteredVectorSpace(gens, {Integer(0):[Integer(0),Integer(1),
↪Integer(2)], Integer(2):[Integer(0)]});  F
QQ^3 >= QQ^1 >= QQ^1 >= 0
>>> F.support()
(0, 2)

>>> F.dual()
QQ^3 >= QQ^2 >= QQ^2 >= 0
>>> F.dual().support()
(-2, 0)
```

exterior_power(n)

Return the n-th graded exterior power.

INPUT:

- `n` – integer; exterior product of how many copies of `self`

OUTPUT:

The graded exterior product, that is, the wedge product of a generator of degree d_1 with a generator in degree d_2 has degree $d_1 + d_2$.

EXAMPLES:

```
sage: # needs sage.groups
sage: F = FilteredVectorSpace(1, 1) + FilteredVectorSpace(1, 2);  F
QQ^2 >= QQ^1 >= 0
sage: F.exterior_power(1)
QQ^2 >= QQ^1 >= 0
sage: F.exterior_power(2)
QQ^1 >= 0
sage: F.exterior_power(3)
0
sage: F.wedge(2)
QQ^1 >= 0
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> F = FilteredVectorSpace(Integer(1), Integer(1)) +␣
↪FilteredVectorSpace(Integer(1), Integer(2));  F
QQ^2 >= QQ^1 >= 0
>>> F.exterior_power(Integer(1))
QQ^2 >= QQ^1 >= 0
>>> F.exterior_power(Integer(2))
QQ^1 >= 0
>>> F.exterior_power(Integer(3))
0
>>> F.wedge(Integer(2))
QQ^1 >= 0
```

get_degree(*d*)

Return the degree-`d` entry of the filtration.

INPUT:

- `d` – integer; the desired degree of the filtration

OUTPUT:

The degree-`d` vector space in the filtration as subspace of the ambient space.

EXAMPLES:

```
sage: rays = [(1,0), (1,1), (1,2), (-1,-1)]
sage: F = FilteredVectorSpace(rays, {3:[1], 1:[1,2]})
sage: F.get_degree(2)
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 1]
sage: F.get_degree(oo)
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
sage: F.get_degree(-oo)
Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1 0]
[0 1]
```

```
>>> from sage.all import *
>>> rays = [(Integer(1),Integer(0)), (Integer(1),Integer(1)), (Integer(1),
↪Integer(2)), (-Integer(1),-Integer(1))]
>>> F = FilteredVectorSpace(rays, {Integer(3):[Integer(1)],␣
↪Integer(1):[Integer(1),Integer(2)]})
>>> F.get_degree(Integer(2))
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 1]
>>> F.get_degree(oo)
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
```

(continues on next page)

(continued from previous page)

```
[]
>>> F.get_degree(-oo)
Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1 0]
[0 1]
```

graded(*d*)

Return the associated graded vectorspace.

INPUT:

- `d` – integer; the degree

OUTPUT: the quotient $G_d = F_d/F_{d+1}$

EXAMPLES:

```
sage: rays = [(1,0), (1,1), (1,2)]
sage: F = FilteredVectorSpace(rays, {3:[1], 1:[1,2]})
sage: F.graded(1)
Vector space quotient V/W of dimension 1 over Rational Field where
V: Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1 0]
[0 1]
W: Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 1]
```

```
>>> from sage.all import *
>>> rays = [(Integer(1),Integer(0)), (Integer(1),Integer(1)), (Integer(1),
↪Integer(2))]
>>> F = FilteredVectorSpace(rays, {Integer(3):[Integer(1)],␣
↪Integer(1):[Integer(1),Integer(2)]})
>>> F.graded(Integer(1))
Vector space quotient V/W of dimension 1 over Rational Field where
V: Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1 0]
[0 1]
W: Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 1]
```

is_constant()

Return whether the filtration is constant.

OUTPUT: boolean; whether the filtered vector spaces are identical in all degrees

EXAMPLES:

```
sage: V = FilteredVectorSpace(2); V
QQ^2
sage: V.is_constant()
```

(continues on next page)

(continued from previous page)

```
True

sage: V = FilteredVectorSpace(1, 0);  V
QQ^1 >= 0
sage: V.is_constant()
False

sage: V = FilteredVectorSpace({0:[(1,)]});  V
QQ^1 >= 0
sage: V.is_constant()
False
```

```
>>> from sage.all import *
>>> V = FilteredVectorSpace(Integer(2)); V
QQ^2
>>> V.is_constant()
True

>>> V = FilteredVectorSpace(Integer(1), Integer(0));  V
QQ^1 >= 0
>>> V.is_constant()
False

>>> V = FilteredVectorSpace({Integer(0):[(Integer(1),)]});  V
QQ^1 >= 0
>>> V.is_constant()
False
```

is_exhaustive()

Return whether the filtration is exhaustive.

A filtration $\{F_d\}$ in an ambient vector space V is exhaustive if $\cup F_d = V$. See also *is_separating()*.

OUTPUT: boolean

EXAMPLES:

```
sage: F = FilteredVectorSpace({0:[(1,1)]});  F
QQ^1 >= 0 in QQ^2
sage: F.is_exhaustive()
False
sage: G = FilteredVectorSpace(2, 0);  G
QQ^2 >= 0
sage: G.is_exhaustive()
True
```

```
>>> from sage.all import *
>>> F = FilteredVectorSpace({Integer(0):[(Integer(1),Integer(1))]});  F
QQ^1 >= 0 in QQ^2
>>> F.is_exhaustive()
False
>>> G = FilteredVectorSpace(Integer(2), Integer(0));  G
QQ^2 >= 0
```

(continues on next page)

(continued from previous page)

```
>>> G.is_exhaustive()
True
```

is_separating()

Return whether the filtration is separating.

A filtration $\{F_d\}$ in an ambient vector space V is exhaustive if $\cap F_d = 0$. See also *is_exhaustive()*.

OUTPUT: boolean

EXAMPLES:

```
sage: F = FilteredVectorSpace({0:[(1,1)]});  F
QQ^1 >= 0 in QQ^2
sage: F.is_separating()
True
sage: G = FilteredVectorSpace({0:[(1,1,0)], oo:[(0,0,1)]});  G
QQ^2 >= QQ^1 in QQ^3
sage: G.is_separating()
False
```

```
>>> from sage.all import *
>>> F = FilteredVectorSpace({Integer(0):[(Integer(1),Integer(1))]});  F
QQ^1 >= 0 in QQ^2
>>> F.is_separating()
True
>>> G = FilteredVectorSpace({Integer(0):[(Integer(1),Integer(1),Integer(0))],␣
↪oo:[(Integer(0),Integer(0),Integer(1))]});  G
QQ^2 >= QQ^1 in QQ^3
>>> G.is_separating()
False
```

max_degree()

Return the highest degree of the filtration.

OUTPUT:

Integer or minus infinity. The smallest degree of the filtration such that the filtration is constant to the right.

EXAMPLES:

```
sage: FilteredVectorSpace(1, 3).max_degree()
4
sage: FilteredVectorSpace({0:[[1]]}).max_degree()
1
sage: FilteredVectorSpace(3).max_degree()
-Infinity
```

```
>>> from sage.all import *
>>> FilteredVectorSpace(Integer(1), Integer(3)).max_degree()
4
>>> FilteredVectorSpace({Integer(0):[[Integer(1)]]}).max_degree()
1
>>> FilteredVectorSpace(Integer(3)).max_degree()
-Infinity
```

min_degree()

Return the lowest degree of the filtration.

OUTPUT:

Integer or plus infinity. The largest degree d of the (descending) filtration such that the filtered vector space F_d is still equal to $F_{-\infty}$.

EXAMPLES:

```
sage: FilteredVectorSpace(1, 3).min_degree()
3
sage: FilteredVectorSpace(2).min_degree()
+Infinity
```

```
>>> from sage.all import *
>>> FilteredVectorSpace(Integer(1), Integer(3)).min_degree()
3
>>> FilteredVectorSpace(Integer(2)).min_degree()
+Infinity
```

presentation()

Return a presentation in term of generators of various degrees.

OUTPUT:

A pair consisting of generators and a filtration suitable as input to *construct_from_generators_indices()*.

EXAMPLES:

```
sage: rays = [(1,0), (1,1), (1,2), (-1,-1)]
sage: F = FilteredVectorSpace(rays, {0:[1, 2], 2:[3]});  F
QQ^2 >= QQ^1 >= QQ^1 >= 0
sage: F.presentation()
(((0, 1), (1, 0), (1, 1)), {0: (1, 0), 2: (2,), +Infinity: ()})
```

```
>>> from sage.all import *
>>> rays = [(Integer(1),Integer(0)), (Integer(1),Integer(1)), (Integer(1),
↪Integer(2)), (-Integer(1),-Integer(1))]
>>> F = FilteredVectorSpace(rays, {Integer(0):[Integer(1), Integer(2)],␣
↪Integer(2):[Integer(3)]});  F
QQ^2 >= QQ^1 >= QQ^1 >= 0
>>> F.presentation()
(((0, 1), (1, 0), (1, 1)), {0: (1, 0), 2: (2,), +Infinity: ()})
```

random_deformation(*epsilon=None*)

Return a random deformation.

INPUT:

- `epsilon` – a number in the base ring

OUTPUT:

A new filtered vector space where the generators of the subspaces are moved by `epsilon` times a random vector.

EXAMPLES:

```
sage: gens = identity_matrix(3).rows()
sage: F = FilteredVectorSpace(gens, {0:[0,1,2], 2:[0]});  F
QQ^3 >= QQ^1 >= QQ^1 >= 0
sage: F.get_degree(2)
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 0 0]
sage: G = F.random_deformation(1/50);  G
QQ^3 >= QQ^1 >= QQ^1 >= 0
sage: D = G.get_degree(2)
sage: D.degree()
3
sage: v = D.basis_matrix()[0]
sage: v[0]
1

sage: while F.random_deformation(1/50).get_degree(2).matrix() == matrix([1, 0,
↪ 0]):
....:     pass
```

```
>>> from sage.all import *
>>> gens = identity_matrix(Integer(3)).rows()
>>> F = FilteredVectorSpace(gens, {Integer(0):[Integer(0),Integer(1),
↪Integer(2)], Integer(2):[Integer(0)]});  F
QQ^3 >= QQ^1 >= QQ^1 >= 0
>>> F.get_degree(Integer(2))
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 0 0]
>>> G = F.random_deformation(Integer(1)/Integer(50));  G
QQ^3 >= QQ^1 >= QQ^1 >= 0
>>> D = G.get_degree(Integer(2))
>>> D.degree()
3
>>> v = D.basis_matrix()[Integer(0)]
>>> v[Integer(0)]
1

>>> while F.random_deformation(Integer(1)/Integer(50)).get_degree(Integer(2)).
↪matrix() == matrix([Integer(1), Integer(0), Integer(0)]):
...     pass
```

shift(*deg*)

Return a filtered vector space with degrees shifted by a constant.

EXAMPLES:

```
sage: gens = identity_matrix(3).rows()
sage: F = FilteredVectorSpace(gens, {0:[0,1,2], 2:[0]});  F
QQ^3 >= QQ^1 >= QQ^1 >= 0
sage: F.support()
(0, 2)
sage: F.shift(-5).support()
```

(continues on next page)

(continued from previous page)

```
(-5, -3)
```

```
>>> from sage.all import *
>>> gens = identity_matrix(Integer(3)).rows()
>>> F = FilteredVectorSpace(gens, {Integer(0):[Integer(0),Integer(1),
↪Integer(2)], Integer(2):[Integer(0)]});  F
QQ^3 >= QQ^1 >= QQ^1 >= 0
>>> F.support()
(0, 2)
>>> F.shift(-Integer(5)).support()
(-5, -3)
```

support()

Return the degrees in which there are non-trivial generators.

OUTPUT:

A tuple of integers (and plus infinity) in ascending order. The last entry is plus infinity if and only if the filtration is not separating (see *is_separating()*).

EXAMPLES:

```
sage: G = FilteredVectorSpace({0:[(1,1,0)], 3:[(0,1,0)]});  G
QQ^2 >= QQ^1 >= QQ^1 >= QQ^1 >= 0 in QQ^3
sage: G.support()
(0, 3)

sage: G = FilteredVectorSpace({0:[(1,1,0)], 3:[(0,1,0)], oo:[(0,0,1)]});  G
QQ^3 >= QQ^2 >= QQ^2 >= QQ^2 >= QQ^1
sage: G.support()
(0, 3, +Infinity)
```

```
>>> from sage.all import *
>>> G = FilteredVectorSpace({Integer(0):[(Integer(1),Integer(1),Integer(0))],␣
↪Integer(3):[(Integer(0),Integer(1),Integer(0))]});  G
QQ^2 >= QQ^1 >= QQ^1 >= QQ^1 >= 0 in QQ^3
>>> G.support()
(0, 3)

>>> G = FilteredVectorSpace({Integer(0):[(Integer(1),Integer(1),Integer(0))],␣
↪Integer(3):[(Integer(0),Integer(1),Integer(0))], oo:[(Integer(0),Integer(0),
↪Integer(1))]});  G
QQ^3 >= QQ^2 >= QQ^2 >= QQ^2 >= QQ^1
>>> G.support()
(0, 3, +Infinity)
```

symmetric_power(n)

Return the n-th graded symmetric power.

INPUT:

- `n` – integer; symmetric product of how many copies of `self`

OUTPUT:

The graded symmetric product, that is, the symmetrization of a generator of degree d_1 with a generator in degree d_2 has degree $d_1 + d_2$.

EXAMPLES:

```
sage: F = FilteredVectorSpace(1, 1) + FilteredVectorSpace(1, 2);  F
QQ^2 >= QQ^1 >= 0
sage: F.symmetric_power(2)
QQ^3 >= QQ^2 >= QQ^1 >= 0
```

```
>>> from sage.all import *
>>> F = FilteredVectorSpace(Integer(1), Integer(1)) +␣
↪FilteredVectorSpace(Integer(1), Integer(2));  F
QQ^2 >= QQ^1 >= 0
>>> F.symmetric_power(Integer(2))
QQ^3 >= QQ^2 >= QQ^1 >= 0
```

tensor_product(*other*)

Return the graded tensor product.

INPUT:

- `other` – a filtered vector space

OUTPUT:

The graded tensor product, that is, the tensor product of a generator of degree d_1 with a generator in degree d_2 has degree $d_1 + d_2$.

EXAMPLES:

```
sage: F1 = FilteredVectorSpace(1, 1)
sage: F2 = FilteredVectorSpace(1, 2)
sage: F1.tensor_product(F2)
QQ^1 >= 0
sage: F1 * F2
QQ^1 >= 0

sage: F1.min_degree()
1
sage: F2.min_degree()
2
sage: (F1*F2).min_degree()
3
```

```
>>> from sage.all import *
>>> F1 = FilteredVectorSpace(Integer(1), Integer(1))
>>> F2 = FilteredVectorSpace(Integer(1), Integer(2))
>>> F1.tensor_product(F2)
QQ^1 >= 0
>>> F1 * F2
QQ^1 >= 0

>>> F1.min_degree()
1
>>> F2.min_degree()
```

(continues on next page)

(continued from previous page)

```
2
>>> (F1*F2).min_degree()
3
```

A suitable base ring is chosen if they do not match:

```
sage: v = [(1,0), (0,1)]
sage: F1 = FilteredVectorSpace(v, {0:[0], 1:[1]}, base_ring=QQ)
sage: F2 = FilteredVectorSpace(v, {0:[0], 1:[1]}, base_ring=RDF)
sage: F1 * F2                                                                 #␣
↪needs scipy
RDF^4 >= RDF^3 >= RDF^1 >= 0
```

```
>>> from sage.all import *
>>> v = [(Integer(1),Integer(0)), (Integer(0),Integer(1))]
>>> F1 = FilteredVectorSpace(v, {Integer(0):[Integer(0)],␣
↪Integer(1):[Integer(1)]}, base_ring=QQ)
>>> F2 = FilteredVectorSpace(v, {Integer(0):[Integer(0)],␣
↪Integer(1):[Integer(1)]}, base_ring=RDF)
>>> F1 * F2                                                                   #␣
↪needs scipy
RDF^4 >= RDF^3 >= RDF^1 >= 0
```

wedge(n)

Return the n-th graded exterior power.

INPUT:

- `n` – integer; exterior product of how many copies of `self`

OUTPUT:

The graded exterior product, that is, the wedge product of a generator of degree d_1 with a generator in degree d_2 has degree $d_1 + d_2$.

EXAMPLES:

```
sage: # needs sage.groups
sage: F = FilteredVectorSpace(1, 1) + FilteredVectorSpace(1, 2);  F
QQ^2 >= QQ^1 >= 0
sage: F.exterior_power(1)
QQ^2 >= QQ^1 >= 0
sage: F.exterior_power(2)
QQ^1 >= 0
sage: F.exterior_power(3)
0
sage: F.wedge(2)
QQ^1 >= 0
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> F = FilteredVectorSpace(Integer(1), Integer(1)) +␣
↪FilteredVectorSpace(Integer(1), Integer(2));  F
QQ^2 >= QQ^1 >= 0
```

(continues on next page)

(continued from previous page)

```
>>> F.exterior_power(Integer(1))
QQ^2 >= QQ^1 >= 0
>>> F.exterior_power(Integer(2))
QQ^1 >= 0
>>> F.exterior_power(Integer(3))
0
>>> F.wedge(Integer(2))
QQ^1 >= 0
```

sage.modules.filtered_vector_space.**construct_from_dim_degree**(*dim*, *max_degree*, *base_ring*, *check*)

Construct a filtered vector space.

INPUT:

- `dim` – integer; the dimension
- `max_degree` – integer or infinity; the maximal degree where the vector subspace of the filtration is still the entire space

EXAMPLES:

```
sage: V = FilteredVectorSpace(2, 5);  V
QQ^2 >= 0
sage: V.get_degree(5)
Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1 0]
[0 1]
sage: V.get_degree(6)
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]

sage: FilteredVectorSpace(2, oo)
QQ^2
sage: FilteredVectorSpace(2, -oo)
0 in QQ^2
```

```
>>> from sage.all import *
>>> V = FilteredVectorSpace(Integer(2), Integer(5));  V
QQ^2 >= 0
>>> V.get_degree(Integer(5))
Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1 0]
[0 1]
>>> V.get_degree(Integer(6))
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]

>>> FilteredVectorSpace(Integer(2), oo)
QQ^2
```

(continues on next page)

(continued from previous page)

```
>>> FilteredVectorSpace(Integer(2), -oo)
0 in QQ^2
```

sage.modules.filtered_vector_space.**construct_from_generators**(*filtration*, *base_ring*, *check*)

Construct a filtered vector space.

INPUT:

- `filtration` – dictionary of filtration steps. Each filtration step is a pair consisting of an integer degree and a list/tuple/iterable of vector space generators. The integer `degree` stipulates that all filtration steps of degree higher or equal than `degree` (up to the next filtration step) are said subspace.

EXAMPLES:

```
sage: from sage.modules.filtered_vector_space import construct_from_generators
sage: r = [1, 2]
sage: construct_from_generators({1:[r]}, QQ, True)
QQ^1 >= 0 in QQ^2
```

```
>>> from sage.all import *
>>> from sage.modules.filtered_vector_space import construct_from_generators
>>> r = [Integer(1), Integer(2)]
>>> construct_from_generators({Integer(1):[r]}, QQ, True)
QQ^1 >= 0 in QQ^2
```

sage.modules.filtered_vector_space.**construct_from_generators_indices**(*generators*, *filtration*, *base_ring*, *check*)

Construct a filtered vector space.

INPUT:

- `generators` – list/tuple/iterable of vectors, or something convertible to them. The generators spanning various subspaces.
- `filtration` – list or iterable of filtration steps. Each filtration step is a pair `(degree, ray_indices)`. The `ray_indices` are a list or iterable of ray indices, which span a subspace of the vector space. The integer `degree` stipulates that all filtration steps of degree higher or equal than `degree` (up to the next filtration step) are said subspace.

EXAMPLES:

```
sage: from sage.modules.filtered_vector_space import construct_from_generators_
↪indices
sage: gens = [(1,0), (0,1), (-1,-1)]
sage: V = construct_from_generators_indices(gens, {1:[0,1], 3:[1]}, QQ, True);  V
QQ^2 >= QQ^1 >= QQ^1 >= 0
```

```
>>> from sage.all import *
>>> from sage.modules.filtered_vector_space import construct_from_generators_
↪indices
>>> gens = [(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-Integer(1),-
↪Integer(1))]
>>> V = construct_from_generators_indices(gens, {Integer(1):[Integer(0),
↪Integer(1)], Integer(3):[Integer(1)]}, QQ, True);  V
QQ^2 >= QQ^1 >= QQ^1 >= 0
```

sage.modules.filtered_vector_space.**is_FilteredVectorSpace**(*X*)

Test whether X is a filtered vector space.

This function is for library use only.

INPUT:

- X – anything

OUTPUT: boolean

EXAMPLES:

```
sage: from sage.modules.filtered_vector_space import is_FilteredVectorSpace
sage: V = FilteredVectorSpace(2, 1)
sage: is_FilteredVectorSpace(V)
doctest:warning...:
DeprecationWarning: the function is_FilteredVectorSpace is deprecated;
use 'isinstance(..., FilteredVectorSpace_class)' instead
See https://github.com/sagemath/sage/issues/37924 for details.
True
sage: is_FilteredVectorSpace('ceci n'est pas une pipe')
False
```

```
>>> from sage.all import *
>>> from sage.modules.filtered_vector_space import is_FilteredVectorSpace
>>> V = FilteredVectorSpace(Integer(2), Integer(1))
>>> is_FilteredVectorSpace(V)
doctest:warning...:
DeprecationWarning: the function is_FilteredVectorSpace is deprecated;
use 'isinstance(..., FilteredVectorSpace_class)' instead
See https://github.com/sagemath/sage/issues/37924 for details.
True
>>> is_FilteredVectorSpace('ceci n'est pas une pipe')
False
```

sage.modules.filtered_vector_space.**normalize_degree**(*deg*)

Normalize the degree.

- deg – something that defines the degree (either integer or infinity)

OUTPUT: plus/minus infinity or a Sage integer

EXAMPLES:

```
sage: from sage.modules.filtered_vector_space import normalize_degree
sage: type(normalize_degree(int(1)))
<class 'sage.rings.integer.Integer'>
sage: normalize_degree(oo)
+Infinity
```

```
>>> from sage.all import *
>>> from sage.modules.filtered_vector_space import normalize_degree
>>> type(normalize_degree(int(Integer(1))))
<class 'sage.rings.integer.Integer'>
>>> normalize_degree(oo)
+Infinity
```

6.6 Multiple $\mathbb{Z}$-graded filtrations of a single vector space

See `filtered_vector_space` for simply graded vector spaces. This module implements the analog but for a collection of filtrations of the same vector space.

The basic syntax to use it is a dictionary whose keys are some arbitrary indexing set and values are *FilteredVectorSpace()*

```
sage: F1 = FilteredVectorSpace(2, 1)
sage: F2 = FilteredVectorSpace({0:[(1,0)], 2:[(2,3)]})
sage: V = MultiFilteredVectorSpace({'first':F1, 'second':F2})
sage: V
Filtrations
     first: QQ^2 >= QQ^2 >=  0   >= 0
    second: QQ^2 >= QQ^1 >= QQ^1 >= 0

sage: V.index_set()   # random output
{'second', 'first'}
sage: sorted(V.index_set())
['first', 'second']

sage: V.get_filtration('first')
QQ^2 >=  0
sage: V.get_degree('second', 1)
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[  1 3/2]
```

```
>>> from sage.all import *
>>> F1 = FilteredVectorSpace(Integer(2), Integer(1))
>>> F2 = FilteredVectorSpace({Integer(0):[(Integer(1),Integer(0))],␣
↪Integer(2):[(Integer(2),Integer(3))]})
>>> V = MultiFilteredVectorSpace({'first':F1, 'second':F2})
>>> V
Filtrations
     first: QQ^2 >= QQ^2 >=  0   >= 0
    second: QQ^2 >= QQ^1 >= QQ^1 >= 0

>>> V.index_set()   # random output
{'second', 'first'}
>>> sorted(V.index_set())
['first', 'second']

>>> V.get_filtration('first')
QQ^2 >=  0
>>> V.get_degree('second', Integer(1))
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[  1 3/2]
```

sage.modules.multi_filtered_vector_space.**MultiFilteredVectorSpace**(*arg*, *base_ring=None*, *check=True*)

Contstruct a multi-filtered vector space.

INPUT:

- `arg` – either a non-empty dictionary of filtrations or an integer. The latter is interpreted as the vector space dimension, and the indexing set of the filtrations is empty.
- `base_ring` – a field (default: `'None'`). The base field of the vector space. Must be a field. If not specified, the base field is derived from the filtrations.
- `check` – boolean (default: `True`); whether to perform consistency checks

EXAMPLES:

```
sage: MultiFilteredVectorSpace(3, QQ)
Unfiltered QQ^3

sage: F1 = FilteredVectorSpace(2, 1)
sage: F2 = FilteredVectorSpace(2, 3)
sage: V = MultiFilteredVectorSpace({1:F1, 2:F2});  V
Filtrations
    1: QQ^2 >=  0   >=  0   >= 0
    2: QQ^2 >= QQ^2 >= QQ^2 >= 0
```

```
>>> from sage.all import *
>>> MultiFilteredVectorSpace(Integer(3), QQ)
Unfiltered QQ^3

>>> F1 = FilteredVectorSpace(Integer(2), Integer(1))
>>> F2 = FilteredVectorSpace(Integer(2), Integer(3))
>>> V = MultiFilteredVectorSpace({Integer(1):F1, Integer(2):F2});  V
Filtrations
    1: QQ^2 >=  0   >=  0   >= 0
    2: QQ^2 >= QQ^2 >= QQ^2 >= 0
```

class sage.modules.multi_filtered_vector_space.**MultiFilteredVectorSpace_class**(*base_ring*, *dim*, *filtrations*, *check=True*)

Bases: *FreeModule_ambient_field*

Python constructor.

Warning

Use *MultiFilteredVectorSpace()* to construct multi-filtered vector spaces.

INPUT:

- `base_ring` – the base ring
- `dim` – integer; the dimension of the ambient vector space
- `filtrations` – dictionary whose values are filtrations
- `check` – boolean (default: `True`); whether to perform additional consistency checks

EXAMPLES:

```
sage: F1 = FilteredVectorSpace(2, 1)
sage: F2 = FilteredVectorSpace(2, 3)
sage: V = MultiFilteredVectorSpace({1:F1, 2:F2});  V
Filtrations
    1: QQ^2 >=  0   >=  0   >= 0
    2: QQ^2 >= QQ^2 >= QQ^2 >= 0
```

```
>>> from sage.all import *
>>> F1 = FilteredVectorSpace(Integer(2), Integer(1))
>>> F2 = FilteredVectorSpace(Integer(2), Integer(3))
>>> V = MultiFilteredVectorSpace({Integer(1):F1, Integer(2):F2});  V
Filtrations
    1: QQ^2 >=  0   >=  0   >= 0
    2: QQ^2 >= QQ^2 >= QQ^2 >= 0
```

ambient_vector_space()

Return the ambient (unfiltered) vector space.

OUTPUT: a vector space

EXAMPLES:

```
sage: V = FilteredVectorSpace(2, 0)
sage: W = FilteredVectorSpace(2, 2)
sage: F = MultiFilteredVectorSpace({'a':V, 'b':W})
sage: F.ambient_vector_space()
Vector space of dimension 2 over Rational Field
```

```
>>> from sage.all import *
>>> V = FilteredVectorSpace(Integer(2), Integer(0))
>>> W = FilteredVectorSpace(Integer(2), Integer(2))
>>> F = MultiFilteredVectorSpace({'a':V, 'b':W})
>>> F.ambient_vector_space()
Vector space of dimension 2 over Rational Field
```

change_ring(*base_ring*)

Return the same multi-filtration over a different base ring.

INPUT:

- base_ring – the new base ring

OUTPUT:

This method returns a new multi-filtered vector space whose subspaces are defined by the same generators but over a different base ring.

EXAMPLES:

```
sage: V = FilteredVectorSpace(2, 0)
sage: W = FilteredVectorSpace(2, 2)
sage: F = MultiFilteredVectorSpace({'a':V, 'b':W});  F
Filtrations
    a: QQ^2 >=  0   >=  0   >= 0
    b: QQ^2 >= QQ^2 >= QQ^2 >= 0
```

(continues on next page)

(continued from previous page)

```
sage: F.change_ring(RDF)
Filtrations
    a: RDF^2 >=   0   >=   0   >= 0
    b: RDF^2 >= RDF^2 >= RDF^2 >= 0

sage: MultiFilteredVectorSpace(3, base_ring=QQ).change_ring(RR)
Unfiltered RR^3
```

```
>>> from sage.all import *
>>> V = FilteredVectorSpace(Integer(2), Integer(0))
>>> W = FilteredVectorSpace(Integer(2), Integer(2))
>>> F = MultiFilteredVectorSpace({'a':V, 'b':W});  F
Filtrations
    a: QQ^2 >=  0   >=  0   >= 0
    b: QQ^2 >= QQ^2 >= QQ^2 >= 0
>>> F.change_ring(RDF)
Filtrations
    a: RDF^2 >=   0   >=   0   >= 0
    b: RDF^2 >= RDF^2 >= RDF^2 >= 0

>>> MultiFilteredVectorSpace(Integer(3), base_ring=QQ).change_ring(RR)
Unfiltered RR^3
```

direct_sum(*other*)

Return the direct sum.

INPUT:

- `other` – a multi-filtered vector space with the same *index_set()*

OUTPUT:

The direct sum as a multi-filtered vector space. See *direct_sum()*.

EXAMPLES:

```
sage: F1 = FilteredVectorSpace(2, 1)
sage: F2 = FilteredVectorSpace(1, 3) + FilteredVectorSpace(1,0)
sage: V = MultiFilteredVectorSpace({'a':F1, 'b':F2})
sage: G1 = FilteredVectorSpace(1, 1)
sage: G2 = FilteredVectorSpace(1, 3)
sage: W = MultiFilteredVectorSpace({'a':G1, 'b':G2})
sage: V.direct_sum(W)
Filtrations
    a: QQ^3 >= QQ^3 >=  0   >=  0   >= 0
    b: QQ^3 >= QQ^2 >= QQ^2 >= QQ^2 >= 0
sage: V + W   # syntactic sugar
Filtrations
    a: QQ^3 >= QQ^3 >=  0   >=  0   >= 0
    b: QQ^3 >= QQ^2 >= QQ^2 >= QQ^2 >= 0
```

```
>>> from sage.all import *
>>> F1 = FilteredVectorSpace(Integer(2), Integer(1))
>>> F2 = FilteredVectorSpace(Integer(1), Integer(3)) +␣
```

(continues on next page)

(continued from previous page)

```
↪FilteredVectorSpace(Integer(1),Integer(0))
>>> V = MultiFilteredVectorSpace({'a':F1, 'b':F2})
>>> G1 = FilteredVectorSpace(Integer(1), Integer(1))
>>> G2 = FilteredVectorSpace(Integer(1), Integer(3))
>>> W = MultiFilteredVectorSpace({'a':G1, 'b':G2})
>>> V.direct_sum(W)
Filtrations
    a: QQ^3 >= QQ^3 >=  0   >=  0   >= 0
    b: QQ^3 >= QQ^2 >= QQ^2 >= QQ^2 >= 0
>>> V + W   # syntactic sugar
Filtrations
    a: QQ^3 >= QQ^3 >=  0   >=  0   >= 0
    b: QQ^3 >= QQ^2 >= QQ^2 >= QQ^2 >= 0
```

dual()

Return the dual.

OUTPUT:

The dual as a multi-filtered vector space. See *dual()*.

EXAMPLES:

```
sage: F1 = FilteredVectorSpace(2, 1)
sage: F2 = FilteredVectorSpace(1, 3) + FilteredVectorSpace(1,0)
sage: V = MultiFilteredVectorSpace({'a':F1, 'b':F2})
sage: V.dual()
Filtrations
    a: QQ^2 >= QQ^2 >= QQ^2 >=  0   >= 0
    b: QQ^2 >= QQ^1 >= QQ^1 >= QQ^1 >= 0
```

```
>>> from sage.all import *
>>> F1 = FilteredVectorSpace(Integer(2), Integer(1))
>>> F2 = FilteredVectorSpace(Integer(1), Integer(3)) +␣
↪FilteredVectorSpace(Integer(1),Integer(0))
>>> V = MultiFilteredVectorSpace({'a':F1, 'b':F2})
>>> V.dual()
Filtrations
    a: QQ^2 >= QQ^2 >= QQ^2 >=  0   >= 0
    b: QQ^2 >= QQ^1 >= QQ^1 >= QQ^1 >= 0
```

exterior_power(n)

Return the n-th graded exterior power.

INPUT:

- `n` – integer; exterior product of how many copies of `self`

OUTPUT:

The exterior power as a multi-filtered vector space. See *exterior_power()*.

EXAMPLES:

```
sage: F1 = FilteredVectorSpace(2, 1)
sage: F2 = FilteredVectorSpace(1, 3) + FilteredVectorSpace(1,0)
```

(continues on next page)

(continued from previous page)

```
sage: V = MultiFilteredVectorSpace({'a':F1, 'b':F2})
sage: V.exterior_power(2)  # long time
Filtrations
    a: QQ^1 >=  0   >= 0
    b: QQ^1 >= QQ^1 >= 0
```

```
>>> from sage.all import *
>>> F1 = FilteredVectorSpace(Integer(2), Integer(1))
>>> F2 = FilteredVectorSpace(Integer(1), Integer(3)) +␣
↪FilteredVectorSpace(Integer(1),Integer(0))
>>> V = MultiFilteredVectorSpace({'a':F1, 'b':F2})
>>> V.exterior_power(Integer(2))  # long time
Filtrations
    a: QQ^1 >=  0   >= 0
    b: QQ^1 >= QQ^1 >= 0
```

get_degree(*key*, *deg*)

Return one filtered vector space.

INPUT:

- `key` – an element of the *index_set()*; specifies which filtration
- `d` – integer; the desired degree of the filtration

OUTPUT:

The vector space of degree `deg` in the filtration indexed by `key` as subspace of the ambient space.

EXAMPLES:

```
sage: F1 = FilteredVectorSpace(2, 1)
sage: F2 = FilteredVectorSpace(2, 3)
sage: V = MultiFilteredVectorSpace({1:F1, 2:F2})
sage: V.get_degree(2, 0)
Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1 0]
[0 1]
```

```
>>> from sage.all import *
>>> F1 = FilteredVectorSpace(Integer(2), Integer(1))
>>> F2 = FilteredVectorSpace(Integer(2), Integer(3))
>>> V = MultiFilteredVectorSpace({Integer(1):F1, Integer(2):F2})
>>> V.get_degree(Integer(2), Integer(0))
Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1 0]
[0 1]
```

get_filtration(*key*)

Return the filtration indexed by `key`.

OUTPUT: a filtered vector space

EXAMPLES:

```
sage: F1 = FilteredVectorSpace(2, 1)
sage: F2 = FilteredVectorSpace(2, 3)
sage: V = MultiFilteredVectorSpace({1:F1, 2:F2})
sage: V.get_filtration(2)
QQ^2 >= 0
```

```
>>> from sage.all import *
>>> F1 = FilteredVectorSpace(Integer(2), Integer(1))
>>> F2 = FilteredVectorSpace(Integer(2), Integer(3))
>>> V = MultiFilteredVectorSpace({Integer(1):F1, Integer(2):F2})
>>> V.get_filtration(Integer(2))
QQ^2 >= 0
```

graded(*key*, *deg*)

Return the associated graded vector space.

INPUT:

- `key` – an element of the *index_set()*; specifies which filtration
- `d` – integer; the desired degree of the filtration

OUTPUT:

The quotient $G_d = F_d/F_{d+1}$ of the filtration F corresponding to `key`.

EXAMPLES:

```
sage: F1 = FilteredVectorSpace(2, 1)
sage: F2 = FilteredVectorSpace(1, 3) + FilteredVectorSpace(1,0)
sage: V = MultiFilteredVectorSpace({1:F1, 2:F2})
sage: V.graded(2, 3)
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:
[1 0]
W: Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
```

```
>>> from sage.all import *
>>> F1 = FilteredVectorSpace(Integer(2), Integer(1))
>>> F2 = FilteredVectorSpace(Integer(1), Integer(3)) +␣
↪FilteredVectorSpace(Integer(1),Integer(0))
>>> V = MultiFilteredVectorSpace({Integer(1):F1, Integer(2):F2})
>>> V.graded(Integer(2), Integer(3))
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:
[1 0]
W: Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
```

index_set()

Return the allowed indices for the different filtrations.

OUTPUT: set

EXAMPLES:

```
sage: F1 = FilteredVectorSpace(2, 1)
sage: F2 = FilteredVectorSpace(2, 3)
sage: V = MultiFilteredVectorSpace({1:F1, 2:F2})
sage: V.index_set()
{1, 2}
```

```
>>> from sage.all import *
>>> F1 = FilteredVectorSpace(Integer(2), Integer(1))
>>> F2 = FilteredVectorSpace(Integer(2), Integer(3))
>>> V = MultiFilteredVectorSpace({Integer(1):F1, Integer(2):F2})
>>> V.index_set()
{1, 2}
```

is_constant()

Return whether the multi-filtration is constant.

OUTPUT: boolean; whether the each filtration is constant, see *is_constant()*

EXAMPLES:

```
sage: V = FilteredVectorSpace(2, 0)
sage: W = FilteredVectorSpace(2, 2)
sage: F = MultiFilteredVectorSpace({'a':V, 'b':W});  F
Filtrations
    a: QQ^2 >=  0   >=  0   >= 0
    b: QQ^2 >= QQ^2 >= QQ^2 >= 0
sage: F.is_constant()
False
```

```
>>> from sage.all import *
>>> V = FilteredVectorSpace(Integer(2), Integer(0))
>>> W = FilteredVectorSpace(Integer(2), Integer(2))
>>> F = MultiFilteredVectorSpace({'a':V, 'b':W});  F
Filtrations
    a: QQ^2 >=  0   >=  0   >= 0
    b: QQ^2 >= QQ^2 >= QQ^2 >= 0
>>> F.is_constant()
False
```

is_exhaustive()

Return whether the multi-filtration is exhaustive.

A filtration $\{F_d\}$ in an ambient vector space V is exhaustive if $\cup F_d = V$. See also *is_separating()*.

OUTPUT: boolean; whether each filtration is constant, see *is_exhaustive()*

EXAMPLES:

```
sage: F1 = FilteredVectorSpace(2, 1)
sage: F2 = FilteredVectorSpace(2, 3)
sage: V = MultiFilteredVectorSpace({1:F1, 2:F2})
```

(continues on next page)

(continued from previous page)

```
sage: V.is_exhaustive()
True
```

```
>>> from sage.all import *
>>> F1 = FilteredVectorSpace(Integer(2), Integer(1))
>>> F2 = FilteredVectorSpace(Integer(2), Integer(3))
>>> V = MultiFilteredVectorSpace({Integer(1):F1, Integer(2):F2})
>>> V.is_exhaustive()
True
```

is_separating()

Return whether the multi-filtration is separating.

A filtration $\{F_d\}$ in an ambient vector space V is exhaustive if $\cap F_d = 0$. See also *is_exhaustive()*.

OUTPUT: boolean; whether each filtration is separating, see *is_separating()*

EXAMPLES:

```
sage: F1 = FilteredVectorSpace(2, 1)
sage: F2 = FilteredVectorSpace(2, 3)
sage: V = MultiFilteredVectorSpace({1:F1, 2:F2})
sage: V.is_separating()
True
```

```
>>> from sage.all import *
>>> F1 = FilteredVectorSpace(Integer(2), Integer(1))
>>> F2 = FilteredVectorSpace(Integer(2), Integer(3))
>>> V = MultiFilteredVectorSpace({Integer(1):F1, Integer(2):F2})
>>> V.is_separating()
True
```

max_degree()

Return the highest degree of the filtration.

OUTPUT:

Integer or minus infinity. The smallest degree of the filtrations such that the filtrations are constant to the right.

EXAMPLES:

```
sage: F1 = FilteredVectorSpace(2, 1)
sage: F2 = FilteredVectorSpace(2, 3)
sage: V = MultiFilteredVectorSpace({1:F1, 2:F2})
sage: V.max_degree()
4
```

```
>>> from sage.all import *
>>> F1 = FilteredVectorSpace(Integer(2), Integer(1))
>>> F2 = FilteredVectorSpace(Integer(2), Integer(3))
>>> V = MultiFilteredVectorSpace({Integer(1):F1, Integer(2):F2})
>>> V.max_degree()
4
```

min_degree()

Return the lowest degree of the filtration.

OUTPUT:

Integer or plus infinity. The largest degree d of the (descending) filtrations such that, for each individual filtration, the filtered vector space F_d still equal to $F_{-\infty}$.

EXAMPLES:

```
sage: F1 = FilteredVectorSpace(2, 1)
sage: F2 = FilteredVectorSpace(2, 3)
sage: V = MultiFilteredVectorSpace({1:F1, 2:F2})
sage: V.min_degree()
1
```

```
>>> from sage.all import *
>>> F1 = FilteredVectorSpace(Integer(2), Integer(1))
>>> F2 = FilteredVectorSpace(Integer(2), Integer(3))
>>> V = MultiFilteredVectorSpace({Integer(1):F1, Integer(2):F2})
>>> V.min_degree()
1
```

random_deformation(*epsilon=None*)

Return a random deformation.

INPUT:

- `epsilon` – a number in the base ring

OUTPUT:

A new multi-filtered vector space where the generating vectors of subspaces are moved by `epsilon` times a random vector.

EXAMPLES:

```
sage: F1 = FilteredVectorSpace(2, 1)
sage: F2 = FilteredVectorSpace(1, 3) + FilteredVectorSpace(1,0)
sage: V = MultiFilteredVectorSpace({'a':F1, 'b':F2})
sage: V.get_degree('b',1)
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 0]
sage: D = V.random_deformation(1/100).get_degree('b',1)
sage: D.degree()
2
sage: D.dimension()
1
sage: D.matrix()[0, 0]
1

sage: while V.random_deformation(1/100).get_degree('b',1).matrix() ==␣
↪matrix([1, 0]):
....:     pass
```

```
>>> from sage.all import *
>>> F1 = FilteredVectorSpace(Integer(2), Integer(1))
>>> F2 = FilteredVectorSpace(Integer(1), Integer(3)) +␣
↪FilteredVectorSpace(Integer(1),Integer(0))
>>> V = MultiFilteredVectorSpace({'a':F1, 'b':F2})
>>> V.get_degree('b',Integer(1))
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 0]
>>> D = V.random_deformation(Integer(1)/Integer(100)).get_degree('b',
↪Integer(1))
>>> D.degree()
2
>>> D.dimension()
1
>>> D.matrix()[Integer(0), Integer(0)]
1

>>> while V.random_deformation(Integer(1)/Integer(100)).get_degree('b',
↪Integer(1)).matrix() == matrix([Integer(1), Integer(0)]):
...     pass
```

shift(*deg*)

Return a filtered vector space with degrees shifted by a constant.

OUTPUT:

The shift of `self`. See *shift()*.

EXAMPLES:

```
sage: F1 = FilteredVectorSpace(2, 1)
sage: F2 = FilteredVectorSpace(1, 3) + FilteredVectorSpace(1,0)
sage: V = MultiFilteredVectorSpace({'a':F1, 'b':F2})
sage: V.support()
(0, 1, 3)
sage: V.shift(-5).support()
(-5, -4, -2)
```

```
>>> from sage.all import *
>>> F1 = FilteredVectorSpace(Integer(2), Integer(1))
>>> F2 = FilteredVectorSpace(Integer(1), Integer(3)) +␣
↪FilteredVectorSpace(Integer(1),Integer(0))
>>> V = MultiFilteredVectorSpace({'a':F1, 'b':F2})
>>> V.support()
(0, 1, 3)
>>> V.shift(-Integer(5)).support()
(-5, -4, -2)
```

support()

Return the degrees in which there are non-trivial generators.

OUTPUT:

A tuple of integers (and plus infinity) in ascending order. The last entry is plus infinity if and only if the filtration is not separating (see *is_separating()*).

EXAMPLES:

```
sage: F1 = FilteredVectorSpace(2, 1)
sage: F2 = FilteredVectorSpace(2, 3)
sage: V = MultiFilteredVectorSpace({1:F1, 2:F2})
sage: V.support()
(1, 3)
```

```
>>> from sage.all import *
>>> F1 = FilteredVectorSpace(Integer(2), Integer(1))
>>> F2 = FilteredVectorSpace(Integer(2), Integer(3))
>>> V = MultiFilteredVectorSpace({Integer(1):F1, Integer(2):F2})
>>> V.support()
(1, 3)
```

symmetric_power(*n*)

Return the n-th graded symmetric power.

INPUT:

- `n` – integer; symmetric product of how many copies of `self`

OUTPUT:

The symmetric power as a multi-filtered vector space. See *symmetric_power()*.

EXAMPLES:

```
sage: F1 = FilteredVectorSpace(2, 1)
sage: F2 = FilteredVectorSpace(1, 3) + FilteredVectorSpace(1,0)
sage: V = MultiFilteredVectorSpace({'a':F1, 'b':F2})
sage: V.symmetric_power(2)
Filtrations
    a: QQ^3 >= QQ^3 >= QQ^3 >=  0   >=  0   >=  0   >=  0   >= 0
    b: QQ^3 >= QQ^2 >= QQ^2 >= QQ^2 >= QQ^1 >= QQ^1 >= QQ^1 >= 0
```

```
>>> from sage.all import *
>>> F1 = FilteredVectorSpace(Integer(2), Integer(1))
>>> F2 = FilteredVectorSpace(Integer(1), Integer(3)) +␣
↪FilteredVectorSpace(Integer(1),Integer(0))
>>> V = MultiFilteredVectorSpace({'a':F1, 'b':F2})
>>> V.symmetric_power(Integer(2))
Filtrations
    a: QQ^3 >= QQ^3 >= QQ^3 >=  0   >=  0   >=  0   >=  0   >= 0
    b: QQ^3 >= QQ^2 >= QQ^2 >= QQ^2 >= QQ^1 >= QQ^1 >= QQ^1 >= 0
```

tensor_product(*other*)

Return the graded tensor product.

INPUT:

- `other` – a multi-filtered vector space with the same *index_set()*

OUTPUT:

The tensor product of `self` and `other` as a multi-filtered vector space. See *tensor_product()*.

EXAMPLES:

```
sage: F1 = FilteredVectorSpace(2, 1)
sage: F2 = FilteredVectorSpace(1, 3) + FilteredVectorSpace(1,0)
sage: V = MultiFilteredVectorSpace({'a':F1, 'b':F2})
sage: G1 = FilteredVectorSpace(1, 1)
sage: G2 = FilteredVectorSpace(1, 3)
sage: W = MultiFilteredVectorSpace({'a':G1, 'b':G2})
sage: V.tensor_product(W)
Filtrations
    a: QQ^2 >=  0   >=  0   >=  0   >=  0   >= 0
    b: QQ^2 >= QQ^2 >= QQ^1 >= QQ^1 >= QQ^1 >= 0
sage: V * W   # syntactic sugar
Filtrations
    a: QQ^2 >=  0   >=  0   >=  0   >=  0   >= 0
    b: QQ^2 >= QQ^2 >= QQ^1 >= QQ^1 >= QQ^1 >= 0
```

```
>>> from sage.all import *
>>> F1 = FilteredVectorSpace(Integer(2), Integer(1))
>>> F2 = FilteredVectorSpace(Integer(1), Integer(3)) +␣
↪FilteredVectorSpace(Integer(1),Integer(0))
>>> V = MultiFilteredVectorSpace({'a':F1, 'b':F2})
>>> G1 = FilteredVectorSpace(Integer(1), Integer(1))
>>> G2 = FilteredVectorSpace(Integer(1), Integer(3))
>>> W = MultiFilteredVectorSpace({'a':G1, 'b':G2})
>>> V.tensor_product(W)
Filtrations
    a: QQ^2 >=  0   >=  0   >=  0   >=  0   >= 0
    b: QQ^2 >= QQ^2 >= QQ^1 >= QQ^1 >= QQ^1 >= 0
>>> V * W   # syntactic sugar
Filtrations
    a: QQ^2 >=  0   >=  0   >=  0   >=  0   >= 0
    b: QQ^2 >= QQ^2 >= QQ^1 >= QQ^1 >= QQ^1 >= 0
```

wedge(n)

Return the n-th graded exterior power.

INPUT:

- `n` – integer; exterior product of how many copies of `self`

OUTPUT:

The exterior power as a multi-filtered vector space. See *exterior_power()*.

EXAMPLES:

```
sage: F1 = FilteredVectorSpace(2, 1)
sage: F2 = FilteredVectorSpace(1, 3) + FilteredVectorSpace(1,0)
sage: V = MultiFilteredVectorSpace({'a':F1, 'b':F2})
sage: V.exterior_power(2)  # long time
Filtrations
    a: QQ^1 >=  0   >= 0
    b: QQ^1 >= QQ^1 >= 0
```

```
>>> from sage.all import *
>>> F1 = FilteredVectorSpace(Integer(2), Integer(1))
```

(continues on next page)

(continued from previous page)

```
>>> F2 = FilteredVectorSpace(Integer(1), Integer(3)) +␣
↪FilteredVectorSpace(Integer(1),Integer(0))
>>> V = MultiFilteredVectorSpace({'a':F1, 'b':F2})
>>> V.exterior_power(Integer(2))  # long time
Filtrations
    a: QQ^1 >=  0   >= 0
    b: QQ^1 >= QQ^1 >= 0
```

CHAPTER

SEVEN

MORPHISMS

7.1 Space of morphisms of vector spaces (linear transformations)

AUTHOR:

- Rob Beezer: (2011-06-29)

A *VectorSpaceHomspace* object represents the set of all possible homomorphisms from one vector space to another. These mappings are usually known as linear transformations.

For more information on the use of linear transformations, consult the documentation for vector space morphisms at *sage.modules.vector_space_morphism*. Also, this is an extremely thin veneer on free module homspaces (*sage.modules.free_module_homspace*) and free module morphisms (*sage.modules.free_module_morphism*) - objects which might also be useful, and places where much of the documentation resides.

EXAMPLES:

Creation and basic examination is simple.

```
sage: V = QQ^3
sage: W = QQ^2
sage: H = Hom(V, W)
sage: H
Set of Morphisms (Linear Transformations) from
Vector space of dimension 3 over Rational Field to
Vector space of dimension 2 over Rational Field
sage: H.domain()
Vector space of dimension 3 over Rational Field
sage: H.codomain()
Vector space of dimension 2 over Rational Field
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3)
>>> W = QQ**Integer(2)
>>> H = Hom(V, W)
>>> H
Set of Morphisms (Linear Transformations) from
Vector space of dimension 3 over Rational Field to
Vector space of dimension 2 over Rational Field
>>> H.domain()
Vector space of dimension 3 over Rational Field
>>> H.codomain()
Vector space of dimension 2 over Rational Field
```

Homspaces have a few useful properties. A basis is provided by a list of matrix representations, where these matrix representatives are relative to the bases of the domain and codomain.

```
sage: K = Hom(GF(3)^2, GF(3)^2)
sage: B = K.basis()
sage: for f in B:
....:     print(f)
....:     print("\n")
Vector space morphism represented by the matrix:
[1 0]
[0 0]
Domain: Vector space of dimension 2 over Finite Field of size 3
Codomain: Vector space of dimension 2 over Finite Field of size 3

Vector space morphism represented by the matrix:
[0 1]
[0 0]
Domain: Vector space of dimension 2 over Finite Field of size 3
Codomain: Vector space of dimension 2 over Finite Field of size 3

Vector space morphism represented by the matrix:
[0 0]
[1 0]
Domain: Vector space of dimension 2 over Finite Field of size 3
Codomain: Vector space of dimension 2 over Finite Field of size 3

Vector space morphism represented by the matrix:
[0 0]
[0 1]
Domain: Vector space of dimension 2 over Finite Field of size 3
Codomain: Vector space of dimension 2 over Finite Field of size 3
```

```
>>> from sage.all import *
>>> K = Hom(GF(Integer(3))**Integer(2), GF(Integer(3))**Integer(2))
>>> B = K.basis()
>>> for f in B:
...     print(f)
...     print("\n")
Vector space morphism represented by the matrix:
[1 0]
[0 0]
Domain: Vector space of dimension 2 over Finite Field of size 3
Codomain: Vector space of dimension 2 over Finite Field of size 3
<BLANKLINE>
Vector space morphism represented by the matrix:
[0 1]
[0 0]
Domain: Vector space of dimension 2 over Finite Field of size 3
Codomain: Vector space of dimension 2 over Finite Field of size 3
<BLANKLINE>
Vector space morphism represented by the matrix:
[0 0]
[1 0]
```

(continues on next page)

(continued from previous page)

```
Domain: Vector space of dimension 2 over Finite Field of size 3
Codomain: Vector space of dimension 2 over Finite Field of size 3
<BLANKLINE>
Vector space morphism represented by the matrix:
[0 0]
[0 1]
Domain: Vector space of dimension 2 over Finite Field of size 3
Codomain: Vector space of dimension 2 over Finite Field of size 3
<BLANKLINE>
```

The zero and identity mappings are properties of the space. The identity mapping will only be available if the domain and codomain allow for endomorphisms (equal vector spaces with equal bases).

```
sage: H = Hom(QQ^3, QQ^3)
sage: g = H.zero()
sage: g([1, 1/2, -3])
(0, 0, 0)
sage: f = H.identity()
sage: f([1, 1/2, -3])
(1, 1/2, -3)
```

```
>>> from sage.all import *
>>> H = Hom(QQ**Integer(3), QQ**Integer(3))
>>> g = H.zero()
>>> g([Integer(1), Integer(1)/Integer(2), -Integer(3)])
(0, 0, 0)
>>> f = H.identity()
>>> f([Integer(1), Integer(1)/Integer(2), -Integer(3)])
(1, 1/2, -3)
```

The homspace may be used with various representations of a morphism in the space to create the morphism. We demonstrate three ways to create the same linear transformation between two two-dimensional subspaces of `QQ^3`. The `V.n` notation is a shortcut to the generators of each vector space, better known as the basis elements. Note that the matrix representations are relative to the bases, which are purposely fixed when the subspaces are created ("user bases").

```
sage: U = QQ^3
sage: V = U.subspace_with_basis([U.0+U.1, U.1-U.2])
sage: W = U.subspace_with_basis([U.0, U.1+U.2])
sage: H = Hom(V, W)
```

```
>>> from sage.all import *
>>> U = QQ**Integer(3)
>>> V = U.subspace_with_basis([U.gen(0)+U.gen(1), U.gen(1)-U.gen(2)])
>>> W = U.subspace_with_basis([U.gen(0), U.gen(1)+U.gen(2)])
>>> H = Hom(V, W)
```

First, with a matrix. Note that the matrix representation acts by matrix multiplication with the vector on the left. The input to the linear transformation, `(3, 1, 2)`, is converted to the coordinate vector `(3, -2)`, then matrix multiplication yields the vector `(-3, -2)`, which represents the vector `(-3, -2, -2)` in the codomain.

```
sage: m = matrix(QQ, [[1, 2], [3, 4]])
sage: f1 = H(m)
```

(continues on next page)

(continued from previous page)

```
sage: f1
Vector space morphism represented by the matrix:
[1 2]
[3 4]
Domain: Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[ 1  1  0]
[ 0  1 -1]
Codomain: Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[1 0 0]
[0 1 1]
sage: f1([3,1,2])
(-3, -2, -2)
```

```
>>> from sage.all import *
>>> m = matrix(QQ, [[Integer(1), Integer(2)], [Integer(3), Integer(4)]])
>>> f1 = H(m)
>>> f1
Vector space morphism represented by the matrix:
[1 2]
[3 4]
Domain: Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[ 1  1  0]
[ 0  1 -1]
Codomain: Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[1 0 0]
[0 1 1]
>>> f1([Integer(3),Integer(1),Integer(2)])
(-3, -2, -2)
```

Second, with a list of images of the domain's basis elements.

```
sage: img = [1*(U.0) + 2*(U.1+U.2), 3*U.0 + 4*(U.1+U.2)]
sage: f2 = H(img)
sage: f2
Vector space morphism represented by the matrix:
[1 2]
[3 4]
Domain: Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[ 1  1  0]
[ 0  1 -1]
Codomain: Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[1 0 0]
[0 1 1]
sage: f2([3,1,2])
(-3, -2, -2)
```

```
>>> from sage.all import *
>>> img = [Integer(1)*(U.gen(0)) + Integer(2)*(U.gen(1)+U.gen(2)), Integer(3)*U.
↪gen(0) + Integer(4)*(U.gen(1)+U.gen(2))]
>>> f2 = H(img)
>>> f2
Vector space morphism represented by the matrix:
[1 2]
[3 4]
Domain: Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[ 1  1  0]
[ 0  1 -1]
Codomain: Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[1 0 0]
[0 1 1]
>>> f2([Integer(3),Integer(1),Integer(2)])
(-3, -2, -2)
```

Third, with a linear function taking the domain to the codomain.

```
sage: g = lambda x: vector(QQ, [-2*x[0]+3*x[1], -2*x[0]+4*x[1], -2*x[0]+4*x[1]])
sage: f3 = H(g)
sage: f3
Vector space morphism represented by the matrix:
[1 2]
[3 4]
Domain: Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[ 1  1  0]
[ 0  1 -1]
Codomain: Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[1 0 0]
[0 1 1]
sage: f3([3,1,2])
(-3, -2, -2)
```

```
>>> from sage.all import *
>>> g = lambda x: vector(QQ, [-Integer(2)*x[Integer(0)]+Integer(3)*x[Integer(1)], -
↪Integer(2)*x[Integer(0)]+Integer(4)*x[Integer(1)], -
↪Integer(2)*x[Integer(0)]+Integer(4)*x[Integer(1)]])
>>> f3 = H(g)
>>> f3
Vector space morphism represented by the matrix:
[1 2]
[3 4]
Domain: Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[ 1  1  0]
[ 0  1 -1]
Codomain: Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
```

(continues on next page)

(continued from previous page)

```
[1 0 0]
[0 1 1]
>>> f3([Integer(3),Integer(1),Integer(2)])
(-3, -2, -2)
```

The three linear transformations look the same, and are the same.

```
sage: f1 == f2
True
sage: f2 == f3
True
```

```
>>> from sage.all import *
>>> f1 == f2
True
>>> f2 == f3
True
```

class sage.modules.vector_space_homspace.**VectorSpaceHomspace**(*X*, *Y*, *category=None*, *check=True*, *base=None*)

Bases: *FreeModuleHomspace*

sage.modules.vector_space_homspace.**is_VectorSpaceHomspace**(*x*)

Return `True` if `x` is a vector space homspace.

INPUT:

- `x` – anything

EXAMPLES:

To be a vector space morphism, the domain and codomain must both be vector spaces, in other words, modules over fields. If either set is just a module, then the `Hom()` constructor will build a space of free module morphisms.

```
sage: H = Hom(QQ^3, QQ^2)
sage: type(H)
<class 'sage.modules.vector_space_homspace.VectorSpaceHomspace_with_category'>
sage: sage.modules.vector_space_homspace.is_VectorSpaceHomspace(H)
doctest:warning...
DeprecationWarning: the function is_VectorSpaceHomspace is deprecated;
use 'isinstance(..., VectorSpaceHomspace)' instead
See https://github.com/sagemath/sage/issues/37924 for details.
True

sage: K = Hom(QQ^3, ZZ^2)
sage: type(K)
<class 'sage.modules.free_module_homspace.FreeModuleHomspace_with_category'>
sage: sage.modules.vector_space_homspace.is_VectorSpaceHomspace(K)
False

sage: L = Hom(ZZ^3, QQ^2)
sage: type(L)
<class 'sage.modules.free_module_homspace.FreeModuleHomspace_with_category'>
sage: sage.modules.vector_space_homspace.is_VectorSpaceHomspace(L)
```

(continues on next page)

(continued from previous page)

```
False

sage: sage.modules.vector_space_homspace.is_VectorSpaceHomspace('junk')
False
```

```
>>> from sage.all import *
>>> H = Hom(QQ**Integer(3), QQ**Integer(2))
>>> type(H)
<class 'sage.modules.vector_space_homspace.VectorSpaceHomspace_with_category'>
>>> sage.modules.vector_space_homspace.is_VectorSpaceHomspace(H)
doctest:warning...
DeprecationWarning: the function is_VectorSpaceHomspace is deprecated;
use 'isinstance(..., VectorSpaceHomspace)' instead
See https://github.com/sagemath/sage/issues/37924 for details.
True

>>> K = Hom(QQ**Integer(3), ZZ**Integer(2))
>>> type(K)
<class 'sage.modules.free_module_homspace.FreeModuleHomspace_with_category'>
>>> sage.modules.vector_space_homspace.is_VectorSpaceHomspace(K)
False

>>> L = Hom(ZZ**Integer(3), QQ**Integer(2))
>>> type(L)
<class 'sage.modules.free_module_homspace.FreeModuleHomspace_with_category'>
>>> sage.modules.vector_space_homspace.is_VectorSpaceHomspace(L)
False

>>> sage.modules.vector_space_homspace.is_VectorSpaceHomspace('junk')
False
```

7.2 Morphisms of vector spaces (linear transformations)

AUTHOR:

- Rob Beezer: (2011-06-29)

A vector space morphism is a homomorphism between vector spaces, better known as a linear transformation. These are a specialization of Sage's free module homomorphisms. (A free module is like a vector space, but with scalars from a ring that may not be a field.) So references to free modules in the documentation or error messages should be understood as simply reflecting a more general situation.

7.2.1 Creation

The constructor *linear_transformation()* is designed to accept a variety of inputs that can define a linear transformation. See the documentation of the function for all the possibilities. Here we give two.

First a matrix representation. By default input matrices are understood to act on vectors placed to left of the matrix. Optionally, an input matrix can be described as acting on vectors placed to the right.

```
sage: A = matrix(QQ, [[-1, 2, 3], [4, 2, 0]])
sage: phi = linear_transformation(A)
```

(continues on next page)

(continued from previous page)

```
sage: phi
Vector space morphism represented by the matrix:
[-1  2  3]
[ 4  2  0]
Domain:   Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 3 over Rational Field
sage: phi([2, -3])
(-14, -2, 6)
```

```
>>> from sage.all import *
>>> A = matrix(QQ, [[-Integer(1), Integer(2), Integer(3)], [Integer(4), Integer(2),␣
↪Integer(0)]])
>>> phi = linear_transformation(A)
>>> phi
Vector space morphism represented by the matrix:
[-1  2  3]
[ 4  2  0]
Domain:   Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 3 over Rational Field
>>> phi([Integer(2), -Integer(3)])
(-14, -2, 6)
```

A symbolic function can be used to specify the "rule" for a linear transformation, along with explicit descriptions of the domain and codomain.

```
sage: # needs sage.symbolic
sage: F = Integers(13)
sage: D = F^3
sage: C = F^2
sage: x, y, z = var('x y z')
sage: f(x, y, z) = [2*x + 3*y + 5*z, x + z]
sage: rho = linear_transformation(D, C, f)
sage: f(1, 2, 3)
(23, 4)
sage: rho([1, 2, 3])
(10, 4)
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> F = Integers(Integer(13))
>>> D = F**Integer(3)
>>> C = F**Integer(2)
>>> x, y, z = var('x y z')
>>> __tmp__=var("x,y,z"); f = symbolic_expression([Integer(2)*x + Integer(3)*y +␣
↪Integer(5)*z, x + z]).function(x,y,z)
>>> rho = linear_transformation(D, C, f)
>>> f(Integer(1), Integer(2), Integer(3))
(23, 4)
>>> rho([Integer(1), Integer(2), Integer(3)])
(10, 4)
```

A "vector space homspace" is the set of all linear transformations between two vector spaces. Various input can be coerced

into a homspace to create a linear transformation. See *sage.modules.vector_space_homspace* for more.

```
sage: D = QQ^4
sage: C = QQ^2
sage: hom_space = Hom(D, C)
sage: images = [[1, 3], [2, -1], [4, 0], [3, 7]]
sage: zeta = hom_space(images)
sage: zeta
Vector space morphism represented by the matrix:
[ 1  3]
[ 2 -1]
[ 4  0]
[ 3  7]
Domain:   Vector space of dimension 4 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
```

```
>>> from sage.all import *
>>> D = QQ**Integer(4)
>>> C = QQ**Integer(2)
>>> hom_space = Hom(D, C)
>>> images = [[Integer(1), Integer(3)], [Integer(2), -Integer(1)], [Integer(4),␣
↪Integer(0)], [Integer(3), Integer(7)]]
>>> zeta = hom_space(images)
>>> zeta
Vector space morphism represented by the matrix:
[ 1  3]
[ 2 -1]
[ 4  0]
[ 3  7]
Domain:   Vector space of dimension 4 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
```

A homomorphism may also be created via a method on the domain.

```
sage: # needs sage.rings.number_field sage.symbolic
sage: F = QQ[sqrt(3)]
sage: a = F.gen(0)
sage: D = F^2
sage: C = F^2
sage: A = matrix(F, [[a, 1], [2*a, 2]])
sage: psi = D.hom(A, C)
sage: psi
Vector space morphism represented by the matrix:
[  sqrt3       1]
[2*sqrt3       2]
Domain:   Vector space of dimension 2 over Number Field in sqrt3
          with defining polynomial x^2 - 3 with sqrt3 = 1.732050807568878?
Codomain: Vector space of dimension 2 over Number Field in sqrt3
          with defining polynomial x^2 - 3 with sqrt3 = 1.732050807568878?
sage: psi([1, 4])
(9*sqrt3, 9)
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> # needs sage.rings.number_field sage.symbolic
>>> F = QQ[sqrt(Integer(3))]
>>> a = F.gen(Integer(0))
>>> D = F**Integer(2)
>>> C = F**Integer(2)
>>> A = matrix(F, [[a, Integer(1)], [Integer(2)*a, Integer(2)]])
>>> psi = D.hom(A, C)
>>> psi
Vector space morphism represented by the matrix:
[  sqrt3       1]
[2*sqrt3       2]
Domain:   Vector space of dimension 2 over Number Field in sqrt3
          with defining polynomial x^2 - 3 with sqrt3 = 1.732050807568878?
Codomain: Vector space of dimension 2 over Number Field in sqrt3
          with defining polynomial x^2 - 3 with sqrt3 = 1.732050807568878?
>>> psi([Integer(1), Integer(4)])
(9*sqrt3, 9)
```

7.2.2 Properties

Many natural properties of a linear transformation can be computed. Some of these are more general methods of objects in the classes *sage.modules.free_module_morphism.FreeModuleMorphism* and *sage.modules.matrix_morphism.MatrixMorphism*.

Values are computed in a natural way, an inverse image of an element can be computed with the `lift()` method, when the inverse image actually exists.

```
sage: A = matrix(QQ, [[1,2], [2,4], [3,6]])
sage: phi = linear_transformation(A)
sage: phi([1,2,0])
(5, 10)
sage: phi.lift([10, 20])
(10, 0, 0)
sage: phi.lift([100, 100])
Traceback (most recent call last):
...
ValueError: element is not in the image
```

```
>>> from sage.all import *
>>> A = matrix(QQ, [[Integer(1),Integer(2)], [Integer(2),Integer(4)], [Integer(3),
↪Integer(6)]])
>>> phi = linear_transformation(A)
>>> phi([Integer(1),Integer(2),Integer(0)])
(5, 10)
>>> phi.lift([Integer(10), Integer(20)])
(10, 0, 0)
>>> phi.lift([Integer(100), Integer(100)])
Traceback (most recent call last):
...
ValueError: element is not in the image
```

Images and pre-images can be computed as vector spaces.

```
sage: A = matrix(QQ, [[1,2], [2,4], [3,6]])
sage: phi = linear_transformation(A)
sage: phi.image()
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 2]

sage: phi.inverse_image( (QQ^2).span([[1,2]]) )
Vector space of degree 3 and dimension 3 over Rational Field
Basis matrix:
[1 0 0]
[0 1 0]
[0 0 1]

sage: phi.inverse_image( (QQ^2).span([[1,1]]) )
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[   1    0 -1/3]
[   0    1 -2/3]
```

```
>>> from sage.all import *
>>> A = matrix(QQ, [[Integer(1),Integer(2)], [Integer(2),Integer(4)], [Integer(3),
↪Integer(6)]])
>>> phi = linear_transformation(A)
>>> phi.image()
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 2]

>>> phi.inverse_image( (QQ**Integer(2)).span([[Integer(1),Integer(2)]]) )
Vector space of degree 3 and dimension 3 over Rational Field
Basis matrix:
[1 0 0]
[0 1 0]
[0 0 1]

>>> phi.inverse_image( (QQ**Integer(2)).span([[Integer(1),Integer(1)]]) )
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[   1    0 -1/3]
[   0    1 -2/3]
```

Injectivity and surjectivity can be checked.

```
sage: A = matrix(QQ, [[1,2], [2,4], [3,6]])
sage: phi = linear_transformation(A)
sage: phi.is_injective()
False
sage: phi.is_surjective()
False
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> A = matrix(QQ, [[Integer(1),Integer(2)], [Integer(2),Integer(4)], [Integer(3),
↪Integer(6)]])
>>> phi = linear_transformation(A)
>>> phi.is_injective()
False
>>> phi.is_surjective()
False
```

7.2.3 Restrictions and representations

It is possible to restrict the domain and codomain of a linear transformation to make a new linear transformation. We will use those commands to replace the domain and codomain by equal vector spaces, but with alternate bases. The point here is that the matrix representation used to represent linear transformations are relative to the bases of both the domain and codomain.

```
sage: A = graphs.PetersenGraph().adjacency_matrix()                    #␣
↪needs sage.graphs
sage: V = QQ^10
sage: phi = linear_transformation(V, V, A)                             #␣
↪needs sage.graphs
sage: phi                                                              #␣
↪needs sage.graphs
Vector space morphism represented by the matrix:
[0 1 0 0 1 1 0 0 0 0]
[1 0 1 0 0 0 1 0 0 0]
[0 1 0 1 0 0 0 1 0 0]
[0 0 1 0 1 0 0 0 1 0]
[1 0 0 1 0 0 0 0 0 1]
[1 0 0 0 0 0 0 1 1 0]
[0 1 0 0 0 0 0 0 1 1]
[0 0 1 0 0 1 0 0 0 1]
[0 0 0 1 0 1 1 0 0 0]
[0 0 0 0 1 0 1 1 0 0]
Domain:   Vector space of dimension 10 over Rational Field
Codomain: Vector space of dimension 10 over Rational Field

sage: # needs sage.graphs
sage: B1 = [V.gen(i) + V.gen(i+1) for i in range(9)] + [V.gen(9)]
sage: B2 = [V.gen(0)] + [-V.gen(i-1) + V.gen(i) for i in range(1,10)]
sage: D = V.subspace_with_basis(B1)
sage: C = V.subspace_with_basis(B2)
sage: rho = phi.restrict_codomain(C)
sage: zeta = rho.restrict_domain(D)
sage: zeta
Vector space morphism represented by the matrix:
[6 5 4 3 3 2 1 0 0 0]
[6 5 4 3 2 2 2 1 0 0]
[6 6 5 4 3 2 2 2 1 0]
[6 5 5 4 3 2 2 2 2 1]
[6 4 4 4 3 3 3 3 2 1]
[6 5 4 4 4 4 4 4 3 1]
[6 6 5 4 4 4 3 3 3 2]
```

(continues on next page)

(continued from previous page)

```
[6 6 6 5 4 4 2 1 1 1]
[6 6 6 6 5 4 3 1 0 0]
[3 3 3 3 3 2 2 1 0 0]
Domain:   Vector space of degree 10 and dimension 10 over Rational Field
          User basis matrix:
          [1 1 0 0 0 0 0 0 0 0]
          [0 1 1 0 0 0 0 0 0 0]
          [0 0 1 1 0 0 0 0 0 0]
          [0 0 0 1 1 0 0 0 0 0]
          [0 0 0 0 1 1 0 0 0 0]
          [0 0 0 0 0 1 1 0 0 0]
          [0 0 0 0 0 0 1 1 0 0]
          [0 0 0 0 0 0 0 1 1 0]
          [0 0 0 0 0 0 0 0 1 1]
          [0 0 0 0 0 0 0 0 0 1]
Codomain: Vector space of degree 10 and dimension 10 over Rational Field
          User basis matrix:
          [ 1  0  0  0  0  0  0  0  0  0]
          [-1  1  0  0  0  0  0  0  0  0]
          [ 0 -1  1  0  0  0  0  0  0  0]
          [ 0  0 -1  1  0  0  0  0  0  0]
          [ 0  0  0 -1  1  0  0  0  0  0]
          [ 0  0  0  0 -1  1  0  0  0  0]
          [ 0  0  0  0  0 -1  1  0  0  0]
          [ 0  0  0  0  0  0 -1  1  0  0]
          [ 0  0  0  0  0  0  0 -1  1  0]
          [ 0  0  0  0  0  0  0  0 -1  1]
```

```
>>> from sage.all import *
>>> A = graphs.PetersenGraph().adjacency_matrix()                              #␣
↪needs sage.graphs
>>> V = QQ**Integer(10)
>>> phi = linear_transformation(V, V, A)                                       #␣
↪needs sage.graphs
>>> phi                                                                        #␣
↪needs sage.graphs
Vector space morphism represented by the matrix:
[0 1 0 0 1 1 0 0 0 0]
[1 0 1 0 0 0 1 0 0 0]
[0 1 0 1 0 0 0 1 0 0]
[0 0 1 0 1 0 0 0 1 0]
[1 0 0 1 0 0 0 0 0 1]
[1 0 0 0 0 0 0 1 1 0]
[0 1 0 0 0 0 0 0 1 1]
[0 0 1 0 0 1 0 0 0 1]
[0 0 0 1 0 1 1 0 0 0]
[0 0 0 0 1 0 1 1 0 0]
Domain:   Vector space of dimension 10 over Rational Field
Codomain: Vector space of dimension 10 over Rational Field

>>> # needs sage.graphs
>>> B1 = [V.gen(i) + V.gen(i+Integer(1)) for i in range(Integer(9))] + [V.
```

(continues on next page)

(continued from previous page)

```
↪gen(Integer(9))]
>>> B2 = [V.gen(Integer(0))] + [-V.gen(i-Integer(1)) + V.gen(i) for i in␣
↪range(Integer(1),Integer(10))]
>>> D = V.subspace_with_basis(B1)
>>> C = V.subspace_with_basis(B2)
>>> rho = phi.restrict_codomain(C)
>>> zeta = rho.restrict_domain(D)
>>> zeta
Vector space morphism represented by the matrix:
[6 5 4 3 3 2 1 0 0 0]
[6 5 4 3 2 2 2 1 0 0]
[6 6 5 4 3 2 2 2 1 0]
[6 5 5 4 3 2 2 2 2 1]
[6 4 4 4 3 3 3 3 2 1]
[6 5 4 4 4 4 4 4 3 1]
[6 6 5 4 4 4 3 3 3 2]
[6 6 6 5 4 4 2 1 1 1]
[6 6 6 6 5 4 3 1 0 0]
[3 3 3 3 3 2 2 1 0 0]
Domain:   Vector space of degree 10 and dimension 10 over Rational Field
          User basis matrix:
          [1 1 0 0 0 0 0 0 0 0]
          [0 1 1 0 0 0 0 0 0 0]
          [0 0 1 1 0 0 0 0 0 0]
          [0 0 0 1 1 0 0 0 0 0]
          [0 0 0 0 1 1 0 0 0 0]
          [0 0 0 0 0 1 1 0 0 0]
          [0 0 0 0 0 0 1 1 0 0]
          [0 0 0 0 0 0 0 1 1 0]
          [0 0 0 0 0 0 0 0 1 1]
          [0 0 0 0 0 0 0 0 0 1]
Codomain: Vector space of degree 10 and dimension 10 over Rational Field
          User basis matrix:
          [ 1  0  0  0  0  0  0  0  0  0]
          [-1  1  0  0  0  0  0  0  0  0]
          [ 0 -1  1  0  0  0  0  0  0  0]
          [ 0  0 -1  1  0  0  0  0  0  0]
          [ 0  0  0 -1  1  0  0  0  0  0]
          [ 0  0  0  0 -1  1  0  0  0  0]
          [ 0  0  0  0  0 -1  1  0  0  0]
          [ 0  0  0  0  0  0 -1  1  0  0]
          [ 0  0  0  0  0  0  0 -1  1  0]
          [ 0  0  0  0  0  0  0  0 -1  1]
```

An endomorphism is a linear transformation with an equal domain and codomain, and here each needs to have the same basis. We are using a matrix that has well-behaved eigenvalues, as part of showing that these do not change as the representation changes.

```
sage: # needs sage.graphs
sage: A = graphs.PetersenGraph().adjacency_matrix()
sage: V = QQ^10
sage: phi = linear_transformation(V, V, A)
```

(continues on next page)

(continued from previous page)

```
sage: phi.eigenvalues()                                                    #␣
↪needs sage.rings.number_field
[3, -2, -2, -2, -2, 1, 1, 1, 1, 1]
sage: B1 = [V.gen(i) + V.gen(i+1) for i in range(9)] + [V.gen(9)]
sage: C = V.subspace_with_basis(B1)
sage: zeta = phi.restrict(C)
sage: zeta
Vector space morphism represented by the matrix:
[ 1  0  1 -1  2 -1  2 -2  2 -2]
[ 1  0  1  0  0  0  1  0  0  0]
[ 0  1  0  1  0  0  0  1  0  0]
[ 1 -1  2 -1  2 -2  2 -2  3 -2]
[ 2 -2  2 -1  1 -1  1  0  1  0]
[ 1  0  0  0  0  0  0  1  1  0]
[ 0  1  0  0  0  1 -1  1  0  2]
[ 0  0  1  0  0  2 -1  1 -1  2]
[ 0  0  0  1  0  1  1  0  0  0]
[ 0  0  0  0  1 -1  2 -1  1 -1]
Domain:   Vector space of degree 10 and dimension 10 over Rational Field
          User basis matrix:
          [1 1 0 0 0 0 0 0 0 0]
          [0 1 1 0 0 0 0 0 0 0]
          [0 0 1 1 0 0 0 0 0 0]
          [0 0 0 1 1 0 0 0 0 0]
          [0 0 0 0 1 1 0 0 0 0]
          [0 0 0 0 0 1 1 0 0 0]
          [0 0 0 0 0 0 1 1 0 0]
          [0 0 0 0 0 0 0 1 1 0]
          [0 0 0 0 0 0 0 0 1 1]
          [0 0 0 0 0 0 0 0 0 1]
Codomain: Vector space of degree 10 and dimension 10 over Rational Field
          User basis matrix:
          [1 1 0 0 0 0 0 0 0 0]
          [0 1 1 0 0 0 0 0 0 0]
          [0 0 1 1 0 0 0 0 0 0]
          [0 0 0 1 1 0 0 0 0 0]
          [0 0 0 0 1 1 0 0 0 0]
          [0 0 0 0 0 1 1 0 0 0]
          [0 0 0 0 0 0 1 1 0 0]
          [0 0 0 0 0 0 0 1 1 0]
          [0 0 0 0 0 0 0 0 1 1]
          [0 0 0 0 0 0 0 0 0 1]
sage: zeta.eigenvalues()                                                   #␣
↪needs sage.rings.number_field
[3, -2, -2, -2, -2, 1, 1, 1, 1, 1]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> A = graphs.PetersenGraph().adjacency_matrix()
>>> V = QQ**Integer(10)
>>> phi = linear_transformation(V, V, A)
>>> phi.eigenvalues()                                                      #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.rings.number_field
[3, -2, -2, -2, -2, 1, 1, 1, 1, 1]
>>> B1 = [V.gen(i) + V.gen(i+Integer(1)) for i in range(Integer(9))] + [V.
↪gen(Integer(9))]
>>> C = V.subspace_with_basis(B1)
>>> zeta = phi.restrict(C)
>>> zeta
Vector space morphism represented by the matrix:
[ 1  0  1 -1  2 -1  2 -2  2 -2]
[ 1  0  1  0  0  0  1  0  0  0]
[ 0  1  0  1  0  0  0  1  0  0]
[ 1 -1  2 -1  2 -2  2 -2  3 -2]
[ 2 -2  2 -1  1 -1  1  0  1  0]
[ 1  0  0  0  0  0  0  1  1  0]
[ 0  1  0  0  0  1 -1  1  0  2]
[ 0  0  1  0  0  2 -1  1 -1  2]
[ 0  0  0  1  0  1  1  0  0  0]
[ 0  0  0  0  1 -1  2 -1  1 -1]
Domain:   Vector space of degree 10 and dimension 10 over Rational Field
          User basis matrix:
          [1 1 0 0 0 0 0 0 0 0]
          [0 1 1 0 0 0 0 0 0 0]
          [0 0 1 1 0 0 0 0 0 0]
          [0 0 0 1 1 0 0 0 0 0]
          [0 0 0 0 1 1 0 0 0 0]
          [0 0 0 0 0 1 1 0 0 0]
          [0 0 0 0 0 0 1 1 0 0]
          [0 0 0 0 0 0 0 1 1 0]
          [0 0 0 0 0 0 0 0 1 1]
          [0 0 0 0 0 0 0 0 0 1]
Codomain: Vector space of degree 10 and dimension 10 over Rational Field
          User basis matrix:
          [1 1 0 0 0 0 0 0 0 0]
          [0 1 1 0 0 0 0 0 0 0]
          [0 0 1 1 0 0 0 0 0 0]
          [0 0 0 1 1 0 0 0 0 0]
          [0 0 0 0 1 1 0 0 0 0]
          [0 0 0 0 0 1 1 0 0 0]
          [0 0 0 0 0 0 1 1 0 0]
          [0 0 0 0 0 0 0 1 1 0]
          [0 0 0 0 0 0 0 0 1 1]
          [0 0 0 0 0 0 0 0 0 1]
>>> zeta.eigenvalues()                                                    #␣
↪needs sage.rings.number_field
[3, -2, -2, -2, -2, 1, 1, 1, 1, 1]
```

7.2.4 Equality

Equality of linear transformations is a bit nuanced. The equality operator == tests if two linear transformations have equal matrix representations, while we determine if two linear transformations are the same function with the `.is_equal_function()` method. Notice in this example that the function never changes, just the representations.

```
sage: f = lambda x: vector(QQ, [x[1], x[0]+x[1], x[0]])
sage: H = Hom(QQ^2, QQ^3)
sage: phi = H(f)

sage: rho = linear_transformation(QQ^2, QQ^3, matrix(QQ,2, 3, [[0,1,1], [1,1,0]]))

sage: phi == rho
True

sage: U = (QQ^2).subspace_with_basis([[1, 2], [-3, 1]])
sage: V = (QQ^3).subspace_with_basis([[0, 1, 0], [2, 3, 1], [-1, 1, 6]])
sage: K = Hom(U, V)
sage: zeta = K(f)

sage: zeta == phi
False
sage: zeta.is_equal_function(phi)
True
sage: zeta.is_equal_function(rho)
True
```

```
>>> from sage.all import *
>>> f = lambda x: vector(QQ, [x[Integer(1)], x[Integer(0)]+x[Integer(1)],␣
↪x[Integer(0)]])
>>> H = Hom(QQ**Integer(2), QQ**Integer(3))
>>> phi = H(f)

>>> rho = linear_transformation(QQ**Integer(2), QQ**Integer(3), matrix(QQ,Integer(2),␣
↪Integer(3), [[Integer(0),Integer(1),Integer(1)], [Integer(1),Integer(1),
↪Integer(0)]]))

>>> phi == rho
True

>>> U = (QQ**Integer(2)).subspace_with_basis([[Integer(1), Integer(2)], [-Integer(3),␣
↪Integer(1)]])
>>> V = (QQ**Integer(3)).subspace_with_basis([[Integer(0), Integer(1), Integer(0)],␣
↪[Integer(2), Integer(3), Integer(1)], [-Integer(1), Integer(1), Integer(6)]])
>>> K = Hom(U, V)
>>> zeta = K(f)

>>> zeta == phi
False
>>> zeta.is_equal_function(phi)
True
>>> zeta.is_equal_function(rho)
True
```

class sage.modules.vector_space_morphism.**VectorSpaceMorphism**(*homspace*, *A*, *side='left'*)

Bases: *FreeModuleMorphism*

Create a linear transformation, a morphism between vector spaces.

INPUT:

- homspace – a homspace (of vector spaces) to serve as a parent for the linear transformation and a home for the domain and codomain of the morphism
- A – a matrix representing the linear transformation, which will act on vectors placed to the left of the matrix

EXAMPLES:

Nominally, we require a homspace to hold the domain and codomain and a matrix representation of the morphism (linear transformation).

```
sage: from sage.modules.vector_space_homspace import VectorSpaceHomspace
sage: from sage.modules.vector_space_morphism import VectorSpaceMorphism
sage: H = VectorSpaceHomspace(QQ^3, QQ^2)
sage: A = matrix(QQ, 3, 2, range(6))
sage: zeta = VectorSpaceMorphism(H, A)
sage: zeta
Vector space morphism represented by the matrix:
[0 1]
[2 3]
[4 5]
Domain:   Vector space of dimension 3 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
```

```
>>> from sage.all import *
>>> from sage.modules.vector_space_homspace import VectorSpaceHomspace
>>> from sage.modules.vector_space_morphism import VectorSpaceMorphism
>>> H = VectorSpaceHomspace(QQ**Integer(3), QQ**Integer(2))
>>> A = matrix(QQ, Integer(3), Integer(2), range(Integer(6)))
>>> zeta = VectorSpaceMorphism(H, A)
>>> zeta
Vector space morphism represented by the matrix:
[0 1]
[2 3]
[4 5]
Domain:   Vector space of dimension 3 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
```

See the constructor, *sage.modules.vector_space_morphism.linear_transformation()* for another way to create linear transformations.

The .hom() method of a vector space will create a vector space morphism.

```
sage: V = QQ^3; W = V.subspace_with_basis([[1,2,3], [-1,2,5/3], [0,1,-1]])
sage: phi = V.hom(matrix(QQ, 3, range(9)), codomain=W) # indirect doctest
sage: type(phi)
<class 'sage.modules.vector_space_morphism.VectorSpaceMorphism'>
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3); W = V.subspace_with_basis([[Integer(1),Integer(2),
↪Integer(3)], [-Integer(1),Integer(2),Integer(5)/Integer(3)], [Integer(0),
↪Integer(1),-Integer(1)]])
>>> phi = V.hom(matrix(QQ, Integer(3), range(Integer(9))), codomain=W) # indirect␣
↪doctest
>>> type(phi)
<class 'sage.modules.vector_space_morphism.VectorSpaceMorphism'>
```

A matrix may be coerced into a vector space homspace to create a vector space morphism.

```
sage: from sage.modules.vector_space_homspace import VectorSpaceHomspace
sage: H = VectorSpaceHomspace(QQ^3, QQ^2)
sage: A = matrix(QQ, 3, 2, range(6))
sage: rho = H(A)  # indirect doctest
sage: type(rho)
<class 'sage.modules.vector_space_morphism.VectorSpaceMorphism'>
```

```
>>> from sage.all import *
>>> from sage.modules.vector_space_homspace import VectorSpaceHomspace
>>> H = VectorSpaceHomspace(QQ**Integer(3), QQ**Integer(2))
>>> A = matrix(QQ, Integer(3), Integer(2), range(Integer(6)))
>>> rho = H(A)  # indirect doctest
>>> type(rho)
<class 'sage.modules.vector_space_morphism.VectorSpaceMorphism'>
```

is_invertible()

Determines if the vector space morphism has an inverse.

OUTPUT:

`True` if the vector space morphism is invertible, otherwise `False`.

EXAMPLES:

If the dimension of the domain does not match the dimension of the codomain, then the morphism cannot be invertible.

```
sage: V = QQ^3
sage: U = V.subspace_with_basis([V.0 + V.1, 2*V.1 + 3*V.2])
sage: phi = V.hom([U.0, U.0 + U.1, U.0 - U.1], U)
sage: phi.is_invertible()
False
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3)
>>> U = V.subspace_with_basis([V.gen(0) + V.gen(1), Integer(2)*V.gen(1) +␣
↪Integer(3)*V.gen(2)])
>>> phi = V.hom([U.gen(0), U.gen(0) + U.gen(1), U.gen(0) - U.gen(1)], U)
>>> phi.is_invertible()
False
```

An invertible linear transformation.

```
sage: A = matrix(QQ, 3, [[-3, 5, -5], [4, -7, 7], [6, -8, 10]])
sage: A.determinant()
2
sage: H = Hom(QQ^3, QQ^3)
sage: rho = H(A)
sage: rho.is_invertible()
True
```

```
>>> from sage.all import *
>>> A = matrix(QQ, Integer(3), [[-Integer(3), Integer(5), -Integer(5)],␣
```

(continues on next page)

(continued from previous page)

```
↪[Integer(4), -Integer(7), Integer(7)], [Integer(6), -Integer(8),␣
↪Integer(10)]])
>>> A.determinant()
2
>>> H = Hom(QQ**Integer(3), QQ**Integer(3))
>>> rho = H(A)
>>> rho.is_invertible()
True
```

A non-invertible linear transformation, an endomorphism of a vector space over a finite field.

```
sage: # needs sage.rings.finite_rings
sage: F.<a> = GF(11^2)
sage: A = matrix(F, [[6*a + 3,   8*a +  2, 10*a + 3],
....:                [2*a + 7,   4*a +  3,  2*a + 3],
....:                [9*a + 2,  10*a + 10,  3*a + 3]])
sage: A.nullity()
1
sage: E = End(F^3)
sage: zeta = E(A)
sage: zeta.is_invertible()
False
```

```
>>> from sage.all import *
>>> # needs sage.rings.finite_rings
>>> F = GF(Integer(11)**Integer(2), names=('a',)); (a,) = F._first_ngens(1)
>>> A = matrix(F, [[Integer(6)*a + Integer(3),   Integer(8)*a +  Integer(2),␣
↪Integer(10)*a + Integer(3)],
...                [Integer(2)*a + Integer(7),   Integer(4)*a +  Integer(3), ␣
↪Integer(2)*a + Integer(3)],
...                [Integer(9)*a + Integer(2),  Integer(10)*a + Integer(10), ␣
↪Integer(3)*a + Integer(3)]])
>>> A.nullity()
1
>>> E = End(F**Integer(3))
>>> zeta = E(A)
>>> zeta.is_invertible()
False
```

sage.modules.vector_space_morphism.**is_VectorSpaceMorphism**(*x*)

Return `True` if `x` is a vector space morphism (a linear transformation).

This function is deprecated.

INPUT:

- `x` – anything

OUTPUT:

`True` only if `x` is an instance of a vector space morphism, which are also known as linear transformations.

EXAMPLES:

```
sage: V = QQ^2; f = V.hom([V.1,-2*V.0])
sage: sage.modules.vector_space_morphism.is_VectorSpaceMorphism(f)
doctest:warning...
DeprecationWarning: is_VectorSpaceMorphism is deprecated;
use isinstance(..., VectorSpaceMorphism) or categories instead
See https://github.com/sagemath/sage/issues/37731 for details.
True
sage: sage.modules.vector_space_morphism.is_VectorSpaceMorphism('junk')
False
```

```
>>> from sage.all import *
>>> V = QQ**Integer(2); f = V.hom([V.gen(1),-Integer(2)*V.gen(0)])
>>> sage.modules.vector_space_morphism.is_VectorSpaceMorphism(f)
doctest:warning...
DeprecationWarning: is_VectorSpaceMorphism is deprecated;
use isinstance(..., VectorSpaceMorphism) or categories instead
See https://github.com/sagemath/sage/issues/37731 for details.
True
>>> sage.modules.vector_space_morphism.is_VectorSpaceMorphism('junk')
False
```

sage.modules.vector_space_morphism.**linear_transformation**(*arg0*, *arg1=None*, *arg2=None*, *side='left'*)

Create a linear transformation from a variety of possible inputs.

FORMATS:

In the following, `D` and `C` are vector spaces over the same field that are the domain and codomain (respectively) of the linear transformation.

`side` is a keyword that is either `'left'` or `'right'`. When a matrix is used to specify a linear transformation, as in the first two call formats below, you may specify if the function is given by matrix multiplication with the vector on the left, or the vector on the right. The default is 'left'. The matrix representation may be obtained as either version, no matter how it is created.

- `linear_transformation(A, side='left')`

  Where `A` is a matrix. The domain and codomain are inferred from the dimension of the matrix and the base ring of the matrix. The base ring must be a field, or have its fraction field implemented in Sage.

- `linear_transformation(D, C, A, side='left')`

  `A` is a matrix that behaves as above. However, now the domain and codomain are given explicitly. The matrix is checked for compatibility with the domain and codomain. Additionally, the domain and codomain may be supplied with alternate ("user") bases and the matrix is interpreted as being a representation relative to those bases.

- `linear_transformation(D, C, f)`

  `f` is any function that can be applied to the basis elements of the domain and that produces elements of the codomain. The linear transformation returned is the unique linear transformation that extends this mapping on the basis elements. `f` may come from a function defined by a Python `def` statement, or may be defined as a `lambda` function.

  Alternatively, `f` may be specified by a callable symbolic function, see the examples below for a demonstration.

- `linear_transformation(D, C, images)`

`images` is a list, or tuple, of codomain elements, equal in number to the size of the basis of the domain. Each basis element of the domain is mapped to the corresponding element of the `images` list, and the linear transformation returned is the unique linear transformation that extends this mapping.

OUTPUT:

A linear transformation described by the input. This is a "vector space morphism", an object of the class *sage.modules.vector_space_morphism*.

EXAMPLES:

We can define a linear transformation with just a matrix, understood to act on a vector placed on one side or the other. The field for the vector spaces used as domain and codomain is obtained from the base ring of the matrix, possibly promoting to a fraction field.

```
sage: A = matrix(ZZ, [[1, -1, 4], [2, 0, 5]])
sage: phi = linear_transformation(A)
sage: phi
Vector space morphism represented by the matrix:
[ 1 -1  4]
[ 2  0  5]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 3 over Rational Field
sage: phi([1/2, 5])
(21/2, -1/2, 27)

sage: B = matrix(Integers(7), [[1, 2, 1], [3, 5, 6]])
sage: rho = linear_transformation(B, side='right')
sage: rho
Vector space morphism represented by the matrix:
[1 3]
[2 5]
[1 6]
Domain: Vector space of dimension 3 over Ring of integers modulo 7
Codomain: Vector space of dimension 2 over Ring of integers modulo 7
sage: rho([2, 4, 6])
(2, 6)
```

```
>>> from sage.all import *
>>> A = matrix(ZZ, [[Integer(1), -Integer(1), Integer(4)], [Integer(2),␣
↪Integer(0), Integer(5)]])
>>> phi = linear_transformation(A)
>>> phi
Vector space morphism represented by the matrix:
[ 1 -1  4]
[ 2  0  5]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 3 over Rational Field
>>> phi([Integer(1)/Integer(2), Integer(5)])
(21/2, -1/2, 27)

>>> B = matrix(Integers(Integer(7)), [[Integer(1), Integer(2), Integer(1)],␣
↪[Integer(3), Integer(5), Integer(6)]])
>>> rho = linear_transformation(B, side='right')
>>> rho
```

(continues on next page)

(continued from previous page)

```
Vector space morphism represented by the matrix:
[1 3]
[2 5]
[1 6]
Domain: Vector space of dimension 3 over Ring of integers modulo 7
Codomain: Vector space of dimension 2 over Ring of integers modulo 7
>>> rho([Integer(2), Integer(4), Integer(6)])
(2, 6)
```

We can define a linear transformation with a matrix, while explicitly giving the domain and codomain. Matrix entries will be coerced into the common field of scalars for the vector spaces.

```
sage: D = QQ^3
sage: C = QQ^2
sage: A = matrix([[1, 7], [2, -1], [0, 5]])
sage: A.parent()
Full MatrixSpace of 3 by 2 dense matrices over Integer Ring
sage: zeta = linear_transformation(D, C, A)
sage: zeta.matrix().parent()
Full MatrixSpace of 3 by 2 dense matrices over Rational Field
sage: zeta
Vector space morphism represented by the matrix:
[ 1  7]
[ 2 -1]
[ 0  5]
Domain: Vector space of dimension 3 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
```

```
>>> from sage.all import *
>>> D = QQ**Integer(3)
>>> C = QQ**Integer(2)
>>> A = matrix([[Integer(1), Integer(7)], [Integer(2), -Integer(1)], [Integer(0),␣
↪Integer(5)]])
>>> A.parent()
Full MatrixSpace of 3 by 2 dense matrices over Integer Ring
>>> zeta = linear_transformation(D, C, A)
>>> zeta.matrix().parent()
Full MatrixSpace of 3 by 2 dense matrices over Rational Field
>>> zeta
Vector space morphism represented by the matrix:
[ 1  7]
[ 2 -1]
[ 0  5]
Domain: Vector space of dimension 3 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field
```

Matrix representations are relative to the bases for the domain and codomain.

```
sage: u = vector(QQ, [1, -1])
sage: v = vector(QQ, [2, 3])
sage: D = (QQ^2).subspace_with_basis([u, v])
sage: x = vector(QQ, [2, 1])
```

(continues on next page)

(continued from previous page)

```
sage: y = vector(QQ, [-1, 4])
sage: C = (QQ^2).subspace_with_basis([x, y])
sage: A = matrix(QQ, [[2, 5], [3, 7]])
sage: psi = linear_transformation(D, C, A)
sage: psi
Vector space morphism represented by the matrix:
[2 5]
[3 7]
Domain: Vector space of degree 2 and dimension 2 over Rational Field
User basis matrix:
[ 1 -1]
[ 2  3]
Codomain: Vector space of degree 2 and dimension 2 over Rational Field
User basis matrix:
[ 2  1]
[-1  4]
sage: psi(u) == 2*x + 5*y
True
sage: psi(v) == 3*x + 7*y
True
```

```
>>> from sage.all import *
>>> u = vector(QQ, [Integer(1), -Integer(1)])
>>> v = vector(QQ, [Integer(2), Integer(3)])
>>> D = (QQ**Integer(2)).subspace_with_basis([u, v])
>>> x = vector(QQ, [Integer(2), Integer(1)])
>>> y = vector(QQ, [-Integer(1), Integer(4)])
>>> C = (QQ**Integer(2)).subspace_with_basis([x, y])
>>> A = matrix(QQ, [[Integer(2), Integer(5)], [Integer(3), Integer(7)]])
>>> psi = linear_transformation(D, C, A)
>>> psi
Vector space morphism represented by the matrix:
[2 5]
[3 7]
Domain: Vector space of degree 2 and dimension 2 over Rational Field
User basis matrix:
[ 1 -1]
[ 2  3]
Codomain: Vector space of degree 2 and dimension 2 over Rational Field
User basis matrix:
[ 2  1]
[-1  4]
>>> psi(u) == Integer(2)*x + Integer(5)*y
True
>>> psi(v) == Integer(3)*x + Integer(7)*y
True
```

Functions that act on the domain may be used to compute images of the domain's basis elements, and this mapping can be extended to a unique linear transformation. The function may be a Python function (via `def` or `lambda`) or a Sage symbolic function.

```
sage: def g(x):
```

(continues on next page)

(continued from previous page)

```
....:     return vector(QQ, [2*x[0]+x[2], 5*x[1]])
sage: phi = linear_transformation(QQ^3, QQ^2, g)
sage: phi
Vector space morphism represented by the matrix:
[2 0]
[0 5]
[1 0]
Domain: Vector space of dimension 3 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field

sage: f = lambda x: vector(QQ, [2*x[0]+x[2], 5*x[1]])
sage: rho = linear_transformation(QQ^3, QQ^2, f)
sage: rho
Vector space morphism represented by the matrix:
[2 0]
[0 5]
[1 0]
Domain: Vector space of dimension 3 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field

sage: # needs sage.symbolic
sage: x, y, z = var('x y z')
sage: h(x, y, z) = [2*x + z, 5*y]
sage: zeta = linear_transformation(QQ^3, QQ^2, h)
sage: zeta
Vector space morphism represented by the matrix:
[2 0]
[0 5]
[1 0]
Domain:   Vector space of dimension 3 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field

sage: phi == rho
True
sage: rho == zeta                                                          #␣
↪needs sage.symbolic
True
```

```
>>> from sage.all import *
>>> def g(x):
...     return vector(QQ, [Integer(2)*x[Integer(0)]+x[Integer(2)],␣
↪Integer(5)*x[Integer(1)]])
>>> phi = linear_transformation(QQ**Integer(3), QQ**Integer(2), g)
>>> phi
Vector space morphism represented by the matrix:
[2 0]
[0 5]
[1 0]
Domain: Vector space of dimension 3 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field

>>> f = lambda x: vector(QQ, [Integer(2)*x[Integer(0)]+x[Integer(2)],␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(5)*x[Integer(1)]])
>>> rho = linear_transformation(QQ**Integer(3), QQ**Integer(2), f)
>>> rho
Vector space morphism represented by the matrix:
[2 0]
[0 5]
[1 0]
Domain: Vector space of dimension 3 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field

>>> # needs sage.symbolic
>>> x, y, z = var('x y z')
>>> __tmp__=var("x,y,z"); h = symbolic_expression([Integer(2)*x + z,␣
↪Integer(5)*y]).function(x,y,z)
>>> zeta = linear_transformation(QQ**Integer(3), QQ**Integer(2), h)
>>> zeta
Vector space morphism represented by the matrix:
[2 0]
[0 5]
[1 0]
Domain:   Vector space of dimension 3 over Rational Field
Codomain: Vector space of dimension 2 over Rational Field

>>> phi == rho
True
>>> rho == zeta                                                          #␣
↪needs sage.symbolic
True
```

We create a linear transformation relative to non-standard bases, and capture its representation relative to standard bases. With this, we can build functions that create the same linear transformation relative to the nonstandard bases.

```
sage: u = vector(QQ, [1, -1])
sage: v = vector(QQ, [2, 3])
sage: D = (QQ^2).subspace_with_basis([u, v])
sage: x = vector(QQ, [2, 1])
sage: y = vector(QQ, [-1, 4])
sage: C = (QQ^2).subspace_with_basis([x, y])
sage: A = matrix(QQ, [[2, 5], [3, 7]])
sage: psi = linear_transformation(D, C, A)
sage: rho = psi.restrict_codomain(QQ^2).restrict_domain(QQ^2)
sage: rho.matrix()
[ -4/5  97/5]
[  1/5 -13/5]

sage: f = lambda x: vector(QQ, [(-4/5)*x[0] + (1/5)*x[1], (97/5)*x[0] + (-13/
↪5)*x[1]])
sage: psi = linear_transformation(D, C, f)
sage: psi.matrix()
[2 5]
[3 7]
```

(continues on next page)

(continued from previous page)

```
sage: # needs sage.symbolic
sage: s, t = var('s t')
sage: h(s, t) = [(-4/5)*s + (1/5)*t, (97/5)*s + (-13/5)*t]
sage: zeta = linear_transformation(D, C, h)
sage: zeta.matrix()
[2 5]
[3 7]
```

```
>>> from sage.all import *
>>> u = vector(QQ, [Integer(1), -Integer(1)])
>>> v = vector(QQ, [Integer(2), Integer(3)])
>>> D = (QQ**Integer(2)).subspace_with_basis([u, v])
>>> x = vector(QQ, [Integer(2), Integer(1)])
>>> y = vector(QQ, [-Integer(1), Integer(4)])
>>> C = (QQ**Integer(2)).subspace_with_basis([x, y])
>>> A = matrix(QQ, [[Integer(2), Integer(5)], [Integer(3), Integer(7)]])
>>> psi = linear_transformation(D, C, A)
>>> rho = psi.restrict_codomain(QQ**Integer(2)).restrict_domain(QQ**Integer(2))
>>> rho.matrix()
[ -4/5  97/5]
[  1/5 -13/5]

>>> f = lambda x: vector(QQ, [(-Integer(4)/Integer(5))*x[Integer(0)] +␣
↪(Integer(1)/Integer(5))*x[Integer(1)], (Integer(97)/Integer(5))*x[Integer(0)] +␣
↪(-Integer(13)/Integer(5))*x[Integer(1)]])
>>> psi = linear_transformation(D, C, f)
>>> psi.matrix()
[2 5]
[3 7]

>>> # needs sage.symbolic
>>> s, t = var('s t')
>>> __tmp__=var("s,t"); h = symbolic_expression([(-Integer(4)/Integer(5))*s +␣
↪(Integer(1)/Integer(5))*t, (Integer(97)/Integer(5))*s + (-Integer(13)/
↪Integer(5))*t]).function(s,t)
>>> zeta = linear_transformation(D, C, h)
>>> zeta.matrix()
[2 5]
[3 7]
```

Finally, we can give an explicit list of images for the basis elements of the domain.

```
sage: # needs sage.rings.number_field
sage: x = polygen(QQ)
sage: F.<a> = NumberField(x^3 + x + 1)
sage: u = vector(F, [1, a, a^2])
sage: v = vector(F, [a, a^2, 2])
sage: w = u + v
sage: D = F^3
sage: C = F^3
sage: rho = linear_transformation(D, C, [u, v, w])
sage: rho.matrix()
```

(continues on next page)

(continued from previous page)

```
[      1       a     a^2]
[      a     a^2       2]
[  a + 1 a^2 + a a^2 + 2]
sage: C = (F^3).subspace_with_basis([u, v])
sage: D = (F^3).subspace_with_basis([u, v])
sage: psi = linear_transformation(C, D, [u+v, u-v])
sage: psi.matrix()
[ 1  1]
[ 1 -1]
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> x = polygen(QQ)
>>> F = NumberField(x**Integer(3) + x + Integer(1), names=('a',)); (a,) = F._
↪first_ngens(1)
>>> u = vector(F, [Integer(1), a, a**Integer(2)])
>>> v = vector(F, [a, a**Integer(2), Integer(2)])
>>> w = u + v
>>> D = F**Integer(3)
>>> C = F**Integer(3)
>>> rho = linear_transformation(D, C, [u, v, w])
>>> rho.matrix()
[      1       a     a^2]
[      a     a^2       2]
[  a + 1 a^2 + a a^2 + 2]
>>> C = (F**Integer(3)).subspace_with_basis([u, v])
>>> D = (F**Integer(3)).subspace_with_basis([u, v])
>>> psi = linear_transformation(C, D, [u+v, u-v])
>>> psi.matrix()
[ 1  1]
[ 1 -1]
```

7.3 Homspaces between free modules

EXAMPLES:

We create $\mathrm{End}(\mathbf{Z}^2)$ and compute a basis.

```
sage: M = FreeModule(IntegerRing(), 2)
sage: E = End(M)
sage: B = E.basis()
sage: len(B)
4
sage: B[0]
Free module morphism defined by the matrix
[1 0]
[0 0]
Domain:   Ambient free module of rank 2 over the principal ideal domain ...
Codomain: Ambient free module of rank 2 over the principal ideal domain ...
```

```
>>> from sage.all import *
>>> M = FreeModule(IntegerRing(), Integer(2))
>>> E = End(M)
>>> B = E.basis()
>>> len(B)
4
>>> B[Integer(0)]
Free module morphism defined by the matrix
[1 0]
[0 0]
Domain:   Ambient free module of rank 2 over the principal ideal domain ...
Codomain: Ambient free module of rank 2 over the principal ideal domain ...
```

We create $\mathrm{Hom}(\mathbf{Z}^3, \mathbf{Z}^2)$ and compute a basis.

```
sage: V3 = FreeModule(IntegerRing(), 3)
sage: V2 = FreeModule(IntegerRing(), 2)
sage: H = Hom(V3, V2)
sage: H
Set of Morphisms
 from Ambient free module of rank 3 over the principal ideal domain Integer Ring
   to Ambient free module of rank 2 over the principal ideal domain Integer Ring
   in Category of finite dimensional modules with basis over
      (Dedekind domains and euclidean domains
       and noetherian rings
       and infinite enumerated sets and metric spaces)
sage: B = H.basis()
sage: len(B)
6
sage: B[0]
Free module morphism defined by the matrix
[1 0]
[0 0]
[0 0]...
```

```
>>> from sage.all import *
>>> V3 = FreeModule(IntegerRing(), Integer(3))
>>> V2 = FreeModule(IntegerRing(), Integer(2))
>>> H = Hom(V3, V2)
>>> H
Set of Morphisms
 from Ambient free module of rank 3 over the principal ideal domain Integer Ring
   to Ambient free module of rank 2 over the principal ideal domain Integer Ring
   in Category of finite dimensional modules with basis over
      (Dedekind domains and euclidean domains
       and noetherian rings
       and infinite enumerated sets and metric spaces)
>>> B = H.basis()
>>> len(B)
6
>>> B[Integer(0)]
Free module morphism defined by the matrix
[1 0]
```

(continues on next page)

(continued from previous page)

```
[0 0]
[0 0]...
```

class sage.modules.free_module_homspace.**FreeModuleHomspace**(*X*, *Y*, *category=None*, *check=True*, *base=None*)

Bases: HomsetWithBase

basis(*side='left'*)

Return a basis for this space of free module homomorphisms.

INPUT:

- `side` – side of the vectors acted on by the matrix (default: `'left'`)

OUTPUT: tuple

EXAMPLES:

```
sage: H = Hom(ZZ^2, ZZ^1)
sage: H.basis()
(Free module morphism defined by the matrix
  [1]
  [0]
  Domain:   Ambient free module of rank 2 over the principal ideal domain ...
  Codomain: Ambient free module of rank 1 over the principal ideal domain ...,
 Free module morphism defined by the matrix
  [0]
  [1]
  Domain:   Ambient free module of rank 2 over the principal ideal domain ...
  Codomain: Ambient free module of rank 1 over the principal ideal domain ...)
sage: H.basis("right")
(Free module morphism defined as left-multiplication by the matrix
  [1 0]
  Domain:   Ambient free module of rank 2 over the principal ideal domain ...
  Codomain: Ambient free module of rank 1 over the principal ideal domain ...,
 Free module morphism defined as left-multiplication by the matrix
  [0 1]
  Domain:   Ambient free module of rank 2 over the principal ideal domain ...
  Codomain: Ambient free module of rank 1 over the principal ideal domain ...)
```

```
>>> from sage.all import *
>>> H = Hom(ZZ**Integer(2), ZZ**Integer(1))
>>> H.basis()
(Free module morphism defined by the matrix
  [1]
  [0]
  Domain:   Ambient free module of rank 2 over the principal ideal domain ...
  Codomain: Ambient free module of rank 1 over the principal ideal domain ...,
 Free module morphism defined by the matrix
  [0]
  [1]
  Domain:   Ambient free module of rank 2 over the principal ideal domain ...
  Codomain: Ambient free module of rank 1 over the principal ideal domain ...)
>>> H.basis("right")
```

(continues on next page)

(continued from previous page)

```
(Free module morphism defined as left-multiplication by the matrix
  [1 0]
  Domain:   Ambient free module of rank 2 over the principal ideal domain ...
  Codomain: Ambient free module of rank 1 over the principal ideal domain ...,
 Free module morphism defined as left-multiplication by the matrix
  [0 1]
  Domain:   Ambient free module of rank 2 over the principal ideal domain ...
  Codomain: Ambient free module of rank 1 over the principal ideal domain ...)
```

identity(*side='left'*)

Return identity morphism in an endomorphism ring.

INPUT:

- `side` – side of the vectors acted on by the matrix (default: `'left'`)

EXAMPLES:

```
sage: V = FreeModule(ZZ,5)
sage: H = V.Hom(V)
sage: H.identity()
Free module morphism defined by the matrix
[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]
Domain:   Ambient free module of rank 5 over the principal ideal domain ...
Codomain: Ambient free module of rank 5 over the principal ideal domain ...
```

```
>>> from sage.all import *
>>> V = FreeModule(ZZ,Integer(5))
>>> H = V.Hom(V)
>>> H.identity()
Free module morphism defined by the matrix
[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]
Domain:   Ambient free module of rank 5 over the principal ideal domain ...
Codomain: Ambient free module of rank 5 over the principal ideal domain ...
```

zero(*side='left'*)

INPUT:

- `side` – side of the vectors acted on by the matrix (default: `'left'`)

EXAMPLES:

```
sage: E = ZZ^2
sage: F = ZZ^3
sage: H = Hom(E, F)
sage: f = H.zero()
```

(continues on next page)

(continued from previous page)

```
sage: f
Free module morphism defined by the matrix
[0 0 0]
[0 0 0]
Domain:   Ambient free module of rank 2 over the principal ideal domain␣
↪Integer Ring
Codomain: Ambient free module of rank 3 over the principal ideal domain␣
↪Integer Ring
sage: f(E.an_element())
(0, 0, 0)
sage: f(E.an_element()) == F.zero()
True
sage: H.zero("right")
Free module morphism defined as left-multiplication by the matrix
[0 0]
[0 0]
[0 0]
Domain:   Ambient free module of rank 2 over the principal ideal domain␣
↪Integer Ring
Codomain: Ambient free module of rank 3 over the principal ideal domain␣
↪Integer Ring
```

```
>>> from sage.all import *
>>> E = ZZ**Integer(2)
>>> F = ZZ**Integer(3)
>>> H = Hom(E, F)
>>> f = H.zero()
>>> f
Free module morphism defined by the matrix
[0 0 0]
[0 0 0]
Domain:   Ambient free module of rank 2 over the principal ideal domain␣
↪Integer Ring
Codomain: Ambient free module of rank 3 over the principal ideal domain␣
↪Integer Ring
>>> f(E.an_element())
(0, 0, 0)
>>> f(E.an_element()) == F.zero()
True
>>> H.zero("right")
Free module morphism defined as left-multiplication by the matrix
[0 0]
[0 0]
[0 0]
Domain:   Ambient free module of rank 2 over the principal ideal domain␣
↪Integer Ring
Codomain: Ambient free module of rank 3 over the principal ideal domain␣
↪Integer Ring
```

sage.modules.free_module_homspace.**is_FreeModuleHomspace**(*x*)

Return `True` if `x` is a free module homspace.

Notice that every vector space is a free module, but when we construct a set of morphisms between two vector

spaces, it is a `VectorSpaceHomspace`, which qualifies as a `FreeModuleHomspace`, since the former is special case of the latter.

EXAMPLES:

```
sage: H = Hom(ZZ^3, ZZ^2)
sage: type(H)
<class 'sage.modules.free_module_homspace.FreeModuleHomspace_with_category'>
sage: sage.modules.free_module_homspace.is_FreeModuleHomspace(H)
doctest:warning...
DeprecationWarning: the function is_FreeModuleHomspace is deprecated;
use 'isinstance(..., FreeModuleHomspace)' instead
See https://github.com/sagemath/sage/issues/37924 for details.
True

sage: K = Hom(QQ^3, ZZ^2)
sage: type(K)
<class 'sage.modules.free_module_homspace.FreeModuleHomspace_with_category'>
sage: sage.modules.free_module_homspace.is_FreeModuleHomspace(K)
True

sage: L = Hom(ZZ^3, QQ^2)
sage: type(L)
<class 'sage.modules.free_module_homspace.FreeModuleHomspace_with_category'>
sage: sage.modules.free_module_homspace.is_FreeModuleHomspace(L)
True

sage: P = Hom(QQ^3, QQ^2)
sage: type(P)
<class 'sage.modules.vector_space_homspace.VectorSpaceHomspace_with_category'>
sage: sage.modules.free_module_homspace.is_FreeModuleHomspace(P)
True

sage: sage.modules.free_module_homspace.is_FreeModuleHomspace('junk')
False
```

```
>>> from sage.all import *
>>> H = Hom(ZZ**Integer(3), ZZ**Integer(2))
>>> type(H)
<class 'sage.modules.free_module_homspace.FreeModuleHomspace_with_category'>
>>> sage.modules.free_module_homspace.is_FreeModuleHomspace(H)
doctest:warning...
DeprecationWarning: the function is_FreeModuleHomspace is deprecated;
use 'isinstance(..., FreeModuleHomspace)' instead
See https://github.com/sagemath/sage/issues/37924 for details.
True

>>> K = Hom(QQ**Integer(3), ZZ**Integer(2))
>>> type(K)
<class 'sage.modules.free_module_homspace.FreeModuleHomspace_with_category'>
>>> sage.modules.free_module_homspace.is_FreeModuleHomspace(K)
True

>>> L = Hom(ZZ**Integer(3), QQ**Integer(2))
```

(continues on next page)

(continued from previous page)

```
>>> type(L)
<class 'sage.modules.free_module_homspace.FreeModuleHomspace_with_category'>
>>> sage.modules.free_module_homspace.is_FreeModuleHomspace(L)
True

>>> P = Hom(QQ**Integer(3), QQ**Integer(2))
>>> type(P)
<class 'sage.modules.vector_space_homspace.VectorSpaceHomspace_with_category'>
>>> sage.modules.free_module_homspace.is_FreeModuleHomspace(P)
True

>>> sage.modules.free_module_homspace.is_FreeModuleHomspace('junk')
False
```

7.4 Morphisms of free modules

AUTHORS:

- William Stein: initial version
- Miguel Marco (2010-06-19): added eigenvalues, eigenvectors and minpoly functions

class sage.modules.free_module_morphism.**BaseIsomorphism1D**

Bases: Morphism

An isomorphism between a ring and a free rank-1 module over the ring.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: V, from_V, to_V = R.free_module(R)
sage: from_V
Isomorphism morphism:
  From: Ambient free module of rank 1 over the integral domain
        Multivariate Polynomial Ring in x, y over Rational Field
  To:   Multivariate Polynomial Ring in x, y over Rational Field
```

```
>>> from sage.all import *
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> V, from_V, to_V = R.free_module(R)
>>> from_V
Isomorphism morphism:
  From: Ambient free module of rank 1 over the integral domain
        Multivariate Polynomial Ring in x, y over Rational Field
  To:   Multivariate Polynomial Ring in x, y over Rational Field
```

is_injective()

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: V, from_V, to_V = R.free_module(R)
sage: from_V.is_injective()
True
```

```
>>> from sage.all import *
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> V, from_V, to_V = R.free_module(R)
>>> from_V.is_injective()
True
```

is_surjective()

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: V, from_V, to_V = R.free_module(R)
sage: from_V.is_surjective()
True
```

```
>>> from sage.all import *
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> V, from_V, to_V = R.free_module(R)
>>> from_V.is_surjective()
True
```

class sage.modules.free_module_morphism.**BaseIsomorphism1D_from_FM**(*parent*, *basis=None*)

Bases: *BaseIsomorphism1D*

An isomorphism to a ring from its 1-dimensional free module.

INPUT:

- `parent` – the homset
- `basis` – (default: 1) an invertible element of the ring

EXAMPLES:

```
sage: R.<x> = QQ[[]]
sage: V, from_V, to_V = R.free_module(R)
sage: v = to_V(1+x); v
(1 + x)
sage: from_V(v)
1 + x
sage: W, from_W, to_W = R.free_module(R, basis=(1-x))
sage: W is V
True
sage: w = to_W(1+x); w
(1 - x^2)
sage: from_W(w)
1 + x + O(x^20)
```

```
>>> from sage.all import *
>>> R = QQ[['x']]; (x,) = R._first_ngens(1)
>>> V, from_V, to_V = R.free_module(R)
>>> v = to_V(Integer(1)+x); v
(1 + x)
>>> from_V(v)
1 + x
```

(continues on next page)

(continued from previous page)

```
>>> W, from_W, to_W = R.free_module(R, basis=(Integer(1)-x))
>>> W is V
True
>>> w = to_W(Integer(1)+x); w
(1 - x^2)
>>> from_W(w)
1 + x + O(x^20)
```

The basis vector has to be a unit so that the map is an isomorphism:

```
sage: W, from_W, to_W = R.free_module(R, basis=x)
Traceback (most recent call last):
...
ValueError: basis element must be a unit
```

```
>>> from sage.all import *
>>> W, from_W, to_W = R.free_module(R, basis=x)
Traceback (most recent call last):
...
ValueError: basis element must be a unit
```

class sage.modules.free_module_morphism.**BaseIsomorphism1D_to_FM**(*parent*, *basis=None*)

Bases: *BaseIsomorphism1D*

An isomorphism from a ring to its 1-dimensional free module.

INPUT:

- `parent` – the homset
- `basis` – (default: 1) an invertible element of the ring

EXAMPLES:

```
sage: R = Zmod(8)
sage: V, from_V, to_V = R.free_module(R)
sage: v = to_V(2); v
(2)
sage: from_V(v)
2
sage: W, from_W, to_W = R.free_module(R, basis=3)
sage: W is V
True
sage: w = to_W(2); w
(6)
sage: from_W(w)
2
```

```
>>> from sage.all import *
>>> R = Zmod(Integer(8))
>>> V, from_V, to_V = R.free_module(R)
>>> v = to_V(Integer(2)); v
(2)
>>> from_V(v)
```

(continues on next page)

(continued from previous page)

```
2
>>> W, from_W, to_W = R.free_module(R, basis=Integer(3))
>>> W is V
True
>>> w = to_W(Integer(2)); w
(6)
>>> from_W(w)
2
```

The basis vector has to be a unit so that the map is an isomorphism:

```
sage: W, from_W, to_W = R.free_module(R, basis=4)
Traceback (most recent call last):
...
ValueError: basis element must be a unit
```

```
>>> from sage.all import *
>>> W, from_W, to_W = R.free_module(R, basis=Integer(4))
Traceback (most recent call last):
...
ValueError: basis element must be a unit
```

class sage.modules.free_module_morphism.**FreeModuleMorphism**(*parent*, *A*, *side='left'*)

Bases: *MatrixMorphism*

INPUT:

- `parent` – a homspace in a (sub) category of free modules
- `A` – matrix
- `side` – side of the vectors acted on by the matrix (default: `'left'`)

EXAMPLES:

```
sage: V = ZZ^3; W = span([[1,2,3], [-1,2,8]], ZZ)
sage: phi = V.hom(matrix(ZZ, 3, [1..9]))
sage: type(phi)
<class 'sage.modules.free_module_morphism.FreeModuleMorphism'>
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(3); W = span([[Integer(1),Integer(2),Integer(3)], [-
↪Integer(1),Integer(2),Integer(8)]], ZZ)
>>> phi = V.hom(matrix(ZZ, Integer(3), (ellipsis_range(Integer(1),Ellipsis,
↪Integer(9)))))
>>> type(phi)
<class 'sage.modules.free_module_morphism.FreeModuleMorphism'>
```

change_ring(*R*)

Change the ring over which this morphism is defined.

This changes the ring of the domain, codomain, and underlying matrix.

EXAMPLES:

```
sage: V0 = span([[0,0,1],[0,2,0]], ZZ); V1 = span([[1/2,0],[0,2]], ZZ)
sage: W = span([[1,0],[0,6]], ZZ)
sage: h = V0.hom([-3*V1.0 - 3*V1.1, -3*V1.0 - 3*V1.1])
sage: h.base_ring()
Integer Ring
sage: h
Free module morphism defined by the matrix
[-3 -3]
[-3 -3]...
sage: h.change_ring(QQ).base_ring()
Rational Field
sage: f = h.change_ring(QQ); f
Vector space morphism represented by the matrix:
[-3 -3]
[-3 -3]
Domain:   Vector space of degree 3 and dimension 2 over Rational Field
          Basis matrix:
          [0 1 0]
          [0 0 1]
Codomain: Vector space of degree 2 and dimension 2 over Rational Field
          Basis matrix:
          [1 0]
          [0 1]
sage: f = h.change_ring(GF(7)); f
Vector space morphism represented by the matrix:
[4 4]
[4 4]
Domain:   Vector space of degree 3 and dimension 2 over Finite Field of size 7
          Basis matrix:
          [0 1 0]
          [0 0 1]
Codomain: Vector space of degree 2 and dimension 2 over Finite Field of size 7
          Basis matrix:
          [1 0]
          [0 1]
```

```
>>> from sage.all import *
>>> V0 = span([[Integer(0),Integer(0),Integer(1)],[Integer(0),Integer(2),
↪Integer(0)]], ZZ); V1 = span([[Integer(1)/Integer(2),Integer(0)],
↪[Integer(0),Integer(2)]], ZZ)
>>> W = span([[Integer(1),Integer(0)],[Integer(0),Integer(6)]], ZZ)
>>> h = V0.hom([-Integer(3)*V1.gen(0) - Integer(3)*V1.gen(1), -Integer(3)*V1.
↪gen(0) - Integer(3)*V1.gen(1)])
>>> h.base_ring()
Integer Ring
>>> h
Free module morphism defined by the matrix
[-3 -3]
[-3 -3]...
>>> h.change_ring(QQ).base_ring()
Rational Field
>>> f = h.change_ring(QQ); f
```

(continues on next page)

(continued from previous page)

```
Vector space morphism represented by the matrix:
[-3 -3]
[-3 -3]
Domain:   Vector space of degree 3 and dimension 2 over Rational Field
          Basis matrix:
          [0 1 0]
          [0 0 1]
Codomain: Vector space of degree 2 and dimension 2 over Rational Field
          Basis matrix:
          [1 0]
          [0 1]
>>> f = h.change_ring(GF(Integer(7))); f
Vector space morphism represented by the matrix:
[4 4]
[4 4]
Domain:   Vector space of degree 3 and dimension 2 over Finite Field of size 7
          Basis matrix:
          [0 1 0]
          [0 0 1]
Codomain: Vector space of degree 2 and dimension 2 over Finite Field of size 7
          Basis matrix:
          [1 0]
          [0 1]
```

eigenspaces(*extend=True*)

Compute a list of subspaces formed by eigenvectors of `self`.

INPUT:

- `extend` – boolean (default: `True`); determines if field extensions should be considered

OUTPUT: a list of pairs `(eigenvalue, eigenspace)`

EXAMPLES:

```
sage: V = QQ^3
sage: h = V.hom([[1,0,0], [0,0,1], [0,-1,0]], V)
sage: h.eigenspaces()                                                          #␣
↪needs sage.rings.number_field
[(1,    Vector space of degree 3 and dimension 1 over Rational Field
         Basis matrix:
         [1 0 0]),
 (-1*I, Vector space of degree 3 and dimension 1 over Algebraic Field
         Basis matrix:
         [  0   1 1*I]),
 (1*I,  Vector space of degree 3 and dimension 1 over Algebraic Field
         Basis matrix:
         [   0    1 -1*I])]

sage: h.eigenspaces(extend=False)                                              #␣
↪needs sage.rings.number_field
[(1,
  Vector space of degree 3 and dimension 1 over Rational Field
  Basis matrix:
```

(continues on next page)

(continued from previous page)

```
  [1 0 0])]

sage: h = V.hom([[2,1,0], [0,2,0], [0,0,-1]], V)
sage: h.eigenspaces()                                                    #␣
↪needs sage.rings.number_field
[(-1, Vector space of degree 3 and dimension 1 over Rational Field
       Basis matrix:
       [0 0 1]),
 (2, Vector space of degree 3 and dimension 1 over Rational Field
      Basis matrix:
      [0 1 0])]

sage: h = V.hom([[2,1,0], [0,2,0], [0,0,2]], V)
sage: h.eigenspaces()                                                    #␣
↪needs sage.rings.number_field
[(2, Vector space of degree 3 and dimension 2 over Rational Field
      Basis matrix:
      [0 1 0]
      [0 0 1])]
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3)
>>> h = V.hom([[Integer(1),Integer(0),Integer(0)], [Integer(0),Integer(0),
↪Integer(1)], [Integer(0),-Integer(1),Integer(0)]], V)
>>> h.eigenspaces()                                                      #␣
↪needs sage.rings.number_field
[(1,    Vector space of degree 3 and dimension 1 over Rational Field
         Basis matrix:
         [1 0 0]),
 (-1*I, Vector space of degree 3 and dimension 1 over Algebraic Field
         Basis matrix:
         [  0   1 1*I]),
 (1*I,  Vector space of degree 3 and dimension 1 over Algebraic Field
         Basis matrix:
         [   0    1 -1*I])]

>>> h.eigenspaces(extend=False)                                          #␣
↪needs sage.rings.number_field
[(1,
  Vector space of degree 3 and dimension 1 over Rational Field
  Basis matrix:
  [1 0 0])]

>>> h = V.hom([[Integer(2),Integer(1),Integer(0)], [Integer(0),Integer(2),
↪Integer(0)], [Integer(0),Integer(0),-Integer(1)]], V)
>>> h.eigenspaces()                                                      #␣
↪needs sage.rings.number_field
[(-1, Vector space of degree 3 and dimension 1 over Rational Field
       Basis matrix:
       [0 0 1]),
 (2, Vector space of degree 3 and dimension 1 over Rational Field
      Basis matrix:
```

(continues on next page)

(continued from previous page)

```
      [0 1 0])]

>>> h = V.hom([[Integer(2),Integer(1),Integer(0)], [Integer(0),Integer(2),
↪Integer(0)], [Integer(0),Integer(0),Integer(2)]], V)
>>> h.eigenspaces()                                                    #␣
↪needs sage.rings.number_field
[(2, Vector space of degree 3 and dimension 2 over Rational Field
      Basis matrix:
      [0 1 0]
      [0 0 1])]
```

```
sage: V = QQ^2; m = matrix(2, [1, 1, 0, 1])
sage: V.hom(m, side='right').eigenspaces()                            #␣
↪needs sage.rings.number_field
[(1, Vector space of degree 2 and dimension 1 over Rational Field
      Basis matrix:
      [1 0])]
sage: V.hom(m).eigenspaces()                                          #␣
↪needs sage.rings.number_field
[(1, Vector space of degree 2 and dimension 1 over Rational Field
      Basis matrix:
      [0 1])]
```

```
>>> from sage.all import *
>>> V = QQ**Integer(2); m = matrix(Integer(2), [Integer(1), Integer(1),␣
↪Integer(0), Integer(1)])
>>> V.hom(m, side='right').eigenspaces()                              #␣
↪needs sage.rings.number_field
[(1, Vector space of degree 2 and dimension 1 over Rational Field
      Basis matrix:
      [1 0])]
>>> V.hom(m).eigenspaces()                                            #␣
↪needs sage.rings.number_field
[(1, Vector space of degree 2 and dimension 1 over Rational Field
      Basis matrix:
      [0 1])]
```

eigenvalues(*extend=True*)

Return a list with the eigenvalues of the endomorphism of vector spaces.

INPUT:

- `extend` – boolean (default: `True`); decides if base field extensions should be considered or not

EXAMPLES:

We compute the eigenvalues of an endomorphism of $\mathbf{Q}^3$:

```
sage: V = QQ^3
sage: H = V.endomorphism_ring()([[1,-1,0], [-1,1,1], [0,3,1]])
sage: H.eigenvalues()                                                 #␣
↪needs sage.rings.number_field
[3, 1, -1]
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3)
>>> H = V.endomorphism_ring()([[Integer(1),-Integer(1),Integer(0)], [-
↪Integer(1),Integer(1),Integer(1)], [Integer(0),Integer(3),Integer(1)]])
>>> H.eigenvalues()                                                    #␣
↪needs sage.rings.number_field
[3, 1, -1]
```

Note the effect of the `extend` option:

```
sage: V = QQ^2
sage: H = V.endomorphism_ring()([[0,-1], [1,0]])
sage: H.eigenvalues()                                                  #␣
↪needs sage.rings.number_field
[-1*I, 1*I]
sage: H.eigenvalues(extend=False)                                      #␣
↪needs sage.libs.pari
[]
```

```
>>> from sage.all import *
>>> V = QQ**Integer(2)
>>> H = V.endomorphism_ring()([[Integer(0),-Integer(1)], [Integer(1),
↪Integer(0)]])
>>> H.eigenvalues()                                                    #␣
↪needs sage.rings.number_field
[-1*I, 1*I]
>>> H.eigenvalues(extend=False)                                        #␣
↪needs sage.libs.pari
[]
```

eigenvectors(*extend=True*)

Compute the subspace of eigenvectors of a given eigenvalue.

INPUT:

- `extend` – boolean (default: `True`); decides if base field extensions should be considered or not

OUTPUT:

A sequence of tuples. Each tuple contains an eigenvalue, a sequence with a basis of the corresponding subspace of eigenvectors, and the algebraic multiplicity of the eigenvalue.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: V = (QQ^4).subspace([[0,2,1,4], [1,2,5,0], [1,1,1,1]])
sage: H = (V.Hom(V))(matrix(QQ, [[0,1,0], [-1,0,0], [0,0,3]]))
sage: H.eigenvectors()
[(3, [(0, 0, 1, -6/7)], 1),
 (-1*I, [(1, 1*I, 0, -0.571428571428572? + 2.428571428571429?*I)], 1),
 (1*I, [(1, -1*I, 0, -0.571428571428572? - 2.428571428571429?*I)], 1)]
sage: H.eigenvectors(extend=False)
[(3, [(0, 0, 1, -6/7)], 1)]
sage: H1 = (V.Hom(V))(matrix(QQ, [[2,1,0],[0,2,0],[0,0,3]]))
sage: H1.eigenvectors()
```

(continues on next page)

(continued from previous page)

```
[(3, [(0, 0, 1, -6/7)], 1), (2, [(0, 1, 0, 17/7)], 2)]
sage: H1.eigenvectors(extend=False)
[(3, [(0, 0, 1, -6/7)], 1), (2, [(0, 1, 0, 17/7)], 2)]
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> V = (QQ**Integer(4)).subspace([[Integer(0),Integer(2),Integer(1),
↪Integer(4)], [Integer(1),Integer(2),Integer(5),Integer(0)], [Integer(1),
↪Integer(1),Integer(1),Integer(1)]])
>>> H = (V.Hom(V))(matrix(QQ, [[Integer(0),Integer(1),Integer(0)], [-
↪Integer(1),Integer(0),Integer(0)], [Integer(0),Integer(0),Integer(3)]]))
>>> H.eigenvectors()
[(3, [(0, 0, 1, -6/7)], 1),
 (-1*I, [(1, 1*I, 0, -0.571428571428572? + 2.428571428571429?*I)], 1),
 (1*I, [(1, -1*I, 0, -0.571428571428572? - 2.428571428571429?*I)], 1)]
>>> H.eigenvectors(extend=False)
[(3, [(0, 0, 1, -6/7)], 1)]
>>> H1 = (V.Hom(V))(matrix(QQ, [[Integer(2),Integer(1),Integer(0)],
↪[Integer(0),Integer(2),Integer(0)],[Integer(0),Integer(0),Integer(3)]]))
>>> H1.eigenvectors()
[(3, [(0, 0, 1, -6/7)], 1), (2, [(0, 1, 0, 17/7)], 2)]
>>> H1.eigenvectors(extend=False)
[(3, [(0, 0, 1, -6/7)], 1), (2, [(0, 1, 0, 17/7)], 2)]
```

```
sage: V = QQ^2
sage: m = matrix(2, [1, 1, 0, 1])
sage: V.hom(m, side='right').eigenvectors()                                 #␣
↪needs sage.rings.number_field
[(1, [(1, 0)], 2)]
sage: V.hom(m).eigenvectors()                                               #␣
↪needs sage.rings.number_field
[(1, [(0, 1)], 2)]
```

```
>>> from sage.all import *
>>> V = QQ**Integer(2)
>>> m = matrix(Integer(2), [Integer(1), Integer(1), Integer(0), Integer(1)])
>>> V.hom(m, side='right').eigenvectors()                                   #␣
↪needs sage.rings.number_field
[(1, [(1, 0)], 2)]
>>> V.hom(m).eigenvectors()                                                 #␣
↪needs sage.rings.number_field
[(1, [(0, 1)], 2)]
```

inverse_image(*V*)

Given a submodule V of the codomain of self, return the inverse image of V under self, i.e., the biggest submodule of the domain of `self` that maps into V.

EXAMPLES:

We test computing inverse images over a field:

```
sage: V = QQ^3; W = span([[1,2,3], [-1,2,5/3]], QQ)
sage: phi = V.hom(matrix(QQ, 3, [1..9]))
sage: phi.rank()
2
sage: I = phi.inverse_image(W); I
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[   1    0    0]
[   0    1 -1/2]
sage: phi(I.0) in W
True
sage: phi(I.1) in W
True
sage: W = phi.image()
sage: phi.inverse_image(W) == V
True
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3); W = span([[Integer(1),Integer(2),Integer(3)], [-
↪Integer(1),Integer(2),Integer(5)/Integer(3)]], QQ)
>>> phi = V.hom(matrix(QQ, Integer(3), (ellipsis_range(Integer(1),Ellipsis,
↪Integer(9)))))
>>> phi.rank()
2
>>> I = phi.inverse_image(W); I
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[   1    0    0]
[   0    1 -1/2]
>>> phi(I.gen(0)) in W
True
>>> phi(I.gen(1)) in W
True
>>> W = phi.image()
>>> phi.inverse_image(W) == V
True
```

We test computing inverse images between two spaces embedded in different ambient spaces.:

```
sage: V0 = span([[0,0,1],[0,2,0]],ZZ); V1 = span([[1/2,0],[0,2]],ZZ)
sage: W = span([[1,0],[0,6]],ZZ)
sage: h = V0.hom([-3*V1.0 - 3*V1.1, -3*V1.0 - 3*V1.1])
sage: h.inverse_image(W)
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[0 2 1]
[0 0 2]
sage: h(h.inverse_image(W)).is_submodule(W)
True
sage: h(h.inverse_image(W)).index_in(W)
+Infinity
sage: h(h.inverse_image(W))
Free module of degree 2 and rank 1 over Integer Ring
```

(continues on next page)

(continued from previous page)

```
Echelon basis matrix:
[ 3 12]
```

```
>>> from sage.all import *
>>> V0 = span([[Integer(0),Integer(0),Integer(1)],[Integer(0),Integer(2),
↪Integer(0)]],ZZ); V1 = span([[Integer(1)/Integer(2),Integer(0)],[Integer(0),
↪Integer(2)]],ZZ)
>>> W = span([[Integer(1),Integer(0)],[Integer(0),Integer(6)]],ZZ)
>>> h = V0.hom([-Integer(3)*V1.gen(0) - Integer(3)*V1.gen(1), -Integer(3)*V1.
↪gen(0) - Integer(3)*V1.gen(1)])
>>> h.inverse_image(W)
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[0 2 1]
[0 0 2]
>>> h(h.inverse_image(W)).is_submodule(W)
True
>>> h(h.inverse_image(W)).index_in(W)
+Infinity
>>> h(h.inverse_image(W))
Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[ 3 12]
```

We test computing inverse images over the integers:

```
sage: V = QQ^3; W = V.span_of_basis([[2,2,3],[-1,2,5/3]], ZZ)
sage: phi = W.hom([W.0, W.0 - W.1])
sage: Z = W.span([2*W.1]); Z
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[    2    -4 -10/3]
sage: Y = phi.inverse_image(Z); Y
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[  6   0 8/3]
sage: phi(Y) == Z
True
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3); W = V.span_of_basis([[Integer(2),Integer(2),
↪Integer(3)],[-Integer(1),Integer(2),Integer(5)/Integer(3)]], ZZ)
>>> phi = W.hom([W.gen(0), W.gen(0) - W.gen(1)])
>>> Z = W.span([Integer(2)*W.gen(1)]); Z
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[    2    -4 -10/3]
>>> Y = phi.inverse_image(Z); Y
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[  6   0 8/3]
>>> phi(Y) == Z
```

(continues on next page)

(continued from previous page)

```
True
```

We test that Issue #24590 is resolved:

```
sage: A = FreeQuadraticModule(ZZ, 1, matrix([2]))
sage: f = A.Hom(A).an_element()
sage: f.inverse_image(A)
Free module of degree 1 and rank 1 over Integer Ring
Echelon basis matrix:
[1]
```

```
>>> from sage.all import *
>>> A = FreeQuadraticModule(ZZ, Integer(1), matrix([Integer(2)]))
>>> f = A.Hom(A).an_element()
>>> f.inverse_image(A)
Free module of degree 1 and rank 1 over Integer Ring
Echelon basis matrix:
[1]
```

We test that it respects the `side`:

```
sage: V = ZZ^2
sage: m = matrix(2, [1, 1, 0, 1])
sage: h = V.hom(m, side='right')
sage: h
Free module morphism defined as left-multiplication by the matrix
[1 1]
[0 1]...
sage: SV = V.span([V.0])
sage: h.inverse_image(SV)
Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[1 0]
sage: V.hom(m).inverse_image(SV)
Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[ 1 -1]
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(2)
>>> m = matrix(Integer(2), [Integer(1), Integer(1), Integer(0), Integer(1)])
>>> h = V.hom(m, side='right')
>>> h
Free module morphism defined as left-multiplication by the matrix
[1 1]
[0 1]...
>>> SV = V.span([V.gen(0)])
>>> h.inverse_image(SV)
Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[1 0]
>>> V.hom(m).inverse_image(SV)
```

(continues on next page)

(continued from previous page)

```
Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[ 1 -1]
```

lift(*x*)

Given an element of the image, return an element of the codomain that maps onto it.

Note that `lift` and `preimage_representative` are equivalent names for this method, with the latter suggesting that the return value is a coset representative of the domain modulo the kernel of the morphism.

EXAMPLES:

```
sage: X = QQ**2
sage: V = X.span([[2, 0], [0, 8]], ZZ)
sage: W = (QQ**1).span([[1/12]], ZZ)
sage: f = V.hom([W([1/3]), W([1/2])], W)
sage: l=f.lift([1/3]); l  # random
(8, -16)
sage: f(l)
(1/3)
sage: f(f.lift([1/2]))
(1/2)
sage: f(f.lift([1/6]))
(1/6)
sage: f.lift([1/12])
Traceback (most recent call last):
...
ValueError: element is not in the image
sage: f.lift([1/24])
Traceback (most recent call last):
...
TypeError: element [1/24] is not in free module
```

```
>>> from sage.all import *
>>> X = QQ**Integer(2)
>>> V = X.span([[Integer(2), Integer(0)], [Integer(0), Integer(8)]], ZZ)
>>> W = (QQ**Integer(1)).span([[Integer(1)/Integer(12)]], ZZ)
>>> f = V.hom([W([Integer(1)/Integer(3)]), W([Integer(1)/Integer(2)])], W)
>>> l=f.lift([Integer(1)/Integer(3)]); l  # random
(8, -16)
>>> f(l)
(1/3)
>>> f(f.lift([Integer(1)/Integer(2)]))
(1/2)
>>> f(f.lift([Integer(1)/Integer(6)]))
(1/6)
>>> f.lift([Integer(1)/Integer(12)])
Traceback (most recent call last):
...
ValueError: element is not in the image
>>> f.lift([Integer(1)/Integer(24)])
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
TypeError: element [1/24] is not in free module
```

This works for vector spaces, too:

```
sage: V = VectorSpace(GF(3), 2)
sage: W = VectorSpace(GF(3), 3)
sage: f = V.hom([W.1, W.1 - W.0])
sage: f.lift(W.1)
(1, 0)
sage: f.lift(W.2)
Traceback (most recent call last):
...
ValueError: element is not in the image
sage: w = W((17, -2, 0))
sage: f(f.lift(w)) == w
True
```

```
>>> from sage.all import *
>>> V = VectorSpace(GF(Integer(3)), Integer(2))
>>> W = VectorSpace(GF(Integer(3)), Integer(3))
>>> f = V.hom([W.gen(1), W.gen(1) - W.gen(0)])
>>> f.lift(W.gen(1))
(1, 0)
>>> f.lift(W.gen(2))
Traceback (most recent call last):
...
ValueError: element is not in the image
>>> w = W((Integer(17), -Integer(2), Integer(0)))
>>> f(f.lift(w)) == w
True
```

This example illustrates the use of the `preimage_representative` as an equivalent name for this method.

```
sage: V = ZZ^3
sage: W = ZZ^2
sage: w = vector(ZZ, [1,2])
sage: f = V.hom([w, w, w], W)
sage: f.preimage_representative(vector(ZZ, [10, 20]))
(0, 0, 10)
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(3)
>>> W = ZZ**Integer(2)
>>> w = vector(ZZ, [Integer(1),Integer(2)])
>>> f = V.hom([w, w, w], W)
>>> f.preimage_representative(vector(ZZ, [Integer(10), Integer(20)]))
(0, 0, 10)
```

```
sage: V = QQ^2; m = matrix(2, [1, 1, 0, 1])
sage: V.hom(m, side='right').lift(V.0 + V.1)
(0, 1)
```

(continues on next page)

(continued from previous page)

```
sage: V.hom(m).lift(V.0 + V.1)
(1, 0)
```

```
>>> from sage.all import *
>>> V = QQ**Integer(2); m = matrix(Integer(2), [Integer(1), Integer(1),␣
↪Integer(0), Integer(1)])
>>> V.hom(m, side='right').lift(V.gen(0) + V.gen(1))
(0, 1)
>>> V.hom(m).lift(V.gen(0) + V.gen(1))
(1, 0)
```

preimage_representative(*x*)

Given an element of the image, return an element of the codomain that maps onto it.

Note that `lift` and `preimage_representative` are equivalent names for this method, with the latter suggesting that the return value is a coset representative of the domain modulo the kernel of the morphism.

EXAMPLES:

```
sage: X = QQ**2
sage: V = X.span([[2, 0], [0, 8]], ZZ)
sage: W = (QQ**1).span([[1/12]], ZZ)
sage: f = V.hom([W([1/3]), W([1/2])], W)
sage: l=f.lift([1/3]); l  # random
(8, -16)
sage: f(l)
(1/3)
sage: f(f.lift([1/2]))
(1/2)
sage: f(f.lift([1/6]))
(1/6)
sage: f.lift([1/12])
Traceback (most recent call last):
...
ValueError: element is not in the image
sage: f.lift([1/24])
Traceback (most recent call last):
...
TypeError: element [1/24] is not in free module
```

```
>>> from sage.all import *
>>> X = QQ**Integer(2)
>>> V = X.span([[Integer(2), Integer(0)], [Integer(0), Integer(8)]], ZZ)
>>> W = (QQ**Integer(1)).span([[Integer(1)/Integer(12)]], ZZ)
>>> f = V.hom([W([Integer(1)/Integer(3)]), W([Integer(1)/Integer(2)])], W)
>>> l=f.lift([Integer(1)/Integer(3)]); l  # random
(8, -16)
>>> f(l)
(1/3)
>>> f(f.lift([Integer(1)/Integer(2)]))
(1/2)
>>> f(f.lift([Integer(1)/Integer(6)]))
```

(continues on next page)

(continued from previous page)

```
(1/6)
>>> f.lift([Integer(1)/Integer(12)])
Traceback (most recent call last):
...
ValueError: element is not in the image
>>> f.lift([Integer(1)/Integer(24)])
Traceback (most recent call last):
...
TypeError: element [1/24] is not in free module
```

This works for vector spaces, too:

```
sage: V = VectorSpace(GF(3), 2)
sage: W = VectorSpace(GF(3), 3)
sage: f = V.hom([W.1, W.1 - W.0])
sage: f.lift(W.1)
(1, 0)
sage: f.lift(W.2)
Traceback (most recent call last):
...
ValueError: element is not in the image
sage: w = W((17, -2, 0))
sage: f(f.lift(w)) == w
True
```

```
>>> from sage.all import *
>>> V = VectorSpace(GF(Integer(3)), Integer(2))
>>> W = VectorSpace(GF(Integer(3)), Integer(3))
>>> f = V.hom([W.gen(1), W.gen(1) - W.gen(0)])
>>> f.lift(W.gen(1))
(1, 0)
>>> f.lift(W.gen(2))
Traceback (most recent call last):
...
ValueError: element is not in the image
>>> w = W((Integer(17), -Integer(2), Integer(0)))
>>> f(f.lift(w)) == w
True
```

This example illustrates the use of the `preimage_representative` as an equivalent name for this method.

```
sage: V = ZZ^3
sage: W = ZZ^2
sage: w = vector(ZZ, [1,2])
sage: f = V.hom([w, w, w], W)
sage: f.preimage_representative(vector(ZZ, [10, 20]))
(0, 0, 10)
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(3)
>>> W = ZZ**Integer(2)
>>> w = vector(ZZ, [Integer(1),Integer(2)])
```

(continues on next page)

(continued from previous page)

```
>>> f = V.hom([w, w, w], W)
>>> f.preimage_representative(vector(ZZ, [Integer(10), Integer(20)]))
(0, 0, 10)
```

```
sage: V = QQ^2; m = matrix(2, [1, 1, 0, 1])
sage: V.hom(m, side='right').lift(V.0 + V.1)
(0, 1)
sage: V.hom(m).lift(V.0 + V.1)
(1, 0)
```

```
>>> from sage.all import *
>>> V = QQ**Integer(2); m = matrix(Integer(2), [Integer(1), Integer(1),␣
↪Integer(0), Integer(1)])
>>> V.hom(m, side='right').lift(V.gen(0) + V.gen(1))
(0, 1)
>>> V.hom(m).lift(V.gen(0) + V.gen(1))
(1, 0)
```

pushforward(*x*)

Compute the image of a sub-module of the domain.

EXAMPLES:

```
sage: V = QQ^3; W = span([[1,2,3], [-1,2,5/3]], QQ)
sage: phi = V.hom(matrix(QQ, 3, [1..9]))
sage: phi.rank()
2
sage: phi(V)   #indirect doctest
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3); W = span([[Integer(1),Integer(2),Integer(3)], [-
↪Integer(1),Integer(2),Integer(5)/Integer(3)]], QQ)
>>> phi = V.hom(matrix(QQ, Integer(3), (ellipsis_range(Integer(1),Ellipsis,
↪Integer(9)))))
>>> phi.rank()
2
>>> phi(V)   #indirect doctest
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
```

We compute the image of a submodule of a ZZ-module embedded in a rational vector space:

```
sage: V = QQ^3; W = V.span_of_basis([[2,2,3], [-1,2,5/3]], ZZ)
sage: phi = W.hom([W.0, W.0 - W.1]); phi
Free module morphism defined by the matrix
[ 1  0]
```

(continues on next page)

(continued from previous page)

```
[ 1 -1]...
sage: phi(span([2*W.1], ZZ))
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[  6   0 8/3]
sage: phi(2*W.1)
(6, 0, 8/3)
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3); W = V.span_of_basis([[Integer(2),Integer(2),
↪Integer(3)], [-Integer(1),Integer(2),Integer(5)/Integer(3)]], ZZ)
>>> phi = W.hom([W.gen(0), W.gen(0) - W.gen(1)]); phi
Free module morphism defined by the matrix
[ 1  0]
[ 1 -1]...
>>> phi(span([Integer(2)*W.gen(1)], ZZ))
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[  6   0 8/3]
>>> phi(Integer(2)*W.gen(1))
(6, 0, 8/3)
```

sage.modules.free_module_morphism.**is_FreeModuleMorphism**(*x*)

This function is deprecated.

EXAMPLES:

```
sage: V = ZZ^2; f = V.hom([V.1, -2*V.0])
sage: sage.modules.free_module_morphism.is_FreeModuleMorphism(f)
doctest:warning...
DeprecationWarning: is_FreeModuleMorphism is deprecated;
use isinstance(..., FreeModuleMorphism) or categories instead
See https://github.com/sagemath/sage/issues/37731 for details.
True
sage: sage.modules.free_module_morphism.is_FreeModuleMorphism(0)
False
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(2); f = V.hom([V.gen(1), -Integer(2)*V.gen(0)])
>>> sage.modules.free_module_morphism.is_FreeModuleMorphism(f)
doctest:warning...
DeprecationWarning: is_FreeModuleMorphism is deprecated;
use isinstance(..., FreeModuleMorphism) or categories instead
See https://github.com/sagemath/sage/issues/37731 for details.
True
>>> sage.modules.free_module_morphism.is_FreeModuleMorphism(Integer(0))
False
```

7.5 Morphisms defined by a matrix

A matrix morphism is a morphism that is defined by multiplication by a matrix. Elements of domain must either have a method `vector()` that returns a vector that the defining matrix can hit from the left, or be coercible into vector space of appropriate dimension.

EXAMPLES:

```
sage: from sage.modules.matrix_morphism import MatrixMorphism, is_MatrixMorphism
sage: V = QQ^3
sage: T = End(V)
sage: M = MatrixSpace(QQ,3)
sage: I = M.identity_matrix()
sage: m = MatrixMorphism(T, I); m
Morphism defined by the matrix
[1 0 0]
[0 1 0]
[0 0 1]
sage: m.charpoly('x')                                                    #␣
↪needs sage.libs.pari
x^3 - 3*x^2 + 3*x - 1
sage: m.base_ring()
Rational Field
sage: m.det()
1
sage: m.fcp('x')                                                         #␣
↪needs sage.libs.pari
(x - 1)^3
sage: m.matrix()
[1 0 0]
[0 1 0]
[0 0 1]
sage: m.rank()
3
sage: m.trace()
3
```

```
>>> from sage.all import *
>>> from sage.modules.matrix_morphism import MatrixMorphism, is_MatrixMorphism
>>> V = QQ**Integer(3)
>>> T = End(V)
>>> M = MatrixSpace(QQ,Integer(3))
>>> I = M.identity_matrix()
>>> m = MatrixMorphism(T, I); m
Morphism defined by the matrix
[1 0 0]
[0 1 0]
[0 0 1]
>>> m.charpoly('x')                                                      #␣
↪needs sage.libs.pari
x^3 - 3*x^2 + 3*x - 1
>>> m.base_ring()
Rational Field
```

(continues on next page)

(continued from previous page)

```
>>> m.det()
1
>>> m.fcp('x')                                                          #␣
↪needs sage.libs.pari
(x - 1)^3
>>> m.matrix()
[1 0 0]
[0 1 0]
[0 0 1]
>>> m.rank()
3
>>> m.trace()
3
```

AUTHOR:

- William Stein: initial versions
- David Joyner (2005-12-17): added examples
- William Stein (2005-01-07): added __reduce__
- Craig Citro (2008-03-18): refactored MatrixMorphism class
- Rob Beezer (2011-07-15): additional methods, bug fixes, documentation

class sage.modules.matrix_morphism.**MatrixMorphism**(*parent*, *A*, *copy_matrix=True*, *side='left'*)

Bases: *MatrixMorphism_abstract*

A morphism defined by a matrix.

INPUT:

- `parent` – a homspace
- `A` – matrix or a *MatrixMorphism_abstract* instance
- `copy_matrix` – boolean (default: `True`); make an immutable copy of the matrix `A` if it is mutable. If `False`, then this makes `A` immutable.

is_injective()

Tell whether `self` is injective.

EXAMPLES:

```
sage: V1 = QQ^2
sage: V2 = QQ^3
sage: phi = V1.hom(Matrix([[1,2,3], [4,5,6]]),V2)
sage: phi.is_injective()
True
sage: psi = V2.hom(Matrix([[1,2], [3,4], [5,6]]),V1)
sage: psi.is_injective()
False
```

```
>>> from sage.all import *
>>> V1 = QQ**Integer(2)
>>> V2 = QQ**Integer(3)
>>> phi = V1.hom(Matrix([[Integer(1),Integer(2),Integer(3)], [Integer(4),
```

(continues on next page)

(continued from previous page)

```
↪Integer(5),Integer(6)]]),V2)
>>> phi.is_injective()
True
>>> psi = V2.hom(Matrix([[Integer(1),Integer(2)], [Integer(3),Integer(4)],␣
↪[Integer(5),Integer(6)]]),V1)
>>> psi.is_injective()
False
```

AUTHOR:

– Simon King (2010-05)

is_surjective()

Tell whether `self` is surjective.

EXAMPLES:

```
sage: V1 = QQ^2
sage: V2 = QQ^3
sage: phi = V1.hom(Matrix([[1,2,3], [4,5,6]]), V2)
sage: phi.is_surjective()
False
sage: psi = V2.hom(Matrix([[1,2], [3,4], [5,6]]), V1)
sage: psi.is_surjective()
True
```

```
>>> from sage.all import *
>>> V1 = QQ**Integer(2)
>>> V2 = QQ**Integer(3)
>>> phi = V1.hom(Matrix([[Integer(1),Integer(2),Integer(3)], [Integer(4),
↪Integer(5),Integer(6)]]), V2)
>>> phi.is_surjective()
False
>>> psi = V2.hom(Matrix([[Integer(1),Integer(2)], [Integer(3),Integer(4)],␣
↪[Integer(5),Integer(6)]]), V1)
>>> psi.is_surjective()
True
```

An example over a PID that is not **Z**.

```
sage: R.<x> = PolynomialRing(QQ)
sage: A = R^2
sage: B = R^2
sage: H = A.hom([B([x^2 - 1, 1]), B([x^2, 1])])
sage: H.image()
Free module of degree 2 and rank 2 over Univariate Polynomial Ring in x over␣
↪Rational Field
Echelon basis matrix:
[ 1  0]
[ 0 -1]
sage: H.is_surjective()
True
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ, names=('x',)); (x,) = R._first_ngens(1)
>>> A = R**Integer(2)
>>> B = R**Integer(2)
>>> H = A.hom([B([x**Integer(2) - Integer(1), Integer(1)]), B([x**Integer(2),␣
↪Integer(1)])])
>>> H.image()
Free module of degree 2 and rank 2 over Univariate Polynomial Ring in x over␣
↪Rational Field
Echelon basis matrix:
[ 1  0]
[ 0 -1]
>>> H.is_surjective()
True
```

This tests if Issue #11552 is fixed.

```
sage: V = ZZ^2
sage: m = matrix(ZZ, [[1,2], [0,2]])
sage: phi = V.hom(m, V)
sage: phi.lift(vector(ZZ, [0, 1]))
Traceback (most recent call last):
...
ValueError: element is not in the image
sage: phi.is_surjective()
False
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(2)
>>> m = matrix(ZZ, [[Integer(1),Integer(2)], [Integer(0),Integer(2)]])
>>> phi = V.hom(m, V)
>>> phi.lift(vector(ZZ, [Integer(0), Integer(1)]))
Traceback (most recent call last):
...
ValueError: element is not in the image
>>> phi.is_surjective()
False
```

AUTHORS:

- Simon King (2010-05)
- Rob Beezer (2011-06-28)

matrix(*side=None*)

Return a matrix that defines this morphism.

INPUT:

- `side` – (default: `None`) the side of the matrix where a vector is placed to effect the morphism (function)

OUTPUT:

A matrix which represents the morphism, relative to bases for the domain and codomain. If the modules are provided with user bases, then the representation is relative to these bases.

Internally, Sage represents a matrix morphism with the matrix multiplying a row vector placed to the left of the matrix. If the option `side='right'` is used, then a matrix is returned that acts on a vector to the right of the matrix. These two matrices are just transposes of each other and the difference is just a preference for the style of representation.

EXAMPLES:

```
sage: V = ZZ^2; W = ZZ^3
sage: m = column_matrix([3*V.0 - 5*V.1, 4*V.0 + 2*V.1, V.0 + V.1])
sage: phi = V.hom(m, W)
sage: phi.matrix()
[ 3  4  1]
[-5  2  1]

sage: phi.matrix(side='right')
[ 3 -5]
[ 4  2]
[ 1  1]
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(2); W = ZZ**Integer(3)
>>> m = column_matrix([Integer(3)*V.gen(0) - Integer(5)*V.gen(1),␣
↪Integer(4)*V.gen(0) + Integer(2)*V.gen(1), V.gen(0) + V.gen(1)])
>>> phi = V.hom(m, W)
>>> phi.matrix()
[ 3  4  1]
[-5  2  1]

>>> phi.matrix(side='right')
[ 3 -5]
[ 4  2]
[ 1  1]
```

class sage.modules.matrix_morphism.**MatrixMorphism_abstract**(*parent*, *side='left'*)

Bases: `Morphism`

INPUT:

- `parent` – a homspace
- `A` – matrix

EXAMPLES:

```
sage: from sage.modules.matrix_morphism import MatrixMorphism
sage: T = End(ZZ^3)
sage: M = MatrixSpace(ZZ,3)
sage: I = M.identity_matrix()
sage: A = MatrixMorphism(T, I)
sage: loads(A.dumps()) == A
True
```

```
>>> from sage.all import *
>>> from sage.modules.matrix_morphism import MatrixMorphism
>>> T = End(ZZ**Integer(3))
```

(continues on next page)

(continued from previous page)

```
>>> M = MatrixSpace(ZZ,Integer(3))
>>> I = M.identity_matrix()
>>> A = MatrixMorphism(T, I)
>>> loads(A.dumps()) == A
True
```

base_ring()

Return the base ring of `self`, that is, the ring over which `self` is given by a matrix.

EXAMPLES:

```
sage: sage.modules.matrix_morphism.MatrixMorphism((ZZ**2).endomorphism_ring(),
↪ Matrix(ZZ,2,[3..6])).base_ring()
Integer Ring
```

```
>>> from sage.all import *
>>> sage.modules.matrix_morphism.MatrixMorphism((ZZ**Integer(2)).endomorphism_
↪ring(), Matrix(ZZ,Integer(2),(ellipsis_range(Integer(3),Ellipsis,
↪Integer(6))))).base_ring()
Integer Ring
```

characteristic_polynomial()

Return the characteristic polynomial of this endomorphism.

characteristic_polynomial() and *charpoly()* are the same method.

INPUT:

- `var` – variable

EXAMPLES:

```
sage: # needs sage.modules
sage: V = ZZ^2; phi = V.hom([V.0 + V.1, 2*V.1])
sage: phi.characteristic_polynomial()
x^2 - 3*x + 2
sage: phi.charpoly()
x^2 - 3*x + 2
sage: phi.matrix().charpoly()
x^2 - 3*x + 2
sage: phi.charpoly('T')
T^2 - 3*T + 2
sage: W = CombinatorialFreeModule(ZZ, ['x', 'y'])
sage: M = matrix(ZZ, [[1, 0], [1, 2]])
sage: psi = W.module_morphism(matrix=M, codomain=W)
sage: psi.charpoly()
x^2 - 3*x + 2
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> V = ZZ**Integer(2); phi = V.hom([V.gen(0) + V.gen(1), Integer(2)*V.
↪gen(1)])
>>> phi.characteristic_polynomial()
x^2 - 3*x + 2
```

(continues on next page)

(continued from previous page)

```
>>> phi.charpoly()
x^2 - 3*x + 2
>>> phi.matrix().charpoly()
x^2 - 3*x + 2
>>> phi.charpoly('T')
T^2 - 3*T + 2
>>> W = CombinatorialFreeModule(ZZ, ['x', 'y'])
>>> M = matrix(ZZ, [[Integer(1), Integer(0)], [Integer(1), Integer(2)]])
>>> psi = W.module_morphism(matrix=M, codomain=W)
>>> psi.charpoly()
x^2 - 3*x + 2
```

charpoly()

Return the characteristic polynomial of this endomorphism.

characteristic_polynomial() and *charpoly()* are the same method.

INPUT:

- var – variable

EXAMPLES:

```
sage: # needs sage.modules
sage: V = ZZ^2; phi = V.hom([V.0 + V.1, 2*V.1])
sage: phi.characteristic_polynomial()
x^2 - 3*x + 2
sage: phi.charpoly()
x^2 - 3*x + 2
sage: phi.matrix().charpoly()
x^2 - 3*x + 2
sage: phi.charpoly('T')
T^2 - 3*T + 2
sage: W = CombinatorialFreeModule(ZZ, ['x', 'y'])
sage: M = matrix(ZZ, [[1, 0], [1, 2]])
sage: psi = W.module_morphism(matrix=M, codomain=W)
sage: psi.charpoly()
x^2 - 3*x + 2
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> V = ZZ**Integer(2); phi = V.hom([V.gen(0) + V.gen(1), Integer(2)*V.
↪gen(1)])
>>> phi.characteristic_polynomial()
x^2 - 3*x + 2
>>> phi.charpoly()
x^2 - 3*x + 2
>>> phi.matrix().charpoly()
x^2 - 3*x + 2
>>> phi.charpoly('T')
T^2 - 3*T + 2
>>> W = CombinatorialFreeModule(ZZ, ['x', 'y'])
>>> M = matrix(ZZ, [[Integer(1), Integer(0)], [Integer(1), Integer(2)]])
>>> psi = W.module_morphism(matrix=M, codomain=W)
```

(continues on next page)

(continued from previous page)

```
>>> psi.charpoly()
x^2 - 3*x + 2
```

decomposition(**args*, ***kwds*)

Return decomposition of this endomorphism, i.e., sequence of subspaces obtained by finding invariant subspaces of `self`.

See the documentation for `self.matrix().decomposition` for more details. All inputs to this function are passed onto the matrix one.

EXAMPLES:

```
sage: V = ZZ^2; phi = V.hom([V.0+V.1, 2*V.1])
sage: phi.decomposition()                                                   #␣
↪needs sage.libs.pari
[Free module of degree 2 and rank 1 over Integer Ring
 Echelon basis matrix:
 [0 1],
 Free module of degree 2 and rank 1 over Integer Ring
 Echelon basis matrix:
 [ 1 -1]]
sage: phi2 = V.hom(phi.matrix(), side='right')
sage: phi2.decomposition()                                                  #␣
↪needs sage.libs.pari
[Free module of degree 2 and rank 1 over Integer Ring
 Echelon basis matrix:
 [1 1],
 Free module of degree 2 and rank 1 over Integer Ring
 Echelon basis matrix:
 [1 0]]
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(2); phi = V.hom([V.gen(0)+V.gen(1), Integer(2)*V.gen(1)])
>>> phi.decomposition()                                                     #␣
↪needs sage.libs.pari
[Free module of degree 2 and rank 1 over Integer Ring
 Echelon basis matrix:
 [0 1],
 Free module of degree 2 and rank 1 over Integer Ring
 Echelon basis matrix:
 [ 1 -1]]
>>> phi2 = V.hom(phi.matrix(), side='right')
>>> phi2.decomposition()                                                    #␣
↪needs sage.libs.pari
[Free module of degree 2 and rank 1 over Integer Ring
 Echelon basis matrix:
 [1 1],
 Free module of degree 2 and rank 1 over Integer Ring
 Echelon basis matrix:
 [1 0]]
```

det()

Return the determinant of this endomorphism.

determinant() and *det()* are the same method.

EXAMPLES:

```
sage: # needs sage.modules
sage: V = ZZ^2; phi = V.hom([V.0 + V.1, 2*V.1])
sage: phi.determinant()
2
sage: phi.det()
2
sage: W = CombinatorialFreeModule(ZZ, ['x', 'y'])
sage: M = matrix(ZZ, [[1, 0], [1, 2]])
sage: psi = W.module_morphism(matrix=M, codomain=W)
sage: psi.det()
2
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> V = ZZ**Integer(2); phi = V.hom([V.gen(0) + V.gen(1), Integer(2)*V.
↪gen(1)])
>>> phi.determinant()
2
>>> phi.det()
2
>>> W = CombinatorialFreeModule(ZZ, ['x', 'y'])
>>> M = matrix(ZZ, [[Integer(1), Integer(0)], [Integer(1), Integer(2)]])
>>> psi = W.module_morphism(matrix=M, codomain=W)
>>> psi.det()
2
```

determinant()

Return the determinant of this endomorphism.

determinant() and *det()* are the same method.

EXAMPLES:

```
sage: # needs sage.modules
sage: V = ZZ^2; phi = V.hom([V.0 + V.1, 2*V.1])
sage: phi.determinant()
2
sage: phi.det()
2
sage: W = CombinatorialFreeModule(ZZ, ['x', 'y'])
sage: M = matrix(ZZ, [[1, 0], [1, 2]])
sage: psi = W.module_morphism(matrix=M, codomain=W)
sage: psi.det()
2
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> V = ZZ**Integer(2); phi = V.hom([V.gen(0) + V.gen(1), Integer(2)*V.
↪gen(1)])
>>> phi.determinant()
```

(continues on next page)

(continued from previous page)

```
2
>>> phi.det()
2
>>> W = CombinatorialFreeModule(ZZ, ['x', 'y'])
>>> M = matrix(ZZ, [[Integer(1), Integer(0)], [Integer(1), Integer(2)]])
>>> psi = W.module_morphism(matrix=M, codomain=W)
>>> psi.det()
2
```

fcp()

Return the factorization of the characteristic polynomial.

INPUT:

- var – variable

EXAMPLES:

```
sage: # needs sage.modules
sage: V = ZZ^2; phi = V.hom([V.0 + V.1, 2*V.1])
sage: phi.fcp()                                                             #␣
↪needs sage.libs.pari
(x - 2) * (x - 1)
sage: phi.fcp('T')                                                          #␣
↪needs sage.libs.pari
(T - 2) * (T - 1)
sage: W = CombinatorialFreeModule(ZZ, ['x', 'y'])
sage: M = matrix(ZZ, [[1, 0], [1, 2]])
sage: psi = W.module_morphism(matrix=M, codomain=W)
sage: psi.fcp()                                                             #␣
↪needs sage.libs.pari
(x - 2) * (x - 1)
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> V = ZZ**Integer(2); phi = V.hom([V.gen(0) + V.gen(1), Integer(2)*V.
↪gen(1)])
>>> phi.fcp()                                                               # needs␣
↪sage.libs.pari
(x - 2) * (x - 1)
>>> phi.fcp('T')                                                            # needs␣
↪sage.libs.pari
(T - 2) * (T - 1)
>>> W = CombinatorialFreeModule(ZZ, ['x', 'y'])
>>> M = matrix(ZZ, [[Integer(1), Integer(0)], [Integer(1), Integer(2)]])
>>> psi = W.module_morphism(matrix=M, codomain=W)
>>> psi.fcp()                                                               # needs␣
↪sage.libs.pari
(x - 2) * (x - 1)
```

image()

Compute the image of this morphism.

EXAMPLES:

```
sage: V = VectorSpace(QQ, 3)
sage: phi = V.Hom(V)(matrix(QQ, 3, range(9)))
sage: phi.image()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
sage: hom(GF(7)^3, GF(7)^2, zero_matrix(GF(7), 3, 2)).image()
Vector space of degree 2 and dimension 0 over Finite Field of size 7
Basis matrix:
[]
sage: m = matrix(3, [1, 0, 0, 1, 0, 0, 0, 0, 1]); m
[1 0 0]
[1 0 0]
[0 0 1]
sage: f1 = V.hom(m)
sage: f2 = V.hom(m, side='right')
sage: f1.image()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[1 0 0]
[0 0 1]
sage: f2.image()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[1 1 0]
[0 0 1]
```

```
>>> from sage.all import *
>>> V = VectorSpace(QQ, Integer(3))
>>> phi = V.Hom(V)(matrix(QQ, Integer(3), range(Integer(9))))
>>> phi.image()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
>>> hom(GF(Integer(7))**Integer(3), GF(Integer(7))**Integer(2), zero_
↪matrix(GF(Integer(7)), Integer(3), Integer(2))).image()
Vector space of degree 2 and dimension 0 over Finite Field of size 7
Basis matrix:
[]
>>> m = matrix(Integer(3), [Integer(1), Integer(0), Integer(0), Integer(1),␣
↪Integer(0), Integer(0), Integer(0), Integer(0), Integer(1)]); m
[1 0 0]
[1 0 0]
[0 0 1]
>>> f1 = V.hom(m)
>>> f2 = V.hom(m, side='right')
>>> f1.image()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[1 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 0 1]
>>> f2.image()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[1 1 0]
[0 0 1]
```

Compute the image of the identity map on a ZZ-submodule:

```
sage: V = (ZZ^2).span([[1,2], [3,4]])
sage: phi = V.Hom(V)(identity_matrix(ZZ,2))
sage: phi(V.0) == V.0
True
sage: phi(V.1) == V.1
True
sage: phi.image()
Free module of degree 2 and rank 2 over Integer Ring
Echelon basis matrix:
[1 0]
[0 2]
sage: phi.image() == V
True
```

```
>>> from sage.all import *
>>> V = (ZZ**Integer(2)).span([[Integer(1),Integer(2)], [Integer(3),
↪Integer(4)]])
>>> phi = V.Hom(V)(identity_matrix(ZZ,Integer(2)))
>>> phi(V.gen(0)) == V.gen(0)
True
>>> phi(V.gen(1)) == V.gen(1)
True
>>> phi.image()
Free module of degree 2 and rank 2 over Integer Ring
Echelon basis matrix:
[1 0]
[0 2]
>>> phi.image() == V
True
```

inverse()

Return the inverse of this matrix morphism, if the inverse exists.

This raises a `ZeroDivisionError` if the inverse does not exist.

EXAMPLES:

An invertible morphism created as a restriction of a non-invertible morphism, and which has an unequal domain and codomain.

```
sage: V = QQ^4
sage: W = QQ^3
sage: m = matrix(QQ, [[2, 0, 3], [-6, 1, 4], [1, 2, -4], [1, 0, 1]])
sage: phi = V.hom(m, W)
```

(continues on next page)

(continued from previous page)

```
sage: rho = phi.restrict_domain(V.span([V.0, V.3]))
sage: zeta = rho.restrict_codomain(W.span([W.0, W.2]))
sage: x = vector(QQ, [2, 0, 0, -7])
sage: y = zeta(x); y
(-3, 0, -1)
sage: inv = zeta.inverse(); inv
Vector space morphism represented by the matrix:
[-1  3]
[ 1 -2]
Domain: Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[1 0 0]
[0 0 1]
Codomain: Vector space of degree 4 and dimension 2 over Rational Field
Basis matrix:
[1 0 0 0]
[0 0 0 1]
sage: inv(y) == x
True
```

```
>>> from sage.all import *
>>> V = QQ**Integer(4)
>>> W = QQ**Integer(3)
>>> m = matrix(QQ, [[Integer(2), Integer(0), Integer(3)], [-Integer(6),␣
↪Integer(1), Integer(4)], [Integer(1), Integer(2), -Integer(4)], [Integer(1),
↪ Integer(0), Integer(1)]])
>>> phi = V.hom(m, W)
>>> rho = phi.restrict_domain(V.span([V.gen(0), V.gen(3)]))
>>> zeta = rho.restrict_codomain(W.span([W.gen(0), W.gen(2)]))
>>> x = vector(QQ, [Integer(2), Integer(0), Integer(0), -Integer(7)])
>>> y = zeta(x); y
(-3, 0, -1)
>>> inv = zeta.inverse(); inv
Vector space morphism represented by the matrix:
[-1  3]
[ 1 -2]
Domain: Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[1 0 0]
[0 0 1]
Codomain: Vector space of degree 4 and dimension 2 over Rational Field
Basis matrix:
[1 0 0 0]
[0 0 0 1]
>>> inv(y) == x
True
```

An example of an invertible morphism between modules, (rather than between vector spaces).

```
sage: M = ZZ^4
sage: p = matrix(ZZ, [[ 0, -1,  1, -2],
....:                 [ 1, -3,  2, -3],
```

(continues on next page)

(continued from previous page)

```
....:                 [ 0,  4, -3,  4],
....:                 [-2,  8, -4,  3]])
sage: phi = M.hom(p, M)
sage: x = vector(ZZ, [1, -3, 5, -2])
sage: y = phi(x); y
(1, 12, -12, 21)
sage: rho = phi.inverse(); rho
Free module morphism defined by the matrix
[ -5   3  -1   1]
[ -9   4  -3   2]
[-20   8  -7   4]
[ -6   2  -2   1]
Domain: Ambient free module of rank 4 over the principal ideal domain ...
Codomain: Ambient free module of rank 4 over the principal ideal domain ...
sage: rho(y) == x
True
```

```
>>> from sage.all import *
>>> M = ZZ**Integer(4)
>>> p = matrix(ZZ, [[ Integer(0), -Integer(1),  Integer(1), -Integer(2)],
...                 [ Integer(1), -Integer(3),  Integer(2), -Integer(3)],
...                 [ Integer(0),  Integer(4), -Integer(3),  Integer(4)],
...                 [-Integer(2),  Integer(8), -Integer(4),  Integer(3)]])
>>> phi = M.hom(p, M)
>>> x = vector(ZZ, [Integer(1), -Integer(3), Integer(5), -Integer(2)])
>>> y = phi(x); y
(1, 12, -12, 21)
>>> rho = phi.inverse(); rho
Free module morphism defined by the matrix
[ -5   3  -1   1]
[ -9   4  -3   2]
[-20   8  -7   4]
[ -6   2  -2   1]
Domain: Ambient free module of rank 4 over the principal ideal domain ...
Codomain: Ambient free module of rank 4 over the principal ideal domain ...
>>> rho(y) == x
True
```

A non-invertible morphism, despite having an appropriate domain and codomain.

```
sage: V = QQ^2
sage: m = matrix(QQ, [[1, 2], [20, 40]])
sage: phi = V.hom(m, V)
sage: phi.is_bijective()
False
sage: phi.inverse()
Traceback (most recent call last):
...
ZeroDivisionError: matrix morphism not invertible
```

```
>>> from sage.all import *
>>> V = QQ**Integer(2)
```

(continues on next page)

(continued from previous page)

```
>>> m = matrix(QQ, [[Integer(1), Integer(2)], [Integer(20), Integer(40)]])
>>> phi = V.hom(m, V)
>>> phi.is_bijective()
False
>>> phi.inverse()
Traceback (most recent call last):
...
ZeroDivisionError: matrix morphism not invertible
```

The matrix representation of this morphism is invertible over the rationals, but not over the integers, thus the morphism is not invertible as a map between modules. It is easy to notice from the definition that every vector of the image will have a second entry that is an even integer.

```
sage: V = ZZ^2
sage: q = matrix(ZZ, [[1, 2], [3, 4]])
sage: phi = V.hom(q, V)
sage: phi.matrix().change_ring(QQ).inverse()
[  -2    1]
[ 3/2 -1/2]
sage: phi.is_bijective()
False
sage: phi.image()
Free module of degree 2 and rank 2 over Integer Ring
Echelon basis matrix:
[1 0]
[0 2]
sage: phi.lift(vector(ZZ, [1, 1]))
Traceback (most recent call last):
...
ValueError: element is not in the image
sage: phi.inverse()
Traceback (most recent call last):
...
ZeroDivisionError: matrix morphism not invertible
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(2)
>>> q = matrix(ZZ, [[Integer(1), Integer(2)], [Integer(3), Integer(4)]])
>>> phi = V.hom(q, V)
>>> phi.matrix().change_ring(QQ).inverse()
[  -2    1]
[ 3/2 -1/2]
>>> phi.is_bijective()
False
>>> phi.image()
Free module of degree 2 and rank 2 over Integer Ring
Echelon basis matrix:
[1 0]
[0 2]
>>> phi.lift(vector(ZZ, [Integer(1), Integer(1)]))
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
ValueError: element is not in the image
>>> phi.inverse()
Traceback (most recent call last):
...
ZeroDivisionError: matrix morphism not invertible
```

The unary invert operator (~, tilde, "wiggle") is synonymous with the `inverse()` method (and a lot easier to type).

```
sage: V = QQ^2
sage: r = matrix(QQ, [[4, 3], [-2, 5]])
sage: phi = V.hom(r, V)
sage: rho = phi.inverse()
sage: zeta = ~phi
sage: rho.is_equal_function(zeta)
True
```

```
>>> from sage.all import *
>>> V = QQ**Integer(2)
>>> r = matrix(QQ, [[Integer(4), Integer(3)], [-Integer(2), Integer(5)]])
>>> phi = V.hom(r, V)
>>> rho = phi.inverse()
>>> zeta = ~phi
>>> rho.is_equal_function(zeta)
True
```

is_bijective()

Tell whether `self` is bijective.

EXAMPLES:

Two morphisms that are obviously not bijective, simply on considerations of the dimensions. However, each fullfills half of the requirements to be a bijection.

```
sage: V1 = QQ^2
sage: V2 = QQ^3
sage: m = matrix(QQ, [[1, 2, 3], [4, 5, 6]])
sage: phi = V1.hom(m, V2)
sage: phi.is_injective()
True
sage: phi.is_bijective()
False
sage: rho = V2.hom(m.transpose(), V1)
sage: rho.is_surjective()
True
sage: rho.is_bijective()
False
```

```
>>> from sage.all import *
>>> V1 = QQ**Integer(2)
>>> V2 = QQ**Integer(3)
>>> m = matrix(QQ, [[Integer(1), Integer(2), Integer(3)], [Integer(4),␣
↪Integer(5), Integer(6)]])
```

(continues on next page)

(continued from previous page)

```
>>> phi = V1.hom(m, V2)
>>> phi.is_injective()
True
>>> phi.is_bijective()
False
>>> rho = V2.hom(m.transpose(), V1)
>>> rho.is_surjective()
True
>>> rho.is_bijective()
False
```

We construct a simple bijection between two one-dimensional vector spaces.

```
sage: V1 = QQ^3
sage: V2 = QQ^2
sage: phi = V1.hom(matrix(QQ, [[1, 2], [3, 4], [5, 6]]), V2)
sage: x = vector(QQ, [1, -1, 4])
sage: y = phi(x); y
(18, 22)
sage: rho = phi.restrict_domain(V1.span([x]))
sage: zeta = rho.restrict_codomain(V2.span([y]))
sage: zeta.is_bijective()
True
```

```
>>> from sage.all import *
>>> V1 = QQ**Integer(3)
>>> V2 = QQ**Integer(2)
>>> phi = V1.hom(matrix(QQ, [[Integer(1), Integer(2)], [Integer(3),␣
↪Integer(4)], [Integer(5), Integer(6)]]), V2)
>>> x = vector(QQ, [Integer(1), -Integer(1), Integer(4)])
>>> y = phi(x); y
(18, 22)
>>> rho = phi.restrict_domain(V1.span([x]))
>>> zeta = rho.restrict_codomain(V2.span([y]))
>>> zeta.is_bijective()
True
```

AUTHOR:

- Rob Beezer (2011-06-28)

is_equal_function(*other*)

Determines if two morphisms are equal functions.

INPUT:

- `other` – a morphism to compare with `self`

OUTPUT:

Returns `True` precisely when the two morphisms have equal domains and codomains (as sets) and produce identical output when given the same input. Otherwise returns `False`.

This is useful when `self` and `other` may have different representations.

Sage's default comparison of matrix morphisms requires the domains to have the same bases and the

codomains to have the same bases, and then compares the matrix representations. This notion of equality is more permissive (it will return `True` "more often"), but is more correct mathematically.

EXAMPLES:

Three morphisms defined by combinations of different bases for the domain and codomain and different functions. Two are equal, the third is different from both of the others.

```
sage: B = matrix(QQ, [[-3,  5, -4,  2],
....:                 [-1,  2, -1,  4],
....:                 [ 4, -6,  5, -1],
....:                 [-5,  7, -6,  1]])
sage: U = (QQ^4).subspace_with_basis(B.rows())
sage: C = matrix(QQ, [[-1, -6, -4],
....:                 [ 3, -5,  6],
....:                 [ 1,  2,  3]])
sage: V = (QQ^3).subspace_with_basis(C.rows())
sage: H = Hom(U, V)

sage: D = matrix(QQ, [[-7, -2, -5,  2],
....:                 [-5,  1, -4, -8],
....:                 [ 1, -1,  1,  4],
....:                 [-4, -1, -3,  1]])
sage: X = (QQ^4).subspace_with_basis(D.rows())
sage: E = matrix(QQ, [[ 4, -1,  4],
....:                 [ 5, -4, -5],
....:                 [-1,  0, -2]])
sage: Y = (QQ^3).subspace_with_basis(E.rows())
sage: K = Hom(X, Y)

sage: f = lambda x: vector(QQ, [x[0]+x[1], 2*x[1]-4*x[2], 5*x[3]])
sage: g = lambda x: vector(QQ, [x[0]-x[2], 2*x[1]-4*x[2], 5*x[3]])

sage: rho = H(f)
sage: phi = K(f)
sage: zeta = H(g)

sage: rho.is_equal_function(phi)
True
sage: phi.is_equal_function(rho)
True
sage: zeta.is_equal_function(rho)
False
sage: phi.is_equal_function(zeta)
False
```

```
>>> from sage.all import *
>>> B = matrix(QQ, [[-Integer(3),  Integer(5), -Integer(4),  Integer(2)],
...                 [-Integer(1),  Integer(2), -Integer(1),  Integer(4)],
...                 [ Integer(4), -Integer(6),  Integer(5), -Integer(1)],
...                 [-Integer(5),  Integer(7), -Integer(6),  Integer(1)]])
>>> U = (QQ**Integer(4)).subspace_with_basis(B.rows())
>>> C = matrix(QQ, [[-Integer(1), -Integer(6), -Integer(4)],
...                 [ Integer(3), -Integer(5),  Integer(6)],
```

(continues on next page)

(continued from previous page)

```
...                 [ Integer(1),  Integer(2),  Integer(3)]])
>>> V = (QQ**Integer(3)).subspace_with_basis(C.rows())
>>> H = Hom(U, V)

>>> D = matrix(QQ, [[-Integer(7), -Integer(2), -Integer(5),  Integer(2)],
...                 [-Integer(5),  Integer(1), -Integer(4), -Integer(8)],
...                 [ Integer(1), -Integer(1),  Integer(1),  Integer(4)],
...                 [-Integer(4), -Integer(1), -Integer(3),   Integer(1)]])
>>> X = (QQ**Integer(4)).subspace_with_basis(D.rows())
>>> E = matrix(QQ, [[ Integer(4), -Integer(1),  Integer(4)],
...                 [ Integer(5), -Integer(4), -Integer(5)],
...                 [-Integer(1),  Integer(0), -Integer(2)]])
>>> Y = (QQ**Integer(3)).subspace_with_basis(E.rows())
>>> K = Hom(X, Y)

>>> f = lambda x: vector(QQ, [x[Integer(0)]+x[Integer(1)],␣
↪Integer(2)*x[Integer(1)]-Integer(4)*x[Integer(2)],␣
↪Integer(5)*x[Integer(3)]])
>>> g = lambda x: vector(QQ, [x[Integer(0)]-x[Integer(2)],␣
↪Integer(2)*x[Integer(1)]-Integer(4)*x[Integer(2)],␣
↪Integer(5)*x[Integer(3)]])

>>> rho = H(f)
>>> phi = K(f)
>>> zeta = H(g)

>>> rho.is_equal_function(phi)
True
>>> phi.is_equal_function(rho)
True
>>> zeta.is_equal_function(rho)
False
>>> phi.is_equal_function(zeta)
False
```

AUTHOR:

- Rob Beezer (2011-07-15)

is_identity()

Determines if this morphism is an identity function or not.

EXAMPLES:

A homomorphism that cannot possibly be the identity due to an unequal domain and codomain.

```
sage: V = QQ^3
sage: W = QQ^2
sage: m = matrix(QQ, [[1, 2], [3, 4], [5, 6]])
sage: phi = V.hom(m, W)
sage: phi.is_identity()
False
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3)
>>> W = QQ**Integer(2)
>>> m = matrix(QQ, [[Integer(1), Integer(2)], [Integer(3), Integer(4)],␣
↪[Integer(5), Integer(6)]])
>>> phi = V.hom(m, W)
>>> phi.is_identity()
False
```

A bijection, but not the identity.

```
sage: V = QQ^3
sage: n = matrix(QQ, [[3, 1, -8], [5, -4, 6], [1, 1, -5]])
sage: phi = V.hom(n, V)
sage: phi.is_bijective()
True
sage: phi.is_identity()
False
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3)
>>> n = matrix(QQ, [[Integer(3), Integer(1), -Integer(8)], [Integer(5), -
↪Integer(4), Integer(6)], [Integer(1), Integer(1), -Integer(5)]])
>>> phi = V.hom(n, V)
>>> phi.is_bijective()
True
>>> phi.is_identity()
False
```

A restriction that is the identity.

```
sage: V = QQ^3
sage: p = matrix(QQ, [[1, 0, 0], [5, 8, 3], [0, 0, 1]])
sage: phi = V.hom(p, V)
sage: rho = phi.restrict(V.span([V.0, V.2]))
sage: rho.is_identity()
True
```

```
>>> from sage.all import *
>>> V = QQ**Integer(3)
>>> p = matrix(QQ, [[Integer(1), Integer(0), Integer(0)], [Integer(5),␣
↪Integer(8), Integer(3)], [Integer(0), Integer(0), Integer(1)]])
>>> phi = V.hom(p, V)
>>> rho = phi.restrict(V.span([V.gen(0), V.gen(2)]))
>>> rho.is_identity()
True
```

An identity linear transformation that is defined with a domain and codomain with wildly different bases, so that the matrix representation is not simply the identity matrix.

```
sage: A = matrix(QQ, [[1, 1, 0], [2, 3, -4], [2, 4, -7]])
sage: B = matrix(QQ, [[2, 7, -2], [-1, -3, 1], [-1, -6, 2]])
sage: U = (QQ^3).subspace_with_basis(A.rows())
```

(continues on next page)

(continued from previous page)

```
sage: V = (QQ^3).subspace_with_basis(B.rows())
sage: H = Hom(U, V)
sage: id = lambda x: x
sage: phi = H(id)
sage: phi([203, -179, 34])
(203, -179, 34)
sage: phi.matrix()
[  1   0   1]
[ -9 -18  -2]
[-17 -31  -5]
sage: phi.is_identity()
True
```

```
>>> from sage.all import *
>>> A = matrix(QQ, [[Integer(1), Integer(1), Integer(0)], [Integer(2),␣
↪Integer(3), -Integer(4)], [Integer(2), Integer(4), -Integer(7)]])
>>> B = matrix(QQ, [[Integer(2), Integer(7), -Integer(2)], [-Integer(1), -
↪Integer(3), Integer(1)], [-Integer(1), -Integer(6), Integer(2)]])
>>> U = (QQ**Integer(3)).subspace_with_basis(A.rows())
>>> V = (QQ**Integer(3)).subspace_with_basis(B.rows())
>>> H = Hom(U, V)
>>> id = lambda x: x
>>> phi = H(id)
>>> phi([Integer(203), -Integer(179), Integer(34)])
(203, -179, 34)
>>> phi.matrix()
[  1   0   1]
[ -9 -18  -2]
[-17 -31  -5]
>>> phi.is_identity()
True
```

AUTHOR:

- Rob Beezer (2011-06-28)

is_zero()

Determines if this morphism is a zero function or not.

EXAMPLES:

A zero morphism created from a function.

```
sage: V = ZZ^5
sage: W = ZZ^3
sage: z = lambda x: zero_vector(ZZ, 3)
sage: phi = V.hom(z, W)
sage: phi.is_zero()
True
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(5)
>>> W = ZZ**Integer(3)
```

(continues on next page)

(continued from previous page)

```
>>> z = lambda x: zero_vector(ZZ, Integer(3))
>>> phi = V.hom(z, W)
>>> phi.is_zero()
True
```

An image list that just barely makes a nonzero morphism.

```
sage: V = ZZ^4
sage: W = ZZ^6
sage: z = zero_vector(ZZ, 6)
sage: images = [z, z, W.5, z]
sage: phi = V.hom(images, W)
sage: phi.is_zero()
False
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(4)
>>> W = ZZ**Integer(6)
>>> z = zero_vector(ZZ, Integer(6))
>>> images = [z, z, W.gen(5), z]
>>> phi = V.hom(images, W)
>>> phi.is_zero()
False
```

AUTHOR:

- Rob Beezer (2011-07-15)

kernel()

Compute the kernel of this morphism.

EXAMPLES:

```
sage: V = VectorSpace(QQ, 3)
sage: id = V.Hom(V)(identity_matrix(QQ,3))
sage: null = V.Hom(V)(0*identity_matrix(QQ,3))
sage: id.kernel()
Vector space of degree 3 and dimension 0 over Rational Field
Basis matrix:
[]
sage: phi = V.Hom(V)(matrix(QQ, 3, range(9)))
sage: phi.kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -2  1]
sage: hom(CC^2, CC^2, matrix(CC, [[1,0], [0,1]])).kernel()
Vector space of degree 2 and dimension 0 over Complex Field with 53 bits of␣
↪precision
Basis matrix:
[]
sage: m = matrix(3, [1, 0, 0, 1, 0, 0, 0, 0, 1]); m
[1 0 0]
[1 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 0 1]
sage: f1 = V.hom(m)
sage: f2 = V.hom(m, side='right')
sage: f1.kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -1  0]
sage: f2.kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[0 1 0]
```

```
>>> from sage.all import *
>>> V = VectorSpace(QQ, Integer(3))
>>> id = V.Hom(V)(identity_matrix(QQ,Integer(3)))
>>> null = V.Hom(V)(Integer(0)*identity_matrix(QQ,Integer(3)))
>>> id.kernel()
Vector space of degree 3 and dimension 0 over Rational Field
Basis matrix:
[]
>>> phi = V.Hom(V)(matrix(QQ, Integer(3), range(Integer(9))))
>>> phi.kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -2  1]
>>> hom(CC**Integer(2), CC**Integer(2), matrix(CC, [[Integer(1),Integer(0)],␣
↪[Integer(0),Integer(1)]])).kernel()
Vector space of degree 2 and dimension 0 over Complex Field with 53 bits of␣
↪precision
Basis matrix:
[]
>>> m = matrix(Integer(3), [Integer(1), Integer(0), Integer(0), Integer(1),␣
↪Integer(0), Integer(0), Integer(0), Integer(0), Integer(1)]); m
[1 0 0]
[1 0 0]
[0 0 1]
>>> f1 = V.hom(m)
>>> f2 = V.hom(m, side='right')
>>> f1.kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -1  0]
>>> f2.kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[0 1 0]
```

matrix()

EXAMPLES:

```
sage: V = ZZ^2; phi = V.hom(V.basis())
sage: phi.matrix()
```

(continues on next page)

(continued from previous page)

```
[1 0]
[0 1]
sage: sage.modules.matrix_morphism.MatrixMorphism_abstract.matrix(phi)
Traceback (most recent call last):
...
NotImplementedError: this method must be overridden in the extension class
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(2); phi = V.hom(V.basis())
>>> phi.matrix()
[1 0]
[0 1]
>>> sage.modules.matrix_morphism.MatrixMorphism_abstract.matrix(phi)
Traceback (most recent call last):
...
NotImplementedError: this method must be overridden in the extension class
```

nullity()

Return the nullity of the matrix representing this morphism, which is the dimension of its kernel.

EXAMPLES:

```
sage: V = ZZ^2; phi = V.hom(V.basis())
sage: phi.nullity()
0
sage: V = ZZ^2; phi = V.hom([V.0, V.0])
sage: phi.nullity()
1
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(2); phi = V.hom(V.basis())
>>> phi.nullity()
0
>>> V = ZZ**Integer(2); phi = V.hom([V.gen(0), V.gen(0)])
>>> phi.nullity()
1
```

```
sage: m = matrix(2, [1, 2])
sage: V = ZZ^2
sage: h1 = V.hom(m)
sage: h1.nullity()
1
sage: W = ZZ^1
sage: h2 = W.hom(m, side='right')
sage: h2.nullity()
0
```

```
>>> from sage.all import *
>>> m = matrix(Integer(2), [Integer(1), Integer(2)])
>>> V = ZZ**Integer(2)
>>> h1 = V.hom(m)
```

(continues on next page)

(continued from previous page)

```
>>> h1.nullity()
1
>>> W = ZZ**Integer(1)
>>> h2 = W.hom(m, side='right')
>>> h2.nullity()
0
```

rank()

Return the rank of the matrix representing this morphism.

EXAMPLES:

```
sage: V = ZZ^2; phi = V.hom(V.basis())
sage: phi.rank()
2
sage: V = ZZ^2; phi = V.hom([V.0, V.0])
sage: phi.rank()
1
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(2); phi = V.hom(V.basis())
>>> phi.rank()
2
>>> V = ZZ**Integer(2); phi = V.hom([V.gen(0), V.gen(0)])
>>> phi.rank()
1
```

restrict(*sub*)

Restrict this matrix morphism to a subspace sub of the domain.

The codomain and domain of the resulting matrix are both sub.

EXAMPLES:

```
sage: V = ZZ^2; phi = V.hom([3*V.0, 2*V.1])
sage: phi.restrict(V.span([V.0]))
Free module morphism defined by the matrix
[3]
Domain: Free module of degree 2 and rank 1 over Integer Ring
Echelon ...
Codomain: Free module of degree 2 and rank 1 over Integer Ring
Echelon ...

sage: V = (QQ^2).span_of_basis([[1,2], [3,4]])
sage: phi = V.hom([V.0 + V.1, 2*V.1])
sage: phi(V.1) == 2*V.1
True
sage: W = span([V.1])
sage: phi(W)
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[  1 4/3]
sage: psi = phi.restrict(W); psi
```

(continues on next page)

(continued from previous page)

```
Vector space morphism represented by the matrix:
[2]
Domain:   Vector space of degree 2 and dimension 1 over Rational Field
          Basis matrix:
          [  1 4/3]
Codomain: Vector space of degree 2 and dimension 1 over Rational Field
          Basis matrix:
          [  1 4/3]
sage: psi.domain() == W
True
sage: psi(W.0) == 2*W.0
True
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(2); phi = V.hom([Integer(3)*V.gen(0), Integer(2)*V.
↪gen(1)])
>>> phi.restrict(V.span([V.gen(0)]))
Free module morphism defined by the matrix
[3]
Domain: Free module of degree 2 and rank 1 over Integer Ring
Echelon ...
Codomain: Free module of degree 2 and rank 1 over Integer Ring
Echelon ...

>>> V = (QQ**Integer(2)).span_of_basis([[Integer(1),Integer(2)], [Integer(3),
↪Integer(4)]])
>>> phi = V.hom([V.gen(0) + V.gen(1), Integer(2)*V.gen(1)])
>>> phi(V.gen(1)) == Integer(2)*V.gen(1)
True
>>> W = span([V.gen(1)])
>>> phi(W)
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[  1 4/3]
>>> psi = phi.restrict(W); psi
Vector space morphism represented by the matrix:
[2]
Domain:   Vector space of degree 2 and dimension 1 over Rational Field
          Basis matrix:
          [  1 4/3]
Codomain: Vector space of degree 2 and dimension 1 over Rational Field
          Basis matrix:
          [  1 4/3]
>>> psi.domain() == W
True
>>> psi(W.gen(0)) == Integer(2)*W.gen(0)
True
```

```
sage: V = ZZ^3
sage: h1 = V.hom([V.0, V.1+V.2, -V.1+V.2])
sage: h2 = h1.side_switch()
```

(continues on next page)

(continued from previous page)

```
sage: SV = V.span([2*V.1,2*V.2])
sage: h1.restrict(SV)
Free module morphism defined by the matrix
[ 1  1]
[-1  1]
Domain:   Free module of degree 3 and rank 2 over Integer Ring
          Echelon basis matrix:
          [0 2 0]
          [0 0 2]
Codomain: Free module of degree 3 and rank 2 over Integer Ring
          Echelon basis matrix:
          [0 2 0]
          [0 0 2]
sage: h2.restrict(SV)
Free module morphism defined as left-multiplication by the matrix
[ 1 -1]
[ 1  1]
Domain:   Free module of degree 3 and rank 2 over Integer Ring
          Echelon basis matrix:
          [0 2 0]
          [0 0 2]
Codomain: Free module of degree 3 and rank 2 over Integer Ring
          Echelon basis matrix:
          [0 2 0]
          [0 0 2]
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(3)
>>> h1 = V.hom([V.gen(0), V.gen(1)+V.gen(2), -V.gen(1)+V.gen(2)])
>>> h2 = h1.side_switch()
>>> SV = V.span([Integer(2)*V.gen(1),Integer(2)*V.gen(2)])
>>> h1.restrict(SV)
Free module morphism defined by the matrix
[ 1  1]
[-1  1]
Domain:   Free module of degree 3 and rank 2 over Integer Ring
          Echelon basis matrix:
          [0 2 0]
          [0 0 2]
Codomain: Free module of degree 3 and rank 2 over Integer Ring
          Echelon basis matrix:
          [0 2 0]
          [0 0 2]
>>> h2.restrict(SV)
Free module morphism defined as left-multiplication by the matrix
[ 1 -1]
[ 1  1]
Domain:   Free module of degree 3 and rank 2 over Integer Ring
          Echelon basis matrix:
          [0 2 0]
          [0 0 2]
Codomain: Free module of degree 3 and rank 2 over Integer Ring
```

(continues on next page)

(continued from previous page)

```
        Echelon basis matrix:
        [0 2 0]
        [0 0 2]
```

restrict_codomain(*sub*)

Restrict this matrix morphism to a subspace sub of the codomain.

The resulting morphism has the same domain as before, but a new codomain.

EXAMPLES:

```
sage: V = ZZ^2; phi = V.hom([4*(V.0+V.1),0])
sage: W = V.span([2*(V.0+V.1)])
sage: phi
Free module morphism defined by the matrix
[4 4]
[0 0]
Domain: Ambient free module of rank 2 over the principal ideal domain ...
Codomain: Ambient free module of rank 2 over the principal ideal domain ...
sage: psi = phi.restrict_codomain(W); psi
Free module morphism defined by the matrix
[2]
[0]
Domain: Ambient free module of rank 2 over the principal ideal domain ...
Codomain: Free module of degree 2 and rank 1 over Integer Ring
Echelon ...
sage: phi2 = phi.side_switch(); phi2.restrict_codomain(W)
Free module morphism defined as left-multiplication by the matrix
[2 0]
Domain: Ambient free module of rank 2 over the principal ideal domain Integer␣
↪Ring
Codomain: Free module of degree 2 and rank 1 over Integer Ring
Echelon ...
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(2); phi = V.hom([Integer(4)*(V.gen(0)+V.gen(1)),
↪Integer(0)])
>>> W = V.span([Integer(2)*(V.gen(0)+V.gen(1))])
>>> phi
Free module morphism defined by the matrix
[4 4]
[0 0]
Domain: Ambient free module of rank 2 over the principal ideal domain ...
Codomain: Ambient free module of rank 2 over the principal ideal domain ...
>>> psi = phi.restrict_codomain(W); psi
Free module morphism defined by the matrix
[2]
[0]
Domain: Ambient free module of rank 2 over the principal ideal domain ...
Codomain: Free module of degree 2 and rank 1 over Integer Ring
Echelon ...
>>> phi2 = phi.side_switch(); phi2.restrict_codomain(W)
Free module morphism defined as left-multiplication by the matrix
```

(continues on next page)

(continued from previous page)

```
[2 0]
Domain: Ambient free module of rank 2 over the principal ideal domain Integer␣
↪Ring
Codomain: Free module of degree 2 and rank 1 over Integer Ring
Echelon ...
```

An example in which the codomain equals the full ambient space, but with a different basis:

```
sage: V = QQ^2
sage: W = V.span_of_basis([[1,2],[3,4]])
sage: phi = V.hom(matrix(QQ,2,[1,0,2,0]),W)
sage: phi.matrix()
[1 0]
[2 0]
sage: phi(V.0)
(1, 2)
sage: phi(V.1)
(2, 4)
sage: X = V.span([[1,2]]); X
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 2]
sage: phi(V.0) in X
True
sage: phi(V.1) in X
True
sage: psi = phi.restrict_codomain(X); psi
Vector space morphism represented by the matrix:
[1]
[2]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 2]
sage: psi(V.0)
(1, 2)
sage: psi(V.1)
(2, 4)
sage: psi(V.0).parent() is X
True
```

```
>>> from sage.all import *
>>> V = QQ**Integer(2)
>>> W = V.span_of_basis([[Integer(1),Integer(2)],[Integer(3),Integer(4)]])
>>> phi = V.hom(matrix(QQ,Integer(2),[Integer(1),Integer(0),Integer(2),
↪Integer(0)]),W)
>>> phi.matrix()
[1 0]
[2 0]
>>> phi(V.gen(0))
(1, 2)
>>> phi(V.gen(1))
```

(continues on next page)

(continued from previous page)

```
(2, 4)
>>> X = V.span([[Integer(1),Integer(2)]]); X
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 2]
>>> phi(V.gen(0)) in X
True
>>> phi(V.gen(1)) in X
True
>>> psi = phi.restrict_codomain(X); psi
Vector space morphism represented by the matrix:
[1]
[2]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 2]
>>> psi(V.gen(0))
(1, 2)
>>> psi(V.gen(1))
(2, 4)
>>> psi(V.gen(0)).parent() is X
True
```

restrict_domain(*sub*)

Restrict this matrix morphism to a subspace sub of the domain. The subspace sub should have a basis() method and elements of the basis should be coercible into domain.

The resulting morphism has the same codomain as before, but a new domain.

EXAMPLES:

```
sage: V = ZZ^2; phi = V.hom([3*V.0, 2*V.1])
sage: phi.restrict_domain(V.span([V.0]))
Free module morphism defined by the matrix
[3 0]
Domain: Free module of degree 2 and rank 1 over Integer Ring
Echelon ...
Codomain: Ambient free module of rank 2 over the principal ideal domain ...
sage: phi.restrict_domain(V.span([V.1]))
Free module morphism defined by the matrix
[0 2]...
sage: m = matrix(2, range(1,5))
sage: f1 = V.hom(m); f2 = V.hom(m, side='right')
sage: SV = V.span([V.0])
sage: f1.restrict_domain(SV)
Free module morphism defined by the matrix
[1 2]...
sage: f2.restrict_domain(SV)
Free module morphism defined as left-multiplication by the matrix
[1]
[3]...
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(2); phi = V.hom([Integer(3)*V.gen(0), Integer(2)*V.
↪gen(1)])
>>> phi.restrict_domain(V.span([V.gen(0)]))
Free module morphism defined by the matrix
[3 0]
Domain: Free module of degree 2 and rank 1 over Integer Ring
Echelon ...
Codomain: Ambient free module of rank 2 over the principal ideal domain ...
>>> phi.restrict_domain(V.span([V.gen(1)]))
Free module morphism defined by the matrix
[0 2]...
>>> m = matrix(Integer(2), range(Integer(1),Integer(5)))
>>> f1 = V.hom(m); f2 = V.hom(m, side='right')
>>> SV = V.span([V.gen(0)])
>>> f1.restrict_domain(SV)
Free module morphism defined by the matrix
[1 2]...
>>> f2.restrict_domain(SV)
Free module morphism defined as left-multiplication by the matrix
[1]
[3]...
```

side()

Return the side of vectors acted on, relative to the matrix.

EXAMPLES:

```
sage: m = matrix(2, [1, 1, 0, 1])
sage: V = ZZ^2
sage: h1 = V.hom(m); h2 = V.hom(m, side='right')
sage: h1.side()
'left'
sage: h1([1, 0])
(1, 1)
sage: h2.side()
'right'
sage: h2([1, 0])
(1, 0)
```

```
>>> from sage.all import *
>>> m = matrix(Integer(2), [Integer(1), Integer(1), Integer(0), Integer(1)])
>>> V = ZZ**Integer(2)
>>> h1 = V.hom(m); h2 = V.hom(m, side='right')
>>> h1.side()
'left'
>>> h1([Integer(1), Integer(0)])
(1, 1)
>>> h2.side()
'right'
>>> h2([Integer(1), Integer(0)])
(1, 0)
```

side_switch()

Return the same morphism, acting on vectors on the opposite side.

EXAMPLES:

```
sage: m = matrix(2, [1,1,0,1]); m
[1 1]
[0 1]
sage: V = ZZ^2
sage: h = V.hom(m); h.side()
'left'
sage: h2 = h.side_switch(); h2
Free module morphism defined as left-multiplication by the matrix
[1 0]
[1 1]
Domain: Ambient free module of rank 2 over the principal ideal domain Integer␣
↪Ring
Codomain: Ambient free module of rank 2 over the principal ideal domain␣
↪Integer Ring
sage: h2.side()
'right'
sage: h2.side_switch().matrix()
[1 1]
[0 1]
```

```
>>> from sage.all import *
>>> m = matrix(Integer(2), [Integer(1),Integer(1),Integer(0),Integer(1)]); m
[1 1]
[0 1]
>>> V = ZZ**Integer(2)
>>> h = V.hom(m); h.side()
'left'
>>> h2 = h.side_switch(); h2
Free module morphism defined as left-multiplication by the matrix
[1 0]
[1 1]
Domain: Ambient free module of rank 2 over the principal ideal domain Integer␣
↪Ring
Codomain: Ambient free module of rank 2 over the principal ideal domain␣
↪Integer Ring
>>> h2.side()
'right'
>>> h2.side_switch().matrix()
[1 1]
[0 1]
```

trace()

Return the trace of this endomorphism.

EXAMPLES:

```
sage: # needs sage.modules
sage: V = ZZ^2; phi = V.hom([V.0 + V.1, 2*V.1])
sage: phi.trace()
```

(continues on next page)

(continued from previous page)

```
3
sage: W = CombinatorialFreeModule(ZZ, ['x', 'y'])
sage: M = matrix(ZZ, [[1, 0], [1, 2]])
sage: psi = W.module_morphism(matrix=M, codomain=W)
sage: psi.trace()
3
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> V = ZZ**Integer(2); phi = V.hom([V.gen(0) + V.gen(1), Integer(2)*V.
↪gen(1)])
>>> phi.trace()
3
>>> W = CombinatorialFreeModule(ZZ, ['x', 'y'])
>>> M = matrix(ZZ, [[Integer(1), Integer(0)], [Integer(1), Integer(2)]])
>>> psi = W.module_morphism(matrix=M, codomain=W)
>>> psi.trace()
3
```

sage.modules.matrix_morphism.**is_MatrixMorphism**(*x*)

Return `True` if x is a Matrix morphism of free modules.

This function is deprecated.

EXAMPLES:

```
sage: V = ZZ^2; phi = V.hom([3*V.0, 2*V.1])
sage: sage.modules.matrix_morphism.is_MatrixMorphism(phi)
doctest:warning...
DeprecationWarning: is_MatrixMorphism is deprecated;
use isinstance(..., MatrixMorphism_abstract) or categories instead
See https://github.com/sagemath/sage/issues/37731 for details.
True
sage: sage.modules.matrix_morphism.is_MatrixMorphism(3)
False
```

```
>>> from sage.all import *
>>> V = ZZ**Integer(2); phi = V.hom([Integer(3)*V.gen(0), Integer(2)*V.gen(1)])
>>> sage.modules.matrix_morphism.is_MatrixMorphism(phi)
doctest:warning...
DeprecationWarning: is_MatrixMorphism is deprecated;
use isinstance(..., MatrixMorphism_abstract) or categories instead
See https://github.com/sagemath/sage/issues/37731 for details.
True
>>> sage.modules.matrix_morphism.is_MatrixMorphism(Integer(3))
False
```

CHAPTER

EIGHT

PSEUDOMORPHISMS

8.1 Space of pseudomorphisms of free modules

AUTHORS:

- Xavier Caruso, Yossef Musleh (2024-09): initial version

class sage.modules.free_module_pseudohomspace.**FreeModulePseudoHomspace**(*domain*, *codomain*, *ore*)

Bases: UniqueRepresentation, HomsetWithBase

This class implements the space of pseudomorphisms with a fixed twist.

For free modules, the elements of a pseudomorphism correspond to matrices which define the mapping on elements of a basis.

This class is not supposed to be instantiated directly; the user should use instead the method sage.rings.module.free_module.FreeModule_generic.pseudoHom() to create a space of pseudomorphisms.

Element

alias of *FreeModulePseudoMorphism*

basis(*side='left'*)

Return a basis for the underlying matrix space.

The result does not depend on the *side* of the homspace, i.e. if matrices are acted upon on the left or on the right.

EXAMPLES:

```
sage: Fq = GF(7^3)
sage: Frob = Fq.frobenius_endomorphism()
sage: V = Fq^2
sage: PHS = V.pseudoHom(Frob)
sage: PHS.basis()
[Free module pseudomorphism (twisted by z3 |--> z3^7) defined by the matrix
[1 0]
[0 0]
Domain: Vector space of dimension 2 over Finite Field in z3 of size 7^3
Codomain: Vector space of dimension 2 over Finite Field in z3 of size 7^3,
Free module pseudomorphism (twisted by z3 |--> z3^7) defined by the matrix
[0 1]
[0 0]
Domain: Vector space of dimension 2 over Finite Field in z3 of size 7^3
Codomain: Vector space of dimension 2 over Finite Field in z3 of size 7^3,
```

(continues on next page)

(continued from previous page)

```
Free module pseudomorphism (twisted by z3 |--> z3^7) defined by the matrix
[0 0]
[1 0]
Domain: Vector space of dimension 2 over Finite Field in z3 of size 7^3
Codomain: Vector space of dimension 2 over Finite Field in z3 of size 7^3,
Free module pseudomorphism (twisted by z3 |--> z3^7) defined by the matrix
[0 0]
[0 1]
Domain: Vector space of dimension 2 over Finite Field in z3 of size 7^3
Codomain: Vector space of dimension 2 over Finite Field in z3 of size 7^3]
```

```
>>> from sage.all import *
>>> Fq = GF(Integer(7)**Integer(3))
>>> Frob = Fq.frobenius_endomorphism()
>>> V = Fq**Integer(2)
>>> PHS = V.pseudoHom(Frob)
>>> PHS.basis()
[Free module pseudomorphism (twisted by z3 |--> z3^7) defined by the matrix
[1 0]
[0 0]
Domain: Vector space of dimension 2 over Finite Field in z3 of size 7^3
Codomain: Vector space of dimension 2 over Finite Field in z3 of size 7^3,
Free module pseudomorphism (twisted by z3 |--> z3^7) defined by the matrix
[0 1]
[0 0]
Domain: Vector space of dimension 2 over Finite Field in z3 of size 7^3
Codomain: Vector space of dimension 2 over Finite Field in z3 of size 7^3,
Free module pseudomorphism (twisted by z3 |--> z3^7) defined by the matrix
[0 0]
[1 0]
Domain: Vector space of dimension 2 over Finite Field in z3 of size 7^3
Codomain: Vector space of dimension 2 over Finite Field in z3 of size 7^3,
Free module pseudomorphism (twisted by z3 |--> z3^7) defined by the matrix
[0 0]
[0 1]
Domain: Vector space of dimension 2 over Finite Field in z3 of size 7^3
Codomain: Vector space of dimension 2 over Finite Field in z3 of size 7^3]
```

matrix_space()

Return the matrix space used for representing the pseudomorphisms in this space.

EXAMPLES:

```
sage: Fq.<z> = GF(7^3)
sage: Frob = Fq.frobenius_endomorphism()
sage: V = Fq^2
sage: W = Fq^3
sage: H = V.pseudoHom(Frob, codomain=W)
sage: H.matrix_space()
Full MatrixSpace of 2 by 3 dense matrices over Finite Field in z of size 7^3
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> Fq = GF(Integer(7)**Integer(3), names=('z',)); (z,) = Fq._first_ngens(1)
>>> Frob = Fq.frobenius_endomorphism()
>>> V = Fq**Integer(2)
>>> W = Fq**Integer(3)
>>> H = V.pseudoHom(Frob, codomain=W)
>>> H.matrix_space()
Full MatrixSpace of 2 by 3 dense matrices over Finite Field in z of size 7^3
```

ore_ring(*var='x'*)

Return the underlying Ore polynomial ring, that is the Ore polynomial ring over the base field twisted by the twisting morphism and the twisting derivation attached to this homspace.

INPUT:

- `var` – string (default: `x`) the name of the variable

EXAMPLES:

```
sage: Fq.<z> = GF(7^3)
sage: Frob = Fq.frobenius_endomorphism()
sage: V = Fq^2
sage: H = V.pseudoHom(Frob)

sage: H.ore_ring()
Ore Polynomial Ring in x over Finite Field in z of size 7^3 twisted by z |-->␣
↪z^7

sage: H.ore_ring('y')
Ore Polynomial Ring in y over Finite Field in z of size 7^3 twisted by z |-->␣
↪z^7
```

```
>>> from sage.all import *
>>> Fq = GF(Integer(7)**Integer(3), names=('z',)); (z,) = Fq._first_ngens(1)
>>> Frob = Fq.frobenius_endomorphism()
>>> V = Fq**Integer(2)
>>> H = V.pseudoHom(Frob)

>>> H.ore_ring()
Ore Polynomial Ring in x over Finite Field in z of size 7^3 twisted by z |-->␣
↪z^7

>>> H.ore_ring('y')
Ore Polynomial Ring in y over Finite Field in z of size 7^3 twisted by z |-->␣
↪z^7
```

8.2 Pseudomorphisms of free modules

AUTHORS:

- Xavier Caruso, Yossef Musleh (2024-09): initial version

class sage.modules.free_module_pseudomorphism.**FreeModulePseudoMorphism**(*parent*, *f*, *side*)

Bases: Morphism

Let M, M' be modules over a ring R, $\theta : R \to R$ a ring homomorphism, and $\delta : R \to R$ a θ-derivation, which is a map such that:

$$\delta(xy) = \theta(x)\delta(y) + \delta(x)y.$$

A pseudomorphism $f : M \to M$ is an additive map such that

$$f(\lambda x) = \theta(\lambda)f(x) + \delta(\lambda)x$$

The map θ (resp. δ) is referred to as the twisting endomorphism (resp. the twisting derivation) of f.

Note

The implementation currently requires that M and M' are free modules.

We represent pseudomorphisms by matrices with coefficient in the base ring R. The matrix $\mathcal{M}_f$ representing f is such that its lines (resp. columns if `side` is `"right"`) are the coordinates of the images of the distinguished basis of the domain (see also method *matrix()*). More concretely, let n (resp. n') be the dimension of M (resp. M'), let $(e_1, \ldots, e_n)$ be a basis of M. For any $x = \sum_{i=1}^{n} x_i e_i \in M$, we have

$$f(x) = \begin{pmatrix}\theta(x_1) & \cdots & \theta(x_n)\end{pmatrix} \mathcal{M}_f + \begin{pmatrix}\delta(x_1) & \cdots & \theta(x_n)\end{pmatrix}.$$

When `side` is `"right"`, the formula is

$$f(x) = \mathcal{M}_f \begin{pmatrix}\theta(x_1) \\ \vdots \\ \theta(x_n)\end{pmatrix} + \begin{pmatrix}\delta(x_1) \\ \vdots \\ \theta(x_n)\end{pmatrix}.$$

This class is not supposed to be instantiated directly; the user should use instead the method `sage.rings.module.free_module.FreeModule_generic.pseudohom()` to create a pseudomorphism.

matrix()

Return the underlying matrix of this pseudomorphism.

It is defined as the matrix M whose lines (resp. columns if `side` is `"right"`) are the coordinates of the images of the distinguished basis of the domain.

EXAMPLES:

```
sage: Fq.<z> = GF(7^3)
sage: Frob = Fq.frobenius_endomorphism()
sage: M = Fq^3
sage: f = M.pseudohom([[1, z, 3], [0, 1, z^2], [z+1, 1, 1]], Frob)
sage: f.matrix()
[    1     z     3]
[    0     1   z^2]
[z + 1     1     1]
```

```
>>> from sage.all import *
>>> Fq = GF(Integer(7)**Integer(3), names=('z',)); (z,) = Fq._first_ngens(1)
>>> Frob = Fq.frobenius_endomorphism()
>>> M = Fq**Integer(3)
>>> f = M.pseudohom([[Integer(1), z, Integer(3)], [Integer(0), Integer(1),␣
↪z**Integer(2)], [z+Integer(1), Integer(1), Integer(1)]], Frob)
```

(continues on next page)

(continued from previous page)

```
>>> f.matrix()
[    1     z     3]
[    0     1   z^2]
[z + 1     1     1]
```

```
sage: e1, e2, e3 = M.basis()
sage: f(e1)
(1, z, 3)
sage: f(e2)
(0, 1, z^2)
sage: f(e3)
(z + 1, 1, 1)
```

```
>>> from sage.all import *
>>> e1, e2, e3 = M.basis()
>>> f(e1)
(1, z, 3)
>>> f(e2)
(0, 1, z^2)
>>> f(e3)
(z + 1, 1, 1)
```

ore_module(*names=None*)

Return the Ore module over which the Ore variable acts through this pseudomorphism.

INPUT:

- `names` – a string, a list of strings or `None`, the names of the vector of the canonical basis of the Ore module; if `None`, elements are represented as vectors in K^d (where K is the base ring)

EXAMPLES:

```
sage: Fq.<z> = GF(7^3)
sage: Frob = Fq.frobenius_endomorphism()
sage: V = Fq^2
sage: mat = matrix(2, [1, z, z^2, z^3])
sage: f = V.pseudohom(mat, Frob)

sage: M = f.ore_module()
sage: M
Ore module of rank 2 over Finite Field in z of size 7^3 twisted by z |--> z^7
```

```
>>> from sage.all import *
>>> Fq = GF(Integer(7)**Integer(3), names=('z',)); (z,) = Fq._first_ngens(1)
>>> Frob = Fq.frobenius_endomorphism()
>>> V = Fq**Integer(2)
>>> mat = matrix(Integer(2), [Integer(1), z, z**Integer(2), z**Integer(3)])
>>> f = V.pseudohom(mat, Frob)

>>> M = f.ore_module()
>>> M
Ore module of rank 2 over Finite Field in z of size 7^3 twisted by z |--> z^7
```

Here M is a module over the Ore ring $\mathbb{F}_q[X; \mathrm{Frob}]$ and the variable X acts on M through f:

```
sage: S.<X> = M.ore_ring()
sage: S
Ore Polynomial Ring in X over Finite Field in z of size 7^3 twisted by z |-->␣
↪z^7
sage: v = M((1,0))
sage: X*v
(1, z)
```

```
>>> from sage.all import *
>>> S = M.ore_ring(names=('X',)); (X,) = S._first_ngens(1)
>>> S
Ore Polynomial Ring in X over Finite Field in z of size 7^3 twisted by z |-->␣
↪z^7
>>> v = M((Integer(1),Integer(0)))
>>> X*v
(1, z)
```

The argument `names` can be used to give chosen names to the vectors in the canonical basis:

```
sage: M = f.ore_module(names=('v', 'w'))
sage: M.basis()
[v, w]
```

```
>>> from sage.all import *
>>> M = f.ore_module(names=('v', 'w'))
>>> M.basis()
[v, w]
```

or even:

```
sage: M = f.ore_module(names='e')
sage: M.basis()
[e0, e1]
```

```
>>> from sage.all import *
>>> M = f.ore_module(names='e')
>>> M.basis()
[e0, e1]
```

Note that the bracket construction also works:

```
sage: M.<v,w> = f.ore_module()
sage: M.basis()
[v, w]
sage: v + w
v + w
```

```
>>> from sage.all import *
>>> M = f.ore_module(names=('v', 'w',)); (v, w,) = M._first_ngens(2)
>>> M.basis()
[v, w]
>>> v + w
v + w
```

We refer to sage.modules.ore_module for a tutorial on Ore modules in SageMath.

See also

sage.modules.ore_module

side()

Return the side of vectors acted on, relative to the matrix.

EXAMPLES:

```
sage: Fq.<z> = GF(7^3)
sage: Frob = Fq.frobenius_endomorphism()
sage: V = Fq^2

sage: m = matrix(2, [1, z, z^2, z^3])
sage: h1 = V.pseudohom(m, Frob)
sage: h1.side()
'left'
sage: h1([1, 0])
(1, z)

sage: h2 = V.pseudohom(m, Frob, side="right")
sage: h2.side()
'right'
sage: h2([1, 0])
(1, z^2)
```

```
>>> from sage.all import *
>>> Fq = GF(Integer(7)**Integer(3), names=('z',)); (z,) = Fq._first_ngens(1)
>>> Frob = Fq.frobenius_endomorphism()
>>> V = Fq**Integer(2)

>>> m = matrix(Integer(2), [Integer(1), z, z**Integer(2), z**Integer(3)])
>>> h1 = V.pseudohom(m, Frob)
>>> h1.side()
'left'
>>> h1([Integer(1), Integer(0)])
(1, z)

>>> h2 = V.pseudohom(m, Frob, side="right")
>>> h2.side()
'right'
>>> h2([Integer(1), Integer(0)])
(1, z^2)
```

side_switch()

Return the same morphism, acting on vectors on the opposite side.

EXAMPLES:

```
sage: Fq.<z> = GF(7^3)
sage: Frob = Fq.frobenius_endomorphism()
```

(continues on next page)

(continued from previous page)

```
sage: V = Fq^2

sage: m = matrix(2, [1, z, z^2, z^3])
sage: h1 = V.pseudohom(m, Frob)
sage: h1
Free module pseudomorphism (twisted by z |--> z^7) defined by the matrix
[      1       z]
[    z^2 z^2 + 3]
Domain: Vector space of dimension 2 over Finite Field in z of size 7^3
Codomain: Vector space of dimension 2 over Finite Field in z of size 7^3

sage: h2 = h1.side_switch()
sage: h2
Free module pseudomorphism (twisted by z |--> z^7) defined as left-
↪multiplication by the matrix
[      1     z^2]
[      z z^2 + 3]
Domain: Vector space of dimension 2 over Finite Field in z of size 7^3
Codomain: Vector space of dimension 2 over Finite Field in z of size 7^3
```

```
>>> from sage.all import *
>>> Fq = GF(Integer(7)**Integer(3), names=('z',)); (z,) = Fq._first_ngens(1)
>>> Frob = Fq.frobenius_endomorphism()
>>> V = Fq**Integer(2)

>>> m = matrix(Integer(2), [Integer(1), z, z**Integer(2), z**Integer(3)])
>>> h1 = V.pseudohom(m, Frob)
>>> h1
Free module pseudomorphism (twisted by z |--> z^7) defined by the matrix
[      1       z]
[    z^2 z^2 + 3]
Domain: Vector space of dimension 2 over Finite Field in z of size 7^3
Codomain: Vector space of dimension 2 over Finite Field in z of size 7^3

>>> h2 = h1.side_switch()
>>> h2
Free module pseudomorphism (twisted by z |--> z^7) defined as left-
↪multiplication by the matrix
[      1     z^2]
[      z z^2 + 3]
Domain: Vector space of dimension 2 over Finite Field in z of size 7^3
Codomain: Vector space of dimension 2 over Finite Field in z of size 7^3
```

We check that `h1` and `h2` are the same:

```
sage: v = V.random_element()
sage: h1(v) == h2(v)
True
```

```
>>> from sage.all import *
>>> v = V.random_element()
>>> h1(v) == h2(v)
```

(continues on next page)

(continued from previous page)

```
True
```

twisting_derivation()

Return the twisting derivation of the pseudomorphism (or `None` if the twisting derivation is zero).

EXAMPLES:

```
sage: P.<x> = ZZ[]
sage: d = P.derivation()
sage: M = P^2
sage: f = M.pseudohom([[1, 2*x], [x, 1]], d)
sage: f.twisting_derivation()
d/dx
```

```
>>> from sage.all import *
>>> P = ZZ['x']; (x,) = P._first_ngens(1)
>>> d = P.derivation()
>>> M = P**Integer(2)
>>> f = M.pseudohom([[Integer(1), Integer(2)*x], [x, Integer(1)]], d)
>>> f.twisting_derivation()
d/dx
```

```
sage: Fq.<z> = GF(7^3)
sage: Frob = Fq.frobenius_endomorphism()
sage: V = Fq^2
sage: f = V.pseudohom([[1, z], [0, z^2]], Frob)
sage: f.twisting_derivation()
```

```
>>> from sage.all import *
>>> Fq = GF(Integer(7)**Integer(3), names=('z',)); (z,) = Fq._first_ngens(1)
>>> Frob = Fq.frobenius_endomorphism()
>>> V = Fq**Integer(2)
>>> f = V.pseudohom([[Integer(1), z], [Integer(0), z**Integer(2)]], Frob)
>>> f.twisting_derivation()
```

twisting_morphism()

Return the twisting morphism of the pseudomorphism (or `None` if the twisting morphism is the identity).

EXAMPLES:

```
sage: Fq.<z> = GF(7^3)
sage: Frob = Fq.frobenius_endomorphism()
sage: V = Fq^2
sage: f = V.pseudohom([[1, z], [0, z^2]], Frob)
sage: f.twisting_morphism()
Frobenius endomorphism z |--> z^7 on Finite Field in z of size 7^3
```

```
>>> from sage.all import *
>>> Fq = GF(Integer(7)**Integer(3), names=('z',)); (z,) = Fq._first_ngens(1)
>>> Frob = Fq.frobenius_endomorphism()
>>> V = Fq**Integer(2)
>>> f = V.pseudohom([[Integer(1), z], [Integer(0), z**Integer(2)]], Frob)
```

(continues on next page)

(continued from previous page)

```
>>> f.twisting_morphism()
Frobenius endomorphism z |--> z^7 on Finite Field in z of size 7^3
```

```
sage: P.<x> = ZZ[]
sage: d = P.derivation()
sage: M = P^2
sage: f = M.pseudohom([[1, 2*x], [x, 1]], d)
sage: f.twisting_morphism()
```

```
>>> from sage.all import *
>>> P = ZZ['x']; (x,) = P._first_ngens(1)
>>> d = P.derivation()
>>> M = P**Integer(2)
>>> f = M.pseudohom([[Integer(1), Integer(2)*x], [x, Integer(1)]], d)
>>> f.twisting_morphism()
```

CHAPTER

NINE

VECTORS

9.1 Vectors with integer entries

AUTHOR:

- William Stein (2007)

EXAMPLES:

```
sage: v = vector(ZZ, [1,2,3,4,5])
sage: v
(1, 2, 3, 4, 5)
sage: 3*v
(3, 6, 9, 12, 15)
sage: v*7
(7, 14, 21, 28, 35)
sage: -v
(-1, -2, -3, -4, -5)
sage: v - v
(0, 0, 0, 0, 0)
sage: v + v
(2, 4, 6, 8, 10)
sage: v * v   # dot product.
55
```

```
>>> from sage.all import *
>>> v = vector(ZZ, [Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)])
>>> v
(1, 2, 3, 4, 5)
>>> Integer(3)*v
(3, 6, 9, 12, 15)
>>> v*Integer(7)
(7, 14, 21, 28, 35)
>>> -v
(-1, -2, -3, -4, -5)
>>> v - v
(0, 0, 0, 0, 0)
>>> v + v
(2, 4, 6, 8, 10)
>>> v * v   # dot product.
55
```

We make a large zero vector:

```
sage: k = ZZ^100000; k
Ambient free module of rank 100000 over the principal ideal domain Integer Ring
sage: v = k(0)
sage: v[:10]
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

```
>>> from sage.all import *
>>> k = ZZ**Integer(100000); k
Ambient free module of rank 100000 over the principal ideal domain Integer Ring
>>> v = k(Integer(0))
>>> v[:Integer(10)]
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

class sage.modules.vector_integer_dense.**Vector_integer_dense**

Bases: *FreeModuleElement*

list(*copy=True*)

The list of entries of the vector.

INPUT:

- copy – ignored optional argument

EXAMPLES:

```
sage: v = vector([1,2,3,4])
sage: a = v.list(copy=False); a
[1, 2, 3, 4]
sage: a[0] = 0
sage: v
(1, 2, 3, 4)
```

```
>>> from sage.all import *
>>> v = vector([Integer(1),Integer(2),Integer(3),Integer(4)])
>>> a = v.list(copy=False); a
[1, 2, 3, 4]
>>> a[Integer(0)] = Integer(0)
>>> v
(1, 2, 3, 4)
```

sage.modules.vector_integer_dense.**unpickle_v0**(*parent*, *entries*, *degree*)

sage.modules.vector_integer_dense.**unpickle_v1**(*parent*, *entries*, *degree*, *is_mutable*)

9.2 File: sage/modules/vector_integer_sparse.pyx (starting at line 1)

9.3 Vectors with elements in $\mathbf{F}_2$

AUTHOR:

- Martin Albrecht (2009-12): initial implementation
- Thomas Feulner (2012-11): added *Vector_mod2_dense.hamming_weight()*

EXAMPLES:

```
sage: VS = GF(2)^3
sage: e = VS.random_element()
sage: e.parent() is VS
True
sage: S = set(vector(v, immutable=True) for v in VS)
sage: S1 = set()
sage: while S != S1:
....:     S1.add(vector(VS.random_element(), immutable=True))
```

```
>>> from sage.all import *
>>> VS = GF(Integer(2))**Integer(3)
>>> e = VS.random_element()
>>> e.parent() is VS
True
>>> S = set(vector(v, immutable=True) for v in VS)
>>> S1 = set()
>>> while S != S1:
...     S1.add(vector(VS.random_element(), immutable=True))
```

class sage.modules.vector_mod2_dense.**Vector_mod2_dense**

Bases: *FreeModuleElement*

EXAMPLES:

```
sage: VS = VectorSpace(GF(2),3)
sage: VS((0,0,1/3))
(0, 0, 1)
sage: type(_)
<class 'sage.modules.vector_mod2_dense.Vector_mod2_dense'>
sage: VS((0,0,int(3)))
(0, 0, 1)
sage: VS((0,0,3))
(0, 0, 1)
sage: VS((0,0,GF(2)(1)))
(0, 0, 1)
```

```
>>> from sage.all import *
>>> VS = VectorSpace(GF(Integer(2)),Integer(3))
>>> VS((Integer(0),Integer(0),Integer(1)/Integer(3)))
(0, 0, 1)
>>> type(_)
<class 'sage.modules.vector_mod2_dense.Vector_mod2_dense'>
>>> VS((Integer(0),Integer(0),int(Integer(3))))
(0, 0, 1)
>>> VS((Integer(0),Integer(0),Integer(3)))
(0, 0, 1)
>>> VS((Integer(0),Integer(0),GF(Integer(2))(Integer(1))))
(0, 0, 1)
```

hamming_weight()

Return the number of positions `i` such that `self[i] != 0`.

EXAMPLES:

```
sage: vector(GF(2), [1,1,0]).hamming_weight()
2
```

```
>>> from sage.all import *
>>> vector(GF(Integer(2)), [Integer(1),Integer(1),Integer(0)]).hamming_
↪weight()
2
```

list(*copy=True*)

Return a list of entries in `self`.

INPUT:

- `copy` – always `True`

EXAMPLES:

```
sage: VS = VectorSpace(GF(2), 10)
sage: entries = [GF(2).random_element() for _ in range(10)]
sage: e = VS(entries)
sage: e.list() == entries
True
```

```
>>> from sage.all import *
>>> VS = VectorSpace(GF(Integer(2)), Integer(10))
>>> entries = [GF(Integer(2)).random_element() for _ in range(Integer(10))]
>>> e = VS(entries)
>>> e.list() == entries
True
```

sage.modules.vector_mod2_dense.**unpickle_v0**(*parent*, *entries*, *degree*, *is_immutable*)

EXAMPLES:

```
sage: from sage.modules.vector_mod2_dense import unpickle_v0
sage: VS = VectorSpace(GF(2),10)
sage: unpickle_v0(VS, [0,1,2,3,4,5,6,7,8,9], 10, 0)
(0, 1, 0, 1, 0, 1, 0, 1, 0, 1)
```

```
>>> from sage.all import *
>>> from sage.modules.vector_mod2_dense import unpickle_v0
>>> VS = VectorSpace(GF(Integer(2)),Integer(10))
>>> unpickle_v0(VS, [Integer(0),Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5),Integer(6),Integer(7),Integer(8),Integer(9)], Integer(10),␣
↪Integer(0))
(0, 1, 0, 1, 0, 1, 0, 1, 0, 1)
```

9.4 Vectors with integer mod n entries, with small n

EXAMPLES:

```
sage: v = vector(Integers(8), [1,2,3,4,5])
sage: type(v)
```

(continues on next page)

(continued from previous page)

```
<class 'sage.modules.vector_modn_dense.Vector_modn_dense'>
sage: v
(1, 2, 3, 4, 5)
sage: 3*v
(3, 6, 1, 4, 7)
sage: v*7
(7, 6, 5, 4, 3)
sage: -v
(7, 6, 5, 4, 3)
sage: v - v
(0, 0, 0, 0, 0)
sage: v + v
(2, 4, 6, 0, 2)
sage: v * v
7

sage: v = vector(Integers(8), [1,2,3,4,5])
sage: u = vector(Integers(8), [1,2,3,4,4])
sage: v - u
(0, 0, 0, 0, 1)
sage: u - v
(0, 0, 0, 0, 7)

sage: v = vector((Integers(5)(1),2,3,4,4))
sage: u = vector((Integers(5)(1),2,3,4,3))
sage: v - u
(0, 0, 0, 0, 1)
sage: u - v
(0, 0, 0, 0, 4)
```

```
>>> from sage.all import *
>>> v = vector(Integers(Integer(8)), [Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5)])
>>> type(v)
<class 'sage.modules.vector_modn_dense.Vector_modn_dense'>
>>> v
(1, 2, 3, 4, 5)
>>> Integer(3)*v
(3, 6, 1, 4, 7)
>>> v*Integer(7)
(7, 6, 5, 4, 3)
>>> -v
(7, 6, 5, 4, 3)
>>> v - v
(0, 0, 0, 0, 0)
>>> v + v
(2, 4, 6, 0, 2)
>>> v * v
7

>>> v = vector(Integers(Integer(8)), [Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5)])
```

(continues on next page)

(continued from previous page)

```
>>> u = vector(Integers(Integer(8)), [Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(4)])
>>> v - u
(0, 0, 0, 0, 1)
>>> u - v
(0, 0, 0, 0, 7)

>>> v = vector((Integers(Integer(5))(Integer(1)),Integer(2),Integer(3),Integer(4),
↪Integer(4)))
>>> u = vector((Integers(Integer(5))(Integer(1)),Integer(2),Integer(3),Integer(4),
↪Integer(3)))
>>> v - u
(0, 0, 0, 0, 1)
>>> u - v
(0, 0, 0, 0, 4)
```

We make a large zero vector:

```
sage: k = Integers(8)^100000; k
Ambient free module of rank 100000 over Ring of integers modulo 8
sage: v = k(0)
sage: v[:10]
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

```
>>> from sage.all import *
>>> k = Integers(Integer(8))**Integer(100000); k
Ambient free module of rank 100000 over Ring of integers modulo 8
>>> v = k(Integer(0))
>>> v[:Integer(10)]
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

We multiply a vector by a matrix:

```
sage: a = (GF(97)^5)(range(5))
sage: m = matrix(GF(97), 5, range(25))
sage: a*m
(53, 63, 73, 83, 93)
```

```
>>> from sage.all import *
>>> a = (GF(Integer(97))**Integer(5))(range(Integer(5)))
>>> m = matrix(GF(Integer(97)), Integer(5), range(Integer(25)))
>>> a*m
(53, 63, 73, 83, 93)
```

AUTHOR:

- William Stein (2007)

class sage.modules.vector_modn_dense.**Vector_modn_dense**

Bases: *FreeModuleElement*

Create an element.

sage.modules.vector_modn_dense.**unpickle_v0**(*parent*, *entries*, *degree*, *p*)

sage.modules.vector_modn_dense.**unpickle_v1**(*parent*, *entries*, *degree*, *p*, *is_mutable*)

9.5 File: sage/modules/vector_modn_sparse.pyx (starting at line 1)

9.6 Vectors with rational entries

AUTHOR:

- William Stein (2007)
- Soroosh Yazdani (2007)

EXAMPLES:

```
sage: v = vector(QQ,[1,2,3,4,5])
sage: v
(1, 2, 3, 4, 5)
sage: 3*v
(3, 6, 9, 12, 15)
sage: v/2
(1/2, 1, 3/2, 2, 5/2)
sage: -v
(-1, -2, -3, -4, -5)
sage: v - v
(0, 0, 0, 0, 0)
sage: v + v
(2, 4, 6, 8, 10)
sage: v * v
55
```

```
>>> from sage.all import *
>>> v = vector(QQ,[Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)])
>>> v
(1, 2, 3, 4, 5)
>>> Integer(3)*v
(3, 6, 9, 12, 15)
>>> v/Integer(2)
(1/2, 1, 3/2, 2, 5/2)
>>> -v
(-1, -2, -3, -4, -5)
>>> v - v
(0, 0, 0, 0, 0)
>>> v + v
(2, 4, 6, 8, 10)
>>> v * v
55
```

We make a large zero vector:

```
sage: k = QQ^100000; k
Vector space of dimension 100000 over Rational Field
sage: v = k(0)
sage: v[:10]
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

```
>>> from sage.all import *
>>> k = QQ**Integer(100000); k
Vector space of dimension 100000 over Rational Field
>>> v = k(Integer(0))
>>> v[:Integer(10)]
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

class sage.modules.vector_rational_dense.**Vector_rational_dense**

Bases: *FreeModuleElement*

list(*copy=True*)

The list of entries of the vector.

INPUT:

- copy – ignored optional argument

EXAMPLES:

```
sage: v = vector(QQ,[1,2,3,4])
sage: a = v.list(copy=False); a
[1, 2, 3, 4]
sage: a[0] = 0
sage: v
(1, 2, 3, 4)
```

```
>>> from sage.all import *
>>> v = vector(QQ,[Integer(1),Integer(2),Integer(3),Integer(4)])
>>> a = v.list(copy=False); a
[1, 2, 3, 4]
>>> a[Integer(0)] = Integer(0)
>>> v
(1, 2, 3, 4)
```

sage.modules.vector_rational_dense.**unpickle_v0**(*parent*, *entries*, *degree*)

sage.modules.vector_rational_dense.**unpickle_v1**(*parent*, *entries*, *degree*, *is_mutable*)

9.7 File: sage/modules/vector_rational_sparse.pyx (starting at line 1)

9.8 Dense vectors over the symbolic ring

Implements dense vectors over the symbolic ring.

AUTHORS:

- Robert Bradshaw (2011-05-25): Added more element-wise simplification methods
- Joris Vankerschaver (2011-05-15): Initial version

EXAMPLES:

```
sage: x, y = var('x, y')
sage: u = vector([sin(x)^2 + cos(x)^2, log(2*y) + log(3*y)]); u
(cos(x)^2 + sin(x)^2, log(3*y) + log(2*y))
```

(continues on next page)

(continued from previous page)

```
sage: type(u)
<class 'sage.modules.free_module.FreeModule_ambient_field_with_category.element_class
↪'>
sage: u.simplify_full()
(1, log(3*y) + log(2*y))
```

```
>>> from sage.all import *
>>> x, y = var('x, y')
>>> u = vector([sin(x)**Integer(2) + cos(x)**Integer(2), log(Integer(2)*y) +␣
↪log(Integer(3)*y)]); u
(cos(x)^2 + sin(x)^2, log(3*y) + log(2*y))
>>> type(u)
<class 'sage.modules.free_module.FreeModule_ambient_field_with_category.element_class
↪'>
>>> u.simplify_full()
(1, log(3*y) + log(2*y))
```

class sage.modules.vector_symbolic_dense.**Vector_symbolic_dense**

Bases: *FreeModuleElement_generic_dense*

canonicalize_radical(**args*, ***kwds*)

Generic function used to implement common symbolic operations elementwise as methods of a vector.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: v = vector([sin(x)^2 + cos(x)^2, log(x*y), sin(x/(x^2 + x)),␣
↪factorial(x+1)/factorial(x)])
sage: v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
sage: v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
sage: v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
sage: v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
sage: v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

sage: v = vector([sin(2*x), sin(3*x)])
sage: v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
sage: v.simplify_trig(False)
(sin(2*x), sin(3*x))
sage: v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> v = vector([sin(x)**Integer(2) + cos(x)**Integer(2), log(x*y),␣
↪sin(x/(x**Integer(2) + x)), factorial(x+Integer(1))/factorial(x)])
>>> v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
>>> v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
>>> v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
>>> v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
>>> v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

>>> v = vector([sin(Integer(2)*x), sin(Integer(3)*x)])
>>> v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
>>> v.simplify_trig(False)
(sin(2*x), sin(3*x))
>>> v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

See Expression.canonicalize_radical() for optional arguments.

simplify(**args*, ***kwds*)

Generic function used to implement common symbolic operations elementwise as methods of a vector.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: v = vector([sin(x)^2 + cos(x)^2, log(x*y), sin(x/(x^2 + x)),␣
↪factorial(x+1)/factorial(x)])
sage: v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
sage: v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
sage: v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
sage: v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
sage: v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

sage: v = vector([sin(2*x), sin(3*x)])
sage: v.simplify_trig()
```

(continues on next page)

(continued from previous page)

```
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
sage: v.simplify_trig(False)
(sin(2*x), sin(3*x))
sage: v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> v = vector([sin(x)**Integer(2) + cos(x)**Integer(2), log(x*y),␣
↪sin(x/(x**Integer(2) + x)), factorial(x+Integer(1))/factorial(x)])
>>> v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
>>> v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
>>> v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
>>> v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
>>> v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

>>> v = vector([sin(Integer(2)*x), sin(Integer(3)*x)])
>>> v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
>>> v.simplify_trig(False)
(sin(2*x), sin(3*x))
>>> v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

See Expression.simplify() for optional arguments.

simplify_factorial(**args*, ***kwds*)

Generic function used to implement common symbolic operations elementwise as methods of a vector.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: v = vector([sin(x)^2 + cos(x)^2, log(x*y), sin(x/(x^2 + x)),␣
↪factorial(x+1)/factorial(x)])
sage: v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
sage: v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
sage: v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
```

(continues on next page)

(continued from previous page)

```
sage: v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
sage: v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

sage: v = vector([sin(2*x), sin(3*x)])
sage: v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
sage: v.simplify_trig(False)
(sin(2*x), sin(3*x))
sage: v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> v = vector([sin(x)**Integer(2) + cos(x)**Integer(2), log(x*y),␣
↪sin(x/(x**Integer(2) + x)), factorial(x+Integer(1))/factorial(x)])
>>> v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
>>> v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
>>> v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
>>> v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
>>> v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

>>> v = vector([sin(Integer(2)*x), sin(Integer(3)*x)])
>>> v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
>>> v.simplify_trig(False)
(sin(2*x), sin(3*x))
>>> v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

See Expression.simplify_factorial() for optional arguments.

simplify_full(**args*, ***kwds*)

Generic function used to implement common symbolic operations elementwise as methods of a vector.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: v = vector([sin(x)^2 + cos(x)^2, log(x*y), sin(x/(x^2 + x)),␣
↪factorial(x+1)/factorial(x)])
sage: v.simplify_trig()
```

(continues on next page)

(continued from previous page)

```
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
sage: v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
sage: v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
sage: v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
sage: v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

sage: v = vector([sin(2*x), sin(3*x)])
sage: v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
sage: v.simplify_trig(False)
(sin(2*x), sin(3*x))
sage: v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> v = vector([sin(x)**Integer(2) + cos(x)**Integer(2), log(x*y),␣
↪sin(x/(x**Integer(2) + x)), factorial(x+Integer(1))/factorial(x)])
>>> v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
>>> v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
>>> v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
>>> v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
>>> v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

>>> v = vector([sin(Integer(2)*x), sin(Integer(3)*x)])
>>> v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
>>> v.simplify_trig(False)
(sin(2*x), sin(3*x))
>>> v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

See Expression.simplify_full() for optional arguments.

simplify_log(**args*, ***kwds*)

Generic function used to implement common symbolic operations elementwise as methods of a vector.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: v = vector([sin(x)^2 + cos(x)^2, log(x*y), sin(x/(x^2 + x)),␣
↪factorial(x+1)/factorial(x)])
sage: v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
sage: v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
sage: v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
sage: v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
sage: v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

sage: v = vector([sin(2*x), sin(3*x)])
sage: v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
sage: v.simplify_trig(False)
(sin(2*x), sin(3*x))
sage: v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> v = vector([sin(x)**Integer(2) + cos(x)**Integer(2), log(x*y),␣
↪sin(x/(x**Integer(2) + x)), factorial(x+Integer(1))/factorial(x)])
>>> v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
>>> v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
>>> v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
>>> v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
>>> v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

>>> v = vector([sin(Integer(2)*x), sin(Integer(3)*x)])
>>> v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
>>> v.simplify_trig(False)
(sin(2*x), sin(3*x))
>>> v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

See Expression.simplify_log() for optional arguments.

simplify_rational(**args*, ***kwds*)

Generic function used to implement common symbolic operations elementwise as methods of a vector.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: v = vector([sin(x)^2 + cos(x)^2, log(x*y), sin(x/(x^2 + x)),␣
↪factorial(x+1)/factorial(x)])
sage: v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
sage: v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
sage: v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
sage: v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
sage: v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

sage: v = vector([sin(2*x), sin(3*x)])
sage: v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
sage: v.simplify_trig(False)
(sin(2*x), sin(3*x))
sage: v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> v = vector([sin(x)**Integer(2) + cos(x)**Integer(2), log(x*y),␣
↪sin(x/(x**Integer(2) + x)), factorial(x+Integer(1))/factorial(x)])
>>> v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
>>> v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
>>> v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
>>> v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
>>> v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

>>> v = vector([sin(Integer(2)*x), sin(Integer(3)*x)])
>>> v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
>>> v.simplify_trig(False)
```

(continues on next page)

(continued from previous page)

```
(sin(2*x), sin(3*x))
>>> v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

See Expression.simplify_rational() for optional arguments.

simplify_trig(**args*, ***kwds*)

Generic function used to implement common symbolic operations elementwise as methods of a vector.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: v = vector([sin(x)^2 + cos(x)^2, log(x*y), sin(x/(x^2 + x)),␣
↪factorial(x+1)/factorial(x)])
sage: v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
sage: v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
sage: v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
sage: v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
sage: v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

sage: v = vector([sin(2*x), sin(3*x)])
sage: v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
sage: v.simplify_trig(False)
(sin(2*x), sin(3*x))
sage: v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> v = vector([sin(x)**Integer(2) + cos(x)**Integer(2), log(x*y),␣
↪sin(x/(x**Integer(2) + x)), factorial(x+Integer(1))/factorial(x)])
>>> v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
>>> v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
>>> v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
>>> v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
```

(continues on next page)

(continued from previous page)

```
>>> v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

>>> v = vector([sin(Integer(2)*x), sin(Integer(3)*x)])
>>> v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
>>> v.simplify_trig(False)
(sin(2*x), sin(3*x))
>>> v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

See Expression.simplify_trig() for optional arguments.

trig_expand(**args*, ***kwds*)

Generic function used to implement common symbolic operations elementwise as methods of a vector.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: v = vector([sin(x)^2 + cos(x)^2, log(x*y), sin(x/(x^2 + x)),␣
↪factorial(x+1)/factorial(x)])
sage: v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
sage: v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
sage: v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
sage: v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
sage: v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

sage: v = vector([sin(2*x), sin(3*x)])
sage: v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
sage: v.simplify_trig(False)
(sin(2*x), sin(3*x))
sage: v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> v = vector([sin(x)**Integer(2) + cos(x)**Integer(2), log(x*y),␣
↪sin(x/(x**Integer(2) + x)), factorial(x+Integer(1))/factorial(x)])
>>> v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
>>> v.canonicalize_radical()
```

(continues on next page)

(continued from previous page)

```
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
>>> v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
>>> v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
>>> v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

>>> v = vector([sin(Integer(2)*x), sin(Integer(3)*x)])
>>> v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
>>> v.simplify_trig(False)
(sin(2*x), sin(3*x))
>>> v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

See Expression.expand_trig() for optional arguments.

trig_reduce(**args*, ***kwds*)

Generic function used to implement common symbolic operations elementwise as methods of a vector.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: v = vector([sin(x)^2 + cos(x)^2, log(x*y), sin(x/(x^2 + x)),␣
↪factorial(x+1)/factorial(x)])
sage: v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
sage: v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
sage: v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
sage: v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
sage: v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

sage: v = vector([sin(2*x), sin(3*x)])
sage: v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
sage: v.simplify_trig(False)
(sin(2*x), sin(3*x))
sage: v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> var('x,y')
(x, y)
>>> v = vector([sin(x)**Integer(2) + cos(x)**Integer(2), log(x*y),␣
↪sin(x/(x**Integer(2) + x)), factorial(x+Integer(1))/factorial(x)])
>>> v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
>>> v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
>>> v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
>>> v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
>>> v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

>>> v = vector([sin(Integer(2)*x), sin(Integer(3)*x)])
>>> v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
>>> v.simplify_trig(False)
(sin(2*x), sin(3*x))
>>> v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

See Expression.reduce_trig() for optional arguments.

sage.modules.vector_symbolic_dense.**apply_map**(*phi*)

Return a function that applies `phi` to its argument.

EXAMPLES:

```
sage: from sage.modules.vector_symbolic_dense import apply_map
sage: v = vector([1,2,3])
sage: f = apply_map(lambda x: x+1)
sage: f(v)
(2, 3, 4)
```

```
>>> from sage.all import *
>>> from sage.modules.vector_symbolic_dense import apply_map
>>> v = vector([Integer(1),Integer(2),Integer(3)])
>>> f = apply_map(lambda x: x+Integer(1))
>>> f(v)
(2, 3, 4)
```

9.9 Sparse vectors over the symbolic ring

Implements vectors over the symbolic ring.

AUTHORS:

- Robert Bradshaw (2011-05-25): Added more element-wise simplification methods
- Joris Vankerschaver (2011-05-15): Initial version

- Dima Pasechnik (2023-06-04): cloning from the dense case

EXAMPLES:

```
sage: x, y = var('x, y')
sage: u = vector([sin(x)^2 + cos(x)^2, log(2*y) + log(3*y)], sparse=True); u
(cos(x)^2 + sin(x)^2, log(3*y) + log(2*y))
sage: type(u)
<class 'sage.modules.free_module.FreeModule_ambient_field_with_category.element_class
↪'>
sage: u.simplify_full()
(1, log(3*y) + log(2*y))
```

```
>>> from sage.all import *
>>> x, y = var('x, y')
>>> u = vector([sin(x)**Integer(2) + cos(x)**Integer(2), log(Integer(2)*y) +␣
↪log(Integer(3)*y)], sparse=True); u
(cos(x)^2 + sin(x)^2, log(3*y) + log(2*y))
>>> type(u)
<class 'sage.modules.free_module.FreeModule_ambient_field_with_category.element_class
↪'>
>>> u.simplify_full()
(1, log(3*y) + log(2*y))
```

class sage.modules.vector_symbolic_sparse.**Vector_symbolic_sparse**

Bases: *FreeModuleElement_generic_sparse*

canonicalize_radical(**args*, ***kwds*)

Generic function used to implement common symbolic operations elementwise as methods of a vector.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: v = vector([sin(x)^2 + cos(x)^2, log(x*y), sin(x/(x^2 + x)),␣
↪factorial(x+1)/factorial(x)], sparse=True)
sage: v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
sage: v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
sage: v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
sage: v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
sage: v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

sage: v = vector([sin(2*x), sin(3*x)], sparse=True)
sage: v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
sage: v.simplify_trig(False)
```

(continues on next page)

(continued from previous page)

```
(sin(2*x), sin(3*x))
sage: v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> v = vector([sin(x)**Integer(2) + cos(x)**Integer(2), log(x*y),␣
↪sin(x/(x**Integer(2) + x)), factorial(x+Integer(1))/factorial(x)],␣
↪sparse=True)
>>> v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
>>> v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
>>> v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
>>> v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
>>> v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

>>> v = vector([sin(Integer(2)*x), sin(Integer(3)*x)], sparse=True)
>>> v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
>>> v.simplify_trig(False)
(sin(2*x), sin(3*x))
>>> v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

See Expression.canonicalize_radical() for optional arguments.

simplify(**args*, ***kwds*)

Generic function used to implement common symbolic operations elementwise as methods of a vector.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: v = vector([sin(x)^2 + cos(x)^2, log(x*y), sin(x/(x^2 + x)),␣
↪factorial(x+1)/factorial(x)], sparse=True)
sage: v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
sage: v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
sage: v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
sage: v.simplify_factorial()
```

(continues on next page)

(continued from previous page)

```
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
sage: v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

sage: v = vector([sin(2*x), sin(3*x)], sparse=True)
sage: v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
sage: v.simplify_trig(False)
(sin(2*x), sin(3*x))
sage: v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> v = vector([sin(x)**Integer(2) + cos(x)**Integer(2), log(x*y),␣
↪sin(x/(x**Integer(2) + x)), factorial(x+Integer(1))/factorial(x)],␣
↪sparse=True)
>>> v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
>>> v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
>>> v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
>>> v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
>>> v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

>>> v = vector([sin(Integer(2)*x), sin(Integer(3)*x)], sparse=True)
>>> v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
>>> v.simplify_trig(False)
(sin(2*x), sin(3*x))
>>> v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

See Expression.simplify() for optional arguments.

simplify_factorial(**args*, ***kwds*)

Generic function used to implement common symbolic operations elementwise as methods of a vector.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: v = vector([sin(x)^2 + cos(x)^2, log(x*y), sin(x/(x^2 + x)),␣
↪factorial(x+1)/factorial(x)], sparse=True)
sage: v.simplify_trig()
```

(continues on next page)

(continued from previous page)

```
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
sage: v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
sage: v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
sage: v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
sage: v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

sage: v = vector([sin(2*x), sin(3*x)], sparse=True)
sage: v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
sage: v.simplify_trig(False)
(sin(2*x), sin(3*x))
sage: v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> v = vector([sin(x)**Integer(2) + cos(x)**Integer(2), log(x*y),␣
↪sin(x/(x**Integer(2) + x)), factorial(x+Integer(1))/factorial(x)],␣
↪sparse=True)
>>> v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
>>> v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
>>> v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
>>> v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
>>> v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

>>> v = vector([sin(Integer(2)*x), sin(Integer(3)*x)], sparse=True)
>>> v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
>>> v.simplify_trig(False)
(sin(2*x), sin(3*x))
>>> v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

See Expression.simplify_factorial() for optional arguments.

simplify_full(**args*, ***kwds*)

Generic function used to implement common symbolic operations elementwise as methods of a vector.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: v = vector([sin(x)^2 + cos(x)^2, log(x*y), sin(x/(x^2 + x)),␣
↪factorial(x+1)/factorial(x)], sparse=True)
sage: v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
sage: v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
sage: v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
sage: v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
sage: v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

sage: v = vector([sin(2*x), sin(3*x)], sparse=True)
sage: v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
sage: v.simplify_trig(False)
(sin(2*x), sin(3*x))
sage: v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> v = vector([sin(x)**Integer(2) + cos(x)**Integer(2), log(x*y),␣
↪sin(x/(x**Integer(2) + x)), factorial(x+Integer(1))/factorial(x)],␣
↪sparse=True)
>>> v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
>>> v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
>>> v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
>>> v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
>>> v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

>>> v = vector([sin(Integer(2)*x), sin(Integer(3)*x)], sparse=True)
>>> v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
>>> v.simplify_trig(False)
(sin(2*x), sin(3*x))
>>> v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

See Expression.simplify_full() for optional arguments.

simplify_log(**args*, ***kwds*)

Generic function used to implement common symbolic operations elementwise as methods of a vector.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: v = vector([sin(x)^2 + cos(x)^2, log(x*y), sin(x/(x^2 + x)),␣
↪factorial(x+1)/factorial(x)], sparse=True)
sage: v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
sage: v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
sage: v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
sage: v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
sage: v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

sage: v = vector([sin(2*x), sin(3*x)], sparse=True)
sage: v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
sage: v.simplify_trig(False)
(sin(2*x), sin(3*x))
sage: v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> v = vector([sin(x)**Integer(2) + cos(x)**Integer(2), log(x*y),␣
↪sin(x/(x**Integer(2) + x)), factorial(x+Integer(1))/factorial(x)],␣
↪sparse=True)
>>> v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
>>> v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
>>> v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
>>> v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
>>> v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

>>> v = vector([sin(Integer(2)*x), sin(Integer(3)*x)], sparse=True)
```

(continues on next page)

(continued from previous page)

```
>>> v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
>>> v.simplify_trig(False)
(sin(2*x), sin(3*x))
>>> v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

See Expression.simplify_log() for optional arguments.

simplify_rational(**args*, ***kwds*)

Generic function used to implement common symbolic operations elementwise as methods of a vector.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: v = vector([sin(x)^2 + cos(x)^2, log(x*y), sin(x/(x^2 + x)),␣
↪factorial(x+1)/factorial(x)], sparse=True)
sage: v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
sage: v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
sage: v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
sage: v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
sage: v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

sage: v = vector([sin(2*x), sin(3*x)], sparse=True)
sage: v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
sage: v.simplify_trig(False)
(sin(2*x), sin(3*x))
sage: v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> v = vector([sin(x)**Integer(2) + cos(x)**Integer(2), log(x*y),␣
↪sin(x/(x**Integer(2) + x)), factorial(x+Integer(1))/factorial(x)],␣
↪sparse=True)
>>> v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
>>> v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
>>> v.simplify_rational()
```

(continues on next page)

(continued from previous page)

```
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
>>> v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
>>> v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

>>> v = vector([sin(Integer(2)*x), sin(Integer(3)*x)], sparse=True)
>>> v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
>>> v.simplify_trig(False)
(sin(2*x), sin(3*x))
>>> v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

See Expression.simplify_rational() for optional arguments.

simplify_trig(**args*, ***kwds*)

Generic function used to implement common symbolic operations elementwise as methods of a vector.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: v = vector([sin(x)^2 + cos(x)^2, log(x*y), sin(x/(x^2 + x)),␣
↪factorial(x+1)/factorial(x)], sparse=True)
sage: v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
sage: v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
sage: v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
sage: v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
sage: v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

sage: v = vector([sin(2*x), sin(3*x)], sparse=True)
sage: v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
sage: v.simplify_trig(False)
(sin(2*x), sin(3*x))
sage: v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> v = vector([sin(x)**Integer(2) + cos(x)**Integer(2), log(x*y),␣
```

(continues on next page)

(continued from previous page)

```
↪sin(x/(x**Integer(2) + x)), factorial(x+Integer(1))/factorial(x)],␣
↪sparse=True)
>>> v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
>>> v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
>>> v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
>>> v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
>>> v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

>>> v = vector([sin(Integer(2)*x), sin(Integer(3)*x)], sparse=True)
>>> v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
>>> v.simplify_trig(False)
(sin(2*x), sin(3*x))
>>> v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

See Expression.simplify_trig() for optional arguments.

trig_expand(**args*, ***kwds*)

Generic function used to implement common symbolic operations elementwise as methods of a vector.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: v = vector([sin(x)^2 + cos(x)^2, log(x*y), sin(x/(x^2 + x)),␣
↪factorial(x+1)/factorial(x)], sparse=True)
sage: v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
sage: v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
sage: v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
sage: v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
sage: v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

sage: v = vector([sin(2*x), sin(3*x)], sparse=True)
sage: v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
sage: v.simplify_trig(False)
```

(continues on next page)

(continued from previous page)

```
(sin(2*x), sin(3*x))
sage: v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> v = vector([sin(x)**Integer(2) + cos(x)**Integer(2), log(x*y),␣
↪sin(x/(x**Integer(2) + x)), factorial(x+Integer(1))/factorial(x)],␣
↪sparse=True)
>>> v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
>>> v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
>>> v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
>>> v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
>>> v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

>>> v = vector([sin(Integer(2)*x), sin(Integer(3)*x)], sparse=True)
>>> v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
>>> v.simplify_trig(False)
(sin(2*x), sin(3*x))
>>> v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

See Expression.expand_trig() for optional arguments.

trig_reduce(**args*, ***kwds*)

Generic function used to implement common symbolic operations elementwise as methods of a vector.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: v = vector([sin(x)^2 + cos(x)^2, log(x*y), sin(x/(x^2 + x)),␣
↪factorial(x+1)/factorial(x)], sparse=True)
sage: v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
sage: v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
sage: v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
sage: v.simplify_factorial()
```

(continues on next page)

(continued from previous page)

```
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
sage: v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

sage: v = vector([sin(2*x), sin(3*x)], sparse=True)
sage: v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
sage: v.simplify_trig(False)
(sin(2*x), sin(3*x))
sage: v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> v = vector([sin(x)**Integer(2) + cos(x)**Integer(2), log(x*y),␣
↪sin(x/(x**Integer(2) + x)), factorial(x+Integer(1))/factorial(x)],␣
↪sparse=True)
>>> v.simplify_trig()
(1, log(x*y), sin(1/(x + 1)), factorial(x + 1)/factorial(x))
>>> v.canonicalize_radical()
(cos(x)^2 + sin(x)^2, log(x) + log(y), sin(1/(x + 1)), factorial(x +␣
↪1)/factorial(x))
>>> v.simplify_rational()
(cos(x)^2 + sin(x)^2, log(x*y), sin(1/(x + 1)), factorial(x + 1)/
↪factorial(x))
>>> v.simplify_factorial()
(cos(x)^2 + sin(x)^2, log(x*y), sin(x/(x^2 + x)), x + 1)
>>> v.simplify_full()
(1, log(x*y), sin(1/(x + 1)), x + 1)

>>> v = vector([sin(Integer(2)*x), sin(Integer(3)*x)], sparse=True)
>>> v.simplify_trig()
(2*cos(x)*sin(x), (4*cos(x)^2 - 1)*sin(x))
>>> v.simplify_trig(False)
(sin(2*x), sin(3*x))
>>> v.simplify_trig(expand=False)
(sin(2*x), sin(3*x))
```

See Expression.reduce_trig() for optional arguments.

sage.modules.vector_symbolic_sparse.**apply_map**(*phi*)

Return a function that applies `phi` to its argument.

EXAMPLES:

```
sage: from sage.modules.vector_symbolic_sparse import apply_map
sage: v = vector([1,2,3], sparse=True)
sage: f = apply_map(lambda x: x+1)
sage: f(v)
(2, 3, 4)
```

```
>>> from sage.all import *
>>> from sage.modules.vector_symbolic_sparse import apply_map
>>> v = vector([Integer(1),Integer(2),Integer(3)], sparse=True)
>>> f = apply_map(lambda x: x+Integer(1))
>>> f(v)
(2, 3, 4)
```

9.10 Vectors over callable symbolic rings

AUTHOR:

- Jason Grout (2010)

EXAMPLES:

```
sage: f(r, theta, z) = (r*cos(theta), r*sin(theta), z)
sage: f.parent()
Vector space of dimension 3 over Callable function ring with arguments (r, theta, z)
sage: f
(r, theta, z) |--> (r*cos(theta), r*sin(theta), z)
sage: f[0]
(r, theta, z) |--> r*cos(theta)
sage: f+f
(r, theta, z) |--> (2*r*cos(theta), 2*r*sin(theta), 2*z)
sage: 3*f
(r, theta, z) |--> (3*r*cos(theta), 3*r*sin(theta), 3*z)
sage: f*f # dot product
(r, theta, z) |--> r^2*cos(theta)^2 + r^2*sin(theta)^2 + z^2
sage: f.diff()(0,1,2) # the matrix derivative
[cos(1)      0      0]
[sin(1)      0      0]
[     0      0      1]
```

```
>>> from sage.all import *
>>> __tmp__=var("r,theta,z"); f = symbolic_expression((r*cos(theta), r*sin(theta),␣
↪z)).function(r,theta,z)
>>> f.parent()
Vector space of dimension 3 over Callable function ring with arguments (r, theta, z)
>>> f
(r, theta, z) |--> (r*cos(theta), r*sin(theta), z)
>>> f[Integer(0)]
(r, theta, z) |--> r*cos(theta)
>>> f+f
(r, theta, z) |--> (2*r*cos(theta), 2*r*sin(theta), 2*z)
>>> Integer(3)*f
(r, theta, z) |--> (3*r*cos(theta), 3*r*sin(theta), 3*z)
>>> f*f # dot product
(r, theta, z) |--> r^2*cos(theta)^2 + r^2*sin(theta)^2 + z^2
>>> f.diff()(Integer(0),Integer(1),Integer(2)) # the matrix derivative
[cos(1)      0      0]
[sin(1)      0      0]
[     0      0      1]
```

class sage.modules.vector_callable_symbolic_dense.**Vector_callable_symbolic_dense**

Bases: *FreeModuleElement_generic_dense*

9.11 Dense vectors using a NumPy backend

This serves as a base class for dense vectors over Real Double Field and Complex Double Field

EXAMPLES:

```
sage: # needs sage.symbolic
sage: v = vector(CDF,[(1,-1), (2,pi), (3,5)]); v
(1.0 - 1.0*I, 2.0 + 3.141592653589793*I, 3.0 + 5.0*I)
sage: type(v)
<class 'sage.modules.vector_complex_double_dense.Vector_complex_double_dense'>
sage: parent(v)
Vector space of dimension 3 over Complex Double Field
sage: v[0] = 5
sage: v
(5.0, 2.0 + 3.141592653589793*I, 3.0 + 5.0*I)
sage: loads(dumps(v)) == v
True

sage: v = vector(RDF, [1,2,3,4]); v
(1.0, 2.0, 3.0, 4.0)
sage: loads(dumps(v)) == v
True
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> v = vector(CDF,[(Integer(1),-Integer(1)), (Integer(2),pi), (Integer(3),
↪Integer(5))]); v
(1.0 - 1.0*I, 2.0 + 3.141592653589793*I, 3.0 + 5.0*I)
>>> type(v)
<class 'sage.modules.vector_complex_double_dense.Vector_complex_double_dense'>
>>> parent(v)
Vector space of dimension 3 over Complex Double Field
>>> v[Integer(0)] = Integer(5)
>>> v
(5.0, 2.0 + 3.141592653589793*I, 3.0 + 5.0*I)
>>> loads(dumps(v)) == v
True

>>> v = vector(RDF, [Integer(1),Integer(2),Integer(3),Integer(4)]); v
(1.0, 2.0, 3.0, 4.0)
>>> loads(dumps(v)) == v
True
```

AUTHORS:

- Jason Grout, Oct 2008: switch to numpy backend, factored out `Vector_double_dense` class
- Josh Kantor
- William Stein

class sage.modules.vector_double_dense.**Vector_double_dense**

Bases: *Vector_numpy_dense*

Base class for vectors over the Real Double Field and the Complex Double Field. These are supposed to be fast vector operations using C doubles. Most operations are implemented using numpy which will call the underlying BLAS, if needed, on the system.

This class cannot be instantiated on its own. The numpy vector creation depends on several variables that are set in the subclasses.

EXAMPLES:

```
sage: v = vector(RDF, [1,2,3,4]); v
(1.0, 2.0, 3.0, 4.0)
sage: v*v
30.0
```

```
>>> from sage.all import *
>>> v = vector(RDF, [Integer(1),Integer(2),Integer(3),Integer(4)]); v
(1.0, 2.0, 3.0, 4.0)
>>> v*v
30.0
```

complex_vector()

Return the associated complex vector, i.e., this vector but with coefficients viewed as complex numbers.

EXAMPLES:

```
sage: v = vector(RDF,4,range(4)); v
(0.0, 1.0, 2.0, 3.0)
sage: v.complex_vector()
(0.0, 1.0, 2.0, 3.0)
sage: v = vector(RDF,0)
sage: v.complex_vector()
()
```

```
>>> from sage.all import *
>>> v = vector(RDF,Integer(4),range(Integer(4))); v
(0.0, 1.0, 2.0, 3.0)
>>> v.complex_vector()
(0.0, 1.0, 2.0, 3.0)
>>> v = vector(RDF,Integer(0))
>>> v.complex_vector()
()
```

fft(*direction='forward'*, *algorithm='radix2'*, *inplace=False*)

This performs a fast Fourier transform on the vector.

INPUT:

- `direction` – string; `'forward'` (default) or `'backward'`

The algorithm and inplace arguments are ignored.

This function is fastest if the vector's length is a power of 2.

EXAMPLES:

```
sage: # needs scipy
sage: v = vector(CDF,[1+2*I,2,3*I,4])
sage: v.fft()
(7.0 + 5.0*I, 1.0 + 1.0*I, -5.0 + 5.0*I, 1.0 - 3.0*I)
sage: v.fft(direction='backward')
(1.75 + 1.25*I, 0.25 - 0.75*I, -1.25 + 1.25*I, 0.25 + 0.25*I)
sage: v.fft().fft(direction='backward')
(1.0 + 2.0*I, 2.0, 3.0*I, 4.0)
sage: v.fft().parent()
Vector space of dimension 4 over Complex Double Field
sage: v.fft(inplace=True)
sage: v
(7.0 + 5.0*I, 1.0 + 1.0*I, -5.0 + 5.0*I, 1.0 - 3.0*I)

sage: # needs scipy
sage: v = vector(RDF,4,range(4)); v
(0.0, 1.0, 2.0, 3.0)
sage: v.fft()
(6.0, -2.0 + 2.0*I, -2.0, -2.0 - 2.0*I)
sage: v.fft(direction='backward')
(1.5, -0.5 - 0.5*I, -0.5, -0.5 + 0.5*I)
sage: v.fft().fft(direction='backward')
(0.0, 1.0, 2.0, 3.0)
sage: v.fft().parent()
Vector space of dimension 4 over Complex Double Field
sage: v.fft(inplace=True)
Traceback (most recent call last):
...
ValueError: inplace can only be True for CDF vectors
```

```
>>> from sage.all import *
>>> # needs scipy
>>> v = vector(CDF,[Integer(1)+Integer(2)*I,Integer(2),Integer(3)*I,
↪Integer(4)])
>>> v.fft()
(7.0 + 5.0*I, 1.0 + 1.0*I, -5.0 + 5.0*I, 1.0 - 3.0*I)
>>> v.fft(direction='backward')
(1.75 + 1.25*I, 0.25 - 0.75*I, -1.25 + 1.25*I, 0.25 + 0.25*I)
>>> v.fft().fft(direction='backward')
(1.0 + 2.0*I, 2.0, 3.0*I, 4.0)
>>> v.fft().parent()
Vector space of dimension 4 over Complex Double Field
>>> v.fft(inplace=True)
>>> v
(7.0 + 5.0*I, 1.0 + 1.0*I, -5.0 + 5.0*I, 1.0 - 3.0*I)

>>> # needs scipy
>>> v = vector(RDF,Integer(4),range(Integer(4))); v
(0.0, 1.0, 2.0, 3.0)
>>> v.fft()
(6.0, -2.0 + 2.0*I, -2.0, -2.0 - 2.0*I)
>>> v.fft(direction='backward')
```

(continues on next page)

(continued from previous page)

```
(1.5, -0.5 - 0.5*I, -0.5, -0.5 + 0.5*I)
>>> v.fft().fft(direction='backward')
(0.0, 1.0, 2.0, 3.0)
>>> v.fft().parent()
Vector space of dimension 4 over Complex Double Field
>>> v.fft(inplace=True)
Traceback (most recent call last):
...
ValueError: inplace can only be True for CDF vectors
```

inv_fft(*algorithm='radix2'*, *inplace=False*)

This performs the inverse fast Fourier transform on the vector.

The Fourier transform can be done in place using the keyword inplace=True

This will be fastest if the vector's length is a power of 2.

EXAMPLES:

```
sage: # needs scipy
sage: v = vector(CDF,[1,2,3,4])
sage: w = v.fft()
sage: max(v - w.inv_fft()) < 1e-12
True
```

```
>>> from sage.all import *
>>> # needs scipy
>>> v = vector(CDF,[Integer(1),Integer(2),Integer(3),Integer(4)])
>>> w = v.fft()
>>> max(v - w.inv_fft()) < RealNumber('1e-12')
True
```

mean()

Calculate the arithmetic mean of the vector.

EXAMPLES:

```
sage: v = vector(RDF, range(9))
sage: w = vector(CDF, [k+(9-k)*I for k in range(9)])
sage: v.mean()
4.0
sage: w.mean()
4.0 + 5.0*I
```

```
>>> from sage.all import *
>>> v = vector(RDF, range(Integer(9)))
>>> w = vector(CDF, [k+(Integer(9)-k)*I for k in range(Integer(9))])
>>> v.mean()
4.0
>>> w.mean()
4.0 + 5.0*I
```

norm(*p=2*)

Return the norm (or related computations) of the vector.

INPUT:

- p – (default: 2) controls which norm is computed, allowable values are any real number and positive and negative infinity. See output discussion for specifics.

OUTPUT:

Returned value is a double precision floating point value in RDF (or an integer when p=0). The default value of p = 2 is the "usual" Euclidean norm. For other values:

- p = Infinity or p = oo: the maximum of the absolute values of the entries, where the absolute value of the complex number $a + bi$ is $\sqrt{a^2 + b^2}$
- p = -Infinity or p = -oo: the minimum of the absolute values of the entries
- p = 0: the number of nonzero entries in the vector
- p any other real number: for a vector $\vec{x}$ this method computes

$$\left(\sum_i x_i^p\right)^{1/p}$$

For p < 0 this function is not a norm, but the above computation may be useful for other purposes.

ALGORITHM:

Computation is performed by the norm() function of the SciPy/NumPy library.

EXAMPLES:

First over the reals.

```
sage: v = vector(RDF, range(9))
sage: v.norm()
14.28285685...
sage: v.norm(p=2)
14.28285685...
sage: v.norm(p=6)
8.744039097...
sage: v.norm(p=Infinity)
8.0
sage: v.norm(p=-oo)
0.0
sage: v.norm(p=0)
8.0
sage: v.norm(p=0.3)
4099.153615...
```

```
>>> from sage.all import *
>>> v = vector(RDF, range(Integer(9)))
>>> v.norm()
14.28285685...
>>> v.norm(p=Integer(2))
14.28285685...
>>> v.norm(p=Integer(6))
8.744039097...
>>> v.norm(p=Infinity)
8.0
```

(continues on next page)

(continued from previous page)

```
>>> v.norm(p=-oo)
0.0
>>> v.norm(p=Integer(0))
8.0
>>> v.norm(p=RealNumber('0.3'))
4099.153615...
```

And over the complex numbers.

```
sage: w = vector(CDF, [3-4*I, 0, 5+12*I])
sage: w.norm()
13.9283882...
sage: w.norm(p=2)
13.9283882...
sage: w.norm(p=0)
2.0
sage: w.norm(p=4.2)
13.0555695...
sage: w.norm(p=oo)
13.0
```

```
>>> from sage.all import *
>>> w = vector(CDF, [Integer(3)-Integer(4)*I, Integer(0),␣
↪Integer(5)+Integer(12)*I])
>>> w.norm()
13.9283882...
>>> w.norm(p=Integer(2))
13.9283882...
>>> w.norm(p=Integer(0))
2.0
>>> w.norm(p=RealNumber('4.2'))
13.0555695...
>>> w.norm(p=oo)
13.0
```

Negative values of `p` are allowed and will provide the same computation as for positive values. A zero entry in the vector will raise a warning and return zero.

```
sage: v = vector(CDF, range(1,10))
sage: v.norm(p=-3.2)
0.953760808...
sage: w = vector(CDF, [-1,0,1])
sage: w.norm(p=-1.6)
doctest:...: RuntimeWarning: divide by zero encountered in power
0.0
```

```
>>> from sage.all import *
>>> v = vector(CDF, range(Integer(1),Integer(10)))
>>> v.norm(p=-RealNumber('3.2'))
0.953760808...
>>> w = vector(CDF, [-Integer(1),Integer(0),Integer(1)])
>>> w.norm(p=-RealNumber('1.6'))
```

(continues on next page)

(continued from previous page)

```
doctest:...: RuntimeWarning: divide by zero encountered in power
0.0
```

Return values are in `RDF`, or an integer when `p = 0`.

```
sage: v = vector(RDF, [1,2,4,8])
sage: v.norm() in RDF
True
sage: v.norm(p=0) in ZZ
True
```

```
>>> from sage.all import *
>>> v = vector(RDF, [Integer(1),Integer(2),Integer(4),Integer(8)])
>>> v.norm() in RDF
True
>>> v.norm(p=Integer(0)) in ZZ
True
```

Improper values of `p` are caught.

```
sage: w = vector(CDF, [-1,0,1])
sage: w.norm(p='junk')
Traceback (most recent call last):
...
ValueError: vector norm 'p' must be +/- infinity or a real number, not junk
```

```
>>> from sage.all import *
>>> w = vector(CDF, [-Integer(1),Integer(0),Integer(1)])
>>> w.norm(p='junk')
Traceback (most recent call last):
...
ValueError: vector norm 'p' must be +/- infinity or a real number, not junk
```

prod()

Return the product of the entries of `self`.

EXAMPLES:

```
sage: v = vector(RDF, range(9))
sage: w = vector(CDF, [k+(9-k)*I for k in range(9)])
sage: v.prod()
0.0
sage: w.prod()
57204225.0*I
```

```
>>> from sage.all import *
>>> v = vector(RDF, range(Integer(9)))
>>> w = vector(CDF, [k+(Integer(9)-k)*I for k in range(Integer(9))])
>>> v.prod()
0.0
>>> w.prod()
57204225.0*I
```

standard_deviation(*population=True*)

Calculate the standard deviation of entries of the vector.

INPUT:

- `population` – If `False`, calculate the sample standard deviation

EXAMPLES:

```
sage: v = vector(RDF, range(9))
sage: w = vector(CDF, [k+(9-k)*I for k in range(9)])
sage: v.standard_deviation()
2.7386127875258306
sage: v.standard_deviation(population=False)
2.581988897471611
sage: w.standard_deviation()
3.872983346207417
sage: w.standard_deviation(population=False)
3.6514837167011076
```

```
>>> from sage.all import *
>>> v = vector(RDF, range(Integer(9)))
>>> w = vector(CDF, [k+(Integer(9)-k)*I for k in range(Integer(9))])
>>> v.standard_deviation()
2.7386127875258306
>>> v.standard_deviation(population=False)
2.581988897471611
>>> w.standard_deviation()
3.872983346207417
>>> w.standard_deviation(population=False)
3.6514837167011076
```

stats_kurtosis()

Compute the kurtosis of a dataset.

Kurtosis is the fourth central moment divided by the square of the variance. Since we use Fisher's definition, 3.0 is subtracted from the result to give 0.0 for a normal distribution. (Paragraph from the scipy.stats docstring.)

EXAMPLES:

```
sage: # needs scipy
sage: v = vector(RDF, range(9))
sage: w = vector(CDF, [k+(9-k)*I for k in range(9)])
sage: v.stats_kurtosis()  # rel tol 5e-15
-1.2300000000000000
sage: w.stats_kurtosis()  # rel tol 5e-15
-1.2300000000000000
```

```
>>> from sage.all import *
>>> # needs scipy
>>> v = vector(RDF, range(Integer(9)))
>>> w = vector(CDF, [k+(Integer(9)-k)*I for k in range(Integer(9))])
>>> v.stats_kurtosis()  # rel tol 5e-15
-1.2300000000000000
```

(continues on next page)

(continued from previous page)

```
>>> w.stats_kurtosis()  # rel tol 5e-15
-1.2300000000000000
```

sum()

Return the sum of the entries of `self`.

EXAMPLES:

```
sage: v = vector(RDF, range(9))
sage: w = vector(CDF, [k+(9-k)*I for k in range(9)])
sage: v.sum()
36.0
sage: w.sum()
36.0 + 45.0*I
```

```
>>> from sage.all import *
>>> v = vector(RDF, range(Integer(9)))
>>> w = vector(CDF, [k+(Integer(9)-k)*I for k in range(Integer(9))])
>>> v.sum()
36.0
>>> w.sum()
36.0 + 45.0*I
```

variance(*population=True*)

Calculate the variance of entries of the vector.

INPUT:

- `population` – if `False`, calculate the sample variance

EXAMPLES:

```
sage: v = vector(RDF, range(9))
sage: w = vector(CDF, [k+(9-k)*I for k in range(9)])
sage: v.variance()
7.5
sage: v.variance(population=False)
6.666666666666667
sage: w.variance()
15.0
sage: w.variance(population=False)
13.333333333333334
```

```
>>> from sage.all import *
>>> v = vector(RDF, range(Integer(9)))
>>> w = vector(CDF, [k+(Integer(9)-k)*I for k in range(Integer(9))])
>>> v.variance()
7.5
>>> v.variance(population=False)
6.666666666666667
>>> w.variance()
15.0
>>> w.variance(population=False)
13.333333333333334
```

zero_at(*eps*)

Return a copy with small entries replaced by zeros.

This is useful for modifying output from algorithms which have large relative errors when producing zero elements, e.g. to create reliable doctests.

INPUT:

- `eps` – cutoff value

OUTPUT:

A modified copy of the vector. Elements smaller than or equal to `eps` are replaced with zeroes. For complex vectors, the real and imaginary parts are considered individually.

EXAMPLES:

```
sage: v = vector(RDF, [1.0, 2.0, 10^-10, 3.0])
sage: v.zero_at(1e-8)
(1.0, 2.0, 0.0, 3.0)
sage: v.zero_at(1e-12)
(1.0, 2.0, 1e-10, 3.0)
```

```
>>> from sage.all import *
>>> v = vector(RDF, [RealNumber('1.0'), RealNumber('2.0'), Integer(10)**-
↪Integer(10), RealNumber('3.0')])
>>> v.zero_at(RealNumber('1e-8'))
(1.0, 2.0, 0.0, 3.0)
>>> v.zero_at(RealNumber('1e-12'))
(1.0, 2.0, 1e-10, 3.0)
```

For complex numbers the real and imaginary parts are considered separately.

```
sage: w = vector(CDF, [10^-6 + 5*I, 5 + 10^-6*I, 5 + 5*I, 10^-6 + 10^-6*I])
sage: w.zero_at(1.0e-4)
(5.0*I, 5.0, 5.0 + 5.0*I, 0.0)
sage: w.zero_at(1.0e-8)
(1e-06 + 5.0*I, 5.0 + 1e-06*I, 5.0 + 5.0*I, 1e-06 + 1e-06*I)
```

```
>>> from sage.all import *
>>> w = vector(CDF, [Integer(10)**-Integer(6) + Integer(5)*I, Integer(5) +␣
↪Integer(10)**-Integer(6)*I, Integer(5) + Integer(5)*I, Integer(10)**-
↪Integer(6) + Integer(10)**-Integer(6)*I])
>>> w.zero_at(RealNumber('1.0e-4'))
(5.0*I, 5.0, 5.0 + 5.0*I, 0.0)
>>> w.zero_at(RealNumber('1.0e-8'))
(1e-06 + 5.0*I, 5.0 + 1e-06*I, 5.0 + 5.0*I, 1e-06 + 1e-06*I)
```

9.12 Dense real double vectors using a NumPy backend

EXAMPLES:

```
sage: # needs sage.symbolic
sage: v = vector(RDF, [1, pi, sqrt(2)]); v
(1.0, 3.141592653589793, 1.414213562373095)
```

(continues on next page)

(continued from previous page)

```
sage: type(v)
<class 'sage.modules.vector_real_double_dense.Vector_real_double_dense'>
sage: parent(v)
Vector space of dimension 3 over Real Double Field
sage: v[0] = 5
sage: v
(5.0, 3.141592653589793, 1.414213562373095)
sage: loads(dumps(v)) == v
True
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> v = vector(RDF, [Integer(1), pi, sqrt(Integer(2))]); v
(1.0, 3.141592653589793, 1.414213562373095)
>>> type(v)
<class 'sage.modules.vector_real_double_dense.Vector_real_double_dense'>
>>> parent(v)
Vector space of dimension 3 over Real Double Field
>>> v[Integer(0)] = Integer(5)
>>> v
(5.0, 3.141592653589793, 1.414213562373095)
>>> loads(dumps(v)) == v
True
```

AUTHORS:

- **Jason Grout, Oct 2008: switch to numpy backend, factored out**
  Vector_double_dense class

class sage.modules.vector_real_double_dense.**Vector_real_double_dense**

Bases: *Vector_double_dense*

Vectors over the Real Double Field. These are supposed to be fast vector operations using C doubles. Most operations are implemented using numpy which will call the underlying BLAS, if needed, on the system.

EXAMPLES:

```
sage: v = vector(RDF, [1,2,3,4]); v
(1.0, 2.0, 3.0, 4.0)
sage: v*v
30.0
```

```
>>> from sage.all import *
>>> v = vector(RDF, [Integer(1),Integer(2),Integer(3),Integer(4)]); v
(1.0, 2.0, 3.0, 4.0)
>>> v*v
30.0
```

stats_skew()

Compute the skewness of a data set.

For normally distributed data, the skewness should be about 0. A skewness value > 0 means that there is more weight in the left tail of the distribution. (Paragraph from the scipy.stats docstring.)

EXAMPLES:

```
sage: v = vector(RDF, range(9))
sage: v.stats_skew()                                                    #␣
↪needs scipy
0.0
```

```
>>> from sage.all import *
>>> v = vector(RDF, range(Integer(9)))
>>> v.stats_skew()                                                      #␣
↪needs scipy
0.0
```

sage.modules.vector_real_double_dense.**unpickle_v0**(*parent*, *entries*, *degree*)

Create a real double vector containing the entries.

EXAMPLES:

```
sage: v = vector(RDF, [1,2,3])
sage: w = sage.modules.vector_real_double_dense.unpickle_v0(v.parent(), list(v),␣
↪v.degree())
sage: v == w
True
```

```
>>> from sage.all import *
>>> v = vector(RDF, [Integer(1),Integer(2),Integer(3)])
>>> w = sage.modules.vector_real_double_dense.unpickle_v0(v.parent(), list(v), v.
↪degree())
>>> v == w
True
```

sage.modules.vector_real_double_dense.**unpickle_v1**(*parent*, *entries*, *degree*, *is_mutable=None*)

Create a real double vector with the given parent, entries, degree, and mutability.

EXAMPLES:

```
sage: v = vector(RDF, [1,2,3])
sage: w = sage.modules.vector_real_double_dense.unpickle_v1(v.parent(), list(v),␣
↪v.degree(), v.is_immutable())
sage: v == w
True
```

```
>>> from sage.all import *
>>> v = vector(RDF, [Integer(1),Integer(2),Integer(3)])
>>> w = sage.modules.vector_real_double_dense.unpickle_v1(v.parent(), list(v), v.
↪degree(), v.is_immutable())
>>> v == w
True
```

9.13 Dense vectors using a NumPy backend.

AUTHORS:

- Jason Grout, Oct 2008: switch to numpy backend, factored out `Vector_double_dense` class

- Josh Kantor
- William Stein

class sage.modules.vector_numpy_dense.**Vector_numpy_dense**

Bases: *FreeModuleElement*

Base class for vectors implemented using numpy arrays.

This class cannot be instantiated on its own. The numpy vector creation depends on several variables that are set in the subclasses.

EXAMPLES:

```
sage: v = vector(RDF, [1,2,3,4]); v
(1.0, 2.0, 3.0, 4.0)
sage: v*v
30.0
```

```
>>> from sage.all import *
>>> v = vector(RDF, [Integer(1),Integer(2),Integer(3),Integer(4)]); v
(1.0, 2.0, 3.0, 4.0)
>>> v*v
30.0
```

numpy(*dtype=None*)

Return numpy array corresponding to this vector.

INPUT:

- `dtype` – if specified, the numpy dtype of the returned array

EXAMPLES:

```
sage: v = vector(CDF,4,range(4))
sage: v.numpy()
array([0.+0.j, 1.+0.j, 2.+0.j, 3.+0.j])
sage: v = vector(CDF,0)
sage: v.numpy()
array([], dtype=complex128)
sage: v = vector(RDF,4,range(4))
sage: v.numpy()
array([0., 1., 2., 3.])
sage: v = vector(RDF,0)
sage: v.numpy()
array([], dtype=float64)
```

```
>>> from sage.all import *
>>> v = vector(CDF,Integer(4),range(Integer(4)))
>>> v.numpy()
array([0.+0.j, 1.+0.j, 2.+0.j, 3.+0.j])
>>> v = vector(CDF,Integer(0))
>>> v.numpy()
array([], dtype=complex128)
>>> v = vector(RDF,Integer(4),range(Integer(4)))
>>> v.numpy()
array([0., 1., 2., 3.])
```

(continues on next page)

(continued from previous page)

```
>>> v = vector(RDF,Integer(0))
>>> v.numpy()
array([], dtype=float64)
```

A numpy dtype may be requested manually:

```
sage: import numpy
sage: v = vector(CDF, 3, range(3))
sage: v.numpy()
array([0.+0.j, 1.+0.j, 2.+0.j])
sage: v.numpy(dtype=numpy.float64)
array([0., 1., 2.])
sage: v.numpy(dtype=numpy.float32)
array([0., 1., 2.], dtype=float32)
```

```
>>> from sage.all import *
>>> import numpy
>>> v = vector(CDF, Integer(3), range(Integer(3)))
>>> v.numpy()
array([0.+0.j, 1.+0.j, 2.+0.j])
>>> v.numpy(dtype=numpy.float64)
array([0., 1., 2.])
>>> v.numpy(dtype=numpy.float32)
array([0., 1., 2.], dtype=float32)
```

9.14 Dense integer vectors using a NumPy backend.

EXAMPLES:

```
sage: from sage.modules.vector_numpy_integer_dense import Vector_numpy_integer_dense
sage: v = Vector_numpy_integer_dense(FreeModule(ZZ, 3), [0, 0, 0]); v
(0, 0, 0)
sage: v[1] = 42
sage: v
(0, 42, 0)
sage: v.numpy()
array([ 0, 42,  0])               # 64-bit
array([ 0, 42,  0], dtype=int64)  # 32-bit
```

```
>>> from sage.all import *
>>> from sage.modules.vector_numpy_integer_dense import Vector_numpy_integer_dense
>>> v = Vector_numpy_integer_dense(FreeModule(ZZ, Integer(3)), [Integer(0),␣
↪Integer(0), Integer(0)]); v
(0, 0, 0)
>>> v[Integer(1)] = Integer(42)
>>> v
(0, 42, 0)
>>> v.numpy()
array([ 0, 42,  0])               # 64-bit
array([ 0, 42,  0], dtype=int64)  # 32-bit
```

class sage.modules.vector_numpy_integer_dense.**Vector_numpy_integer_dense**

Bases: *Vector_numpy_dense*

9.15 Dense complex double vectors using a NumPy backend

EXAMPLES:

```
sage: # needs sage.symbolic
sage: v = vector(CDF, [(1,-1), (2,pi), (3,5)]); v
(1.0 - 1.0*I, 2.0 + 3.141592653589793*I, 3.0 + 5.0*I)
sage: type(v)
<class 'sage.modules.vector_complex_double_dense.Vector_complex_double_dense'>
sage: parent(v)
Vector space of dimension 3 over Complex Double Field
sage: v[0] = 5
sage: v
(5.0, 2.0 + 3.141592653589793*I, 3.0 + 5.0*I)
sage: loads(dumps(v)) == v
True
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> v = vector(CDF, [(Integer(1),-Integer(1)), (Integer(2),pi), (Integer(3),
↪Integer(5))]); v
(1.0 - 1.0*I, 2.0 + 3.141592653589793*I, 3.0 + 5.0*I)
>>> type(v)
<class 'sage.modules.vector_complex_double_dense.Vector_complex_double_dense'>
>>> parent(v)
Vector space of dimension 3 over Complex Double Field
>>> v[Integer(0)] = Integer(5)
>>> v
(5.0, 2.0 + 3.141592653589793*I, 3.0 + 5.0*I)
>>> loads(dumps(v)) == v
True
```

AUTHORS:

- **Jason Grout, Oct 2008: switch to NumPy backend, factored out** Vector_double_dense class

class sage.modules.vector_complex_double_dense.**Vector_complex_double_dense**

Bases: *Vector_double_dense*

Vectors over the Complex Double Field. These are supposed to be fast vector operations using C doubles. Most operations are implemented using numpy which will call the underlying BLAS, if needed, on the system.

EXAMPLES:

```
sage: v = vector(CDF, [(1,-1), (2,pi), (3,5)]); v                              #␣
↪needs sage.symbolic
(1.0 - 1.0*I, 2.0 + 3.141592653589793*I, 3.0 + 5.0*I)
sage: v*v  # rel tol 1e-15                                                     #␣
↪needs sage.symbolic
-21.86960440108936 + 40.56637061435917*I
```

```
>>> from sage.all import *
>>> v = vector(CDF, [(Integer(1),-Integer(1)), (Integer(2),pi), (Integer(3),
↪Integer(5))]); v                                  # needs sage.symbolic
(1.0 - 1.0*I, 2.0 + 3.141592653589793*I, 3.0 + 5.0*I)
>>> v*v  # rel tol 1e-15                                                    #␣
↪needs sage.symbolic
-21.86960440108936 + 40.56637061435917*I
```

sage.modules.vector_complex_double_dense.**unpickle_v0**(*parent*, *entries*, *degree*)

Create a complex double vector containing the entries.

EXAMPLES:

```
sage: v = vector(CDF, [1,2,3])
sage: w = sage.modules.vector_complex_double_dense.unpickle_v0(v.parent(),␣
↪list(v), v.degree())
sage: v == w
True
```

```
>>> from sage.all import *
>>> v = vector(CDF, [Integer(1),Integer(2),Integer(3)])
>>> w = sage.modules.vector_complex_double_dense.unpickle_v0(v.parent(), list(v),␣
↪v.degree())
>>> v == w
True
```

sage.modules.vector_complex_double_dense.**unpickle_v1**(*parent*, *entries*, *degree*, *is_mutable=None*)

Create a complex double vector with the given parent, entries, degree, and mutability.

EXAMPLES:

```
sage: v = vector(CDF, [1,2,3])
sage: w = sage.modules.vector_complex_double_dense.unpickle_v1(v.parent(),␣
↪list(v), v.degree(), v.is_immutable())
sage: v == w
True
```

```
>>> from sage.all import *
>>> v = vector(CDF, [Integer(1),Integer(2),Integer(3)])
>>> w = sage.modules.vector_complex_double_dense.unpickle_v1(v.parent(), list(v),␣
↪v.degree(), v.is_immutable())
>>> v == w
True
```

9.16 Pickling for the old CDF vector class

AUTHORS:

- Jason Grout

9.17 Pickling for the old RDF vector class

AUTHORS:

- Jason Grout

CHAPTER

TEN

MISC

10.1 Diamond cutting implementation

AUTHORS:

- Jan Poeschko (2012-07-02): initial version

sage.modules.diamond_cutting.**calculate_voronoi_cell**(*basis*, *radius=None*, *verbose=False*)

Calculate the Voronoi cell of the lattice defined by basis.

INPUT:

- `basis` – embedded basis matrix of the lattice
- `radius` – radius of basis vectors to consider
- `verbose` – whether to print debug information

OUTPUT: a `Polyhedron` instance

EXAMPLES:

```
sage: from sage.modules.diamond_cutting import calculate_voronoi_cell
sage: V = calculate_voronoi_cell(matrix([[1, 0], [0, 1]]))
sage: V.volume()
1
```

```
>>> from sage.all import *
>>> from sage.modules.diamond_cutting import calculate_voronoi_cell
>>> V = calculate_voronoi_cell(matrix([[Integer(1), Integer(0)], [Integer(0),␣
↪Integer(1)]]))
>>> V.volume()
1
```

sage.modules.diamond_cutting.**diamond_cut**(*V*, *GM*, *C*, *verbose=False*)

Perform diamond cutting on polyhedron `V` with basis matrix `GM` and radius `C`.

INPUT:

- `V` – polyhedron to cut from
- `GM` – half of the basis matrix of the lattice
- `C` – radius to use in cutting algorithm
- `verbose` – boolean (default: `False`); whether to print debug information

OUTPUT: a `Polyhedron` instance

EXAMPLES:

```
sage: from sage.modules.diamond_cutting import diamond_cut
sage: V = Polyhedron([[0], [2]])
sage: GM = matrix([2])
sage: V = diamond_cut(V, GM, 4)
sage: V.vertices()
(A vertex at (2), A vertex at (0))
```

```
>>> from sage.all import *
>>> from sage.modules.diamond_cutting import diamond_cut
>>> V = Polyhedron([[Integer(0)], [Integer(2)]])
>>> GM = matrix([Integer(2)])
>>> V = diamond_cut(V, GM, Integer(4))
>>> V.vertices()
(A vertex at (2), A vertex at (0))
```

`sage.modules.diamond_cutting.`**`jacobi`**`(`M`)`

Compute the upper-triangular part of the Cholesky/Jacobi decomposition of the symmetric matrix `M`.

Let M be a symmetric $n \times n$-matrix over a field F. Let $m_{i,j}$ denote the (i,j)-th entry of M for any $1 \leq i \leq n$ and $1 \leq j \leq n$. Then, the upper-triangular part computed by this method is the upper-triangular $n \times n$-matrix Q whose (i,j)-th entry $q_{i,j}$ satisfies

$$q_{i,j} = \begin{cases} \frac{1}{q_{i,i}} \left(m_{i,j} - \sum_{r<i} q_{r,r} q_{r,i} q_{r,j} \right) & i < j, \\ a_{i,j} - \sum_{r<i} q_{r,r} q_{r,i}^2 & i = j, \\ 0 & i > j, \end{cases}$$

for all $1 \leq i \leq n$ and $1 \leq j \leq n$. (These equalities determine the entries of Q uniquely by recursion.) This matrix Q is defined for all M in a certain Zariski-dense open subset of the set of all $n \times n$-matrices.

Note

This should be a method of matrices.

EXAMPLES:

```
sage: from sage.modules.diamond_cutting import jacobi
sage: jacobi(identity_matrix(3) * 4)
[4 0 0]
[0 4 0]
[0 0 4]

sage: def testall(M):
....:       Q = jacobi(M)
....:       for j in range(3):
....:           for i in range(j):
....:               if Q[i,j] * Q[i,i] != M[i,j] - sum(Q[r,i] * Q[r,j] * Q[r,r]␣
↪for r in range(i)):
....:                   return False
....:       for i in range(3):
```

(continues on next page)

(continued from previous page)

```
....:           if Q[i,i] != M[i,i] - sum(Q[r,i] ** 2 * Q[r,r] for r in range(i)):
....:               return False
....:           for j in range(i):
....:               if Q[i,j] != 0:
....:                   return False
....:       return True

sage: M = Matrix(QQ, [[8,1,5], [1,6,0], [5,0,3]])
sage: Q = jacobi(M); Q
[    8   1/8   5/8]
[    0  47/8 -5/47]
[    0     0 -9/47]
sage: testall(M)
True

sage: M = Matrix(QQ, [[3,6,-1,7],[6,9,8,5],[-1,8,2,4],[7,5,4,0]])
sage: testall(M)
True
```

```
>>> from sage.all import *
>>> from sage.modules.diamond_cutting import jacobi
>>> jacobi(identity_matrix(Integer(3)) * Integer(4))
[4 0 0]
[0 4 0]
[0 0 4]

>>> def testall(M):
...       Q = jacobi(M)
...       for j in range(Integer(3)):
...           for i in range(j):
...               if Q[i,j] * Q[i,i] != M[i,j] - sum(Q[r,i] * Q[r,j] * Q[r,r] for␣
↪r in range(i)):
...                   return False
...       for i in range(Integer(3)):
...           if Q[i,i] != M[i,i] - sum(Q[r,i] ** Integer(2) * Q[r,r] for r in␣
↪range(i)):
...               return False
...           for j in range(i):
...               if Q[i,j] != Integer(0):
...                   return False
...       return True

>>> M = Matrix(QQ, [[Integer(8),Integer(1),Integer(5)], [Integer(1),Integer(6),
↪Integer(0)], [Integer(5),Integer(0),Integer(3)]])
>>> Q = jacobi(M); Q
[    8   1/8   5/8]
[    0  47/8 -5/47]
[    0     0 -9/47]
>>> testall(M)
True

>>> M = Matrix(QQ, [[Integer(3),Integer(6),-Integer(1),Integer(7)],[Integer(6),
```

(continues on next page)

(continued from previous page)

```
↪Integer(9),Integer(8),Integer(5)],[-Integer(1),Integer(8),Integer(2),
↪Integer(4)],[Integer(7),Integer(5),Integer(4),Integer(0)]])
>>> testall(M)
True
```

sage.modules.diamond_cutting.**plane_inequality**(*v*)

Return the inequality for points on the same side as the origin with respect to the plane through `v` normal to `v`.

EXAMPLES:

```
sage: from sage.modules.diamond_cutting import plane_inequality
sage: ieq = plane_inequality([1, -1]); ieq
[2, -1, 1]
sage: ieq[0] + vector(ieq[1:]) * vector([1, -1])
0
```

```
>>> from sage.all import *
>>> from sage.modules.diamond_cutting import plane_inequality
>>> ieq = plane_inequality([Integer(1), -Integer(1)]); ieq
[2, -1, 1]
>>> ieq[Integer(0)] + vector(ieq[Integer(1):]) * vector([Integer(1), -Integer(1)])
0
```

10.2 Helper classes to implement tensor operations

Warning

This module is not meant to be used directly. It just provides functionality for other classes to implement tensor operations.

The *VectorCollection* constructs the basis of tensor products (and symmetric/exterior powers) in terms of a chosen collection of vectors that generate the vector space(s).

EXAMPLES:

```
sage: from sage.modules.tensor_operations import VectorCollection, TensorOperation
sage: V = VectorCollection([(1,0), (-1, 0), (1,2)], QQ, 2)
sage: W = VectorCollection([(1,1), (1,-1), (-1, 1)], QQ, 2)
sage: VW = TensorOperation([V, W], operation='product')
```

```
>>> from sage.all import *
>>> from sage.modules.tensor_operations import VectorCollection, TensorOperation
>>> V = VectorCollection([(Integer(1),Integer(0)), (-Integer(1), Integer(0)),␣
↪(Integer(1),Integer(2))], QQ, Integer(2))
>>> W = VectorCollection([(Integer(1),Integer(1)), (Integer(1),-Integer(1)), (-
↪Integer(1), Integer(1))], QQ, Integer(2))
>>> VW = TensorOperation([V, W], operation='product')
```

Here is the tensor product of two vectors:

```
sage: V.vectors()[0]
(1, 0)
sage: W.vectors()[1]
(1, -1)
```

```
>>> from sage.all import *
>>> V.vectors()[Integer(0)]
(1, 0)
>>> W.vectors()[Integer(1)]
(1, -1)
```

In a convenient choice of basis, the tensor product is $(a, b) \otimes (c, d) = (ac, ad, bc, bd)$. In this example, it is one of the vectors of the vector collection `VW`

```
sage: VW.index_map(0, 1)
1
sage: VW.vectors()[VW.index_map(0, 1)]
(1, -1, 0, 0)

sage: rows = []
sage: for i, j in cartesian_product((range(3), range(3))):
....:     v = V.vectors()[i]
....:     w = W.vectors()[j]
....:     i_tensor_j = VW.index_map(i, j)
....:     vw = VW.vectors()[i_tensor_j]
....:     rows.append([i, v, j, w, i_tensor_j, vw])
sage: table(rows)
      0   (1, 0)    0   (1, 1)    0   (1, 1, 0, 0)
      0   (1, 0)    1   (1, -1)   1   (1, -1, 0, 0)
      0   (1, 0)    2   (-1, 1)   2   (-1, 1, 0, 0)
      1   (-1, 0)   0   (1, 1)    3   (-1, -1, 0, 0)
      1   (-1, 0)   1   (1, -1)   2   (-1, 1, 0, 0)
      1   (-1, 0)   2   (-1, 1)   1   (1, -1, 0, 0)
      2   (1, 2)    0   (1, 1)    4   (1, 1, 2, 2)
      2   (1, 2)    1   (1, -1)   5   (1, -1, 2, -2)
      2   (1, 2)    2   (-1, 1)   6   (-1, 1, -2, 2)
```

```
>>> from sage.all import *
>>> VW.index_map(Integer(0), Integer(1))
1
>>> VW.vectors()[VW.index_map(Integer(0), Integer(1))]
(1, -1, 0, 0)

>>> rows = []
>>> for i, j in cartesian_product((range(Integer(3)), range(Integer(3)))):
...     v = V.vectors()[i]
...     w = W.vectors()[j]
...     i_tensor_j = VW.index_map(i, j)
...     vw = VW.vectors()[i_tensor_j]
...     rows.append([i, v, j, w, i_tensor_j, vw])
>>> table(rows)
      0   (1, 0)    0   (1, 1)    0   (1, 1, 0, 0)
```

(continues on next page)

(continued from previous page)

```
    0   (1, 0)    1   (1, -1)   1   (1, -1, 0, 0)
    0   (1, 0)    2   (-1, 1)   2   (-1, 1, 0, 0)
    1   (-1, 0)   0   (1, 1)    3   (-1, -1, 0, 0)
    1   (-1, 0)   1   (1, -1)   2   (-1, 1, 0, 0)
    1   (-1, 0)   2   (-1, 1)   1   (1, -1, 0, 0)
    2   (1, 2)    0   (1, 1)    4   (1, 1, 2, 2)
    2   (1, 2)    1   (1, -1)   5   (1, -1, 2, -2)
    2   (1, 2)    2   (-1, 1)   6   (-1, 1, -2, 2)
```

class sage.modules.tensor_operations.**TensorOperation**(*vector_collections*, *operation='product'*)

Bases: *VectorCollection*

Auxiliary class to compute the tensor product of two *VectorCollection* objects.

Warning

This class is only used as a base class for filtered vector spaces. You should not use it yourself.

INPUT:

- `vector_collections` – a nonempty list/tuple/iterable of *VectorCollection* objects
- `operation` – string; the tensor operation. Currently allowed values are `'product'`, `'symmetric'`, and `'antisymmetric'`.

Todo

More general tensor operations (specified by Young tableaux) should be implemented.

EXAMPLES:

```
sage: from sage.modules.tensor_operations import VectorCollection, TensorOperation
sage: R = VectorCollection([(1,0), (1,2), (-1,-2)], QQ, 2)
sage: S = VectorCollection([(1,), (-1,)], QQ, 1)
sage: R_tensor_S = TensorOperation([R, S])
sage: R_tensor_S.index_map(0, 0)
0
sage: matrix(ZZ, 3, 2, lambda i,j: R_tensor_S.index_map(i, j))
[0 1]
[2 3]
[3 2]
sage: R_tensor_S.vectors()
((1, 0), (-1, 0), (1, 2), (-1, -2))
```

```
>>> from sage.all import *
>>> from sage.modules.tensor_operations import VectorCollection, TensorOperation
>>> R = VectorCollection([(Integer(1),Integer(0)), (Integer(1),Integer(2)), (-
↪Integer(1),-Integer(2))], QQ, Integer(2))
>>> S = VectorCollection([(Integer(1),), (-Integer(1),)], QQ, Integer(1))
>>> R_tensor_S = TensorOperation([R, S])
>>> R_tensor_S.index_map(Integer(0), Integer(0))
```

(continues on next page)

(continued from previous page)

```
0
>>> matrix(ZZ, Integer(3), Integer(2), lambda i,j: R_tensor_S.index_map(i, j))
[0 1]
[2 3]
[3 2]
>>> R_tensor_S.vectors()
((1, 0), (-1, 0), (1, 2), (-1, -2))
```

codomain()

The codomain of the index map.

OUTPUT: list of integers; the image of *index_map()*

EXAMPLES:

```
sage: from sage.modules.tensor_operations import                ....:     ␣
↪VectorCollection, TensorOperation
sage: R = VectorCollection([(1,0), (0,1), (-2,-3)], QQ, 2)
sage: detR = TensorOperation([R]*2, 'antisymmetric')                        #␣
↪needs sage.groups
sage: sorted(detR.preimage())                                               #␣
↪needs sage.groups
[(0, 1), (0, 2), (1, 2)]
sage: sorted(detR.codomain())                                               #␣
↪needs sage.groups
[0, 1, 2]
```

```
>>> from sage.all import *
>>> from sage.modules.tensor_operations import                Ellipsis.:     ␣
↪VectorCollection, TensorOperation
>>> R = VectorCollection([(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-
↪Integer(2),-Integer(3))], QQ, Integer(2))
>>> detR = TensorOperation([R]*Integer(2), 'antisymmetric')                  ␣
↪      # needs sage.groups
>>> sorted(detR.preimage())                                                 #␣
↪needs sage.groups
[(0, 1), (0, 2), (1, 2)]
>>> sorted(detR.codomain())                                                 #␣
↪needs sage.groups
[0, 1, 2]
```

index_map(**i*)

Return the result of the tensor operation.

INPUT:

- `*i` – list of integers. The indices (in the corresponding factor of the tensor operation) of the domain vector.

OUTPUT:

The index (in `vectors()`) of the image of the tensor product/operation acting on the domain vectors indexed by i.

`None` is returned if the tensor operation maps the generators to zero (usually because of antisymmetry).

EXAMPLES:

```
sage: from sage.modules.tensor_operations import                    ....:     ␣
↪VectorCollection, TensorOperation
sage: R = VectorCollection([(1,0), (1,2), (-1,-2)], QQ, 2)
sage: Sym3_R = TensorOperation([R]*3, 'symmetric')
```

```
>>> from sage.all import *
>>> from sage.modules.tensor_operations import                    Ellipsis.:     ␣
↪VectorCollection, TensorOperation
>>> R = VectorCollection([(Integer(1),Integer(0)), (Integer(1),Integer(2)), (-
↪Integer(1),-Integer(2))], QQ, Integer(2))
>>> Sym3_R = TensorOperation([R]*Integer(3), 'symmetric')
```

The symmetric product of the first vector `(1,0)`, the second vector `(1,2)`, and the third vector `(-1,-2)` equals the vector with index number 4 (that is, the fifth) in the symmetric product vector collection:

```
sage: Sym3_R.index_map(0, 1, 2)
4
```

```
>>> from sage.all import *
>>> Sym3_R.index_map(Integer(0), Integer(1), Integer(2))
4
```

In suitable coordinates, this is the vector:

```
sage: Sym3_R.vectors()[4]
(-1, -4, -4, 0)
```

```
>>> from sage.all import *
>>> Sym3_R.vectors()[Integer(4)]
(-1, -4, -4, 0)
```

The product is symmetric:

```
sage: Sym3_R.index_map(2, 0, 1)
4
sage: Sym3_R.index_map(2, 1, 0)
4
```

```
>>> from sage.all import *
>>> Sym3_R.index_map(Integer(2), Integer(0), Integer(1))
4
>>> Sym3_R.index_map(Integer(2), Integer(1), Integer(0))
4
```

As another example, here is the rank-2 determinant:

```
sage: from sage.modules.tensor_operations import                    ....:     ␣
↪VectorCollection, TensorOperation
sage: R = VectorCollection([(1,0), (0,1), (-2,-3)], QQ, 2)
sage: detR = TensorOperation([R]*2, 'antisymmetric')                          #␣
↪needs sage.groups
```

(continues on next page)

(continued from previous page)

```
sage: detR.index_map(1, 0)                                                  #␣
↪needs sage.groups
0
sage: detR.index_map(0, 1)                                                  #␣
↪needs sage.groups
0
```

```
>>> from sage.all import *
>>> from sage.modules.tensor_operations import              Ellipsis.:     ␣
↪VectorCollection, TensorOperation
>>> R = VectorCollection([(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-
↪Integer(2),-Integer(3))], QQ, Integer(2))
>>> detR = TensorOperation([R]*Integer(2), 'antisymmetric')                  ␣
↪     # needs sage.groups
>>> detR.index_map(Integer(1), Integer(0))                                  ␣
↪              # needs sage.groups
0
>>> detR.index_map(Integer(0), Integer(1))                                  ␣
↪              # needs sage.groups
0
```

preimage()

A choice of pre-image multi-indices.

OUTPUT:

A list of multi-indices (tuples of integers) whose image is the entire image under the *index_map()*.

EXAMPLES:

```
sage: from sage.modules.tensor_operations import              ....:     ␣
↪VectorCollection, TensorOperation
sage: R = VectorCollection([(1,0), (0,1), (-2,-3)], QQ, 2)
sage: detR = TensorOperation([R]*2, 'antisymmetric')                        #␣
↪needs sage.groups
sage: sorted(detR.preimage())                                               #␣
↪needs sage.groups
[(0, 1), (0, 2), (1, 2)]
sage: sorted(detR.codomain())                                               #␣
↪needs sage.groups
[0, 1, 2]
```

```
>>> from sage.all import *
>>> from sage.modules.tensor_operations import              Ellipsis.:     ␣
↪VectorCollection, TensorOperation
>>> R = VectorCollection([(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-
↪Integer(2),-Integer(3))], QQ, Integer(2))
>>> detR = TensorOperation([R]*Integer(2), 'antisymmetric')                  ␣
↪     # needs sage.groups
>>> sorted(detR.preimage())                                                 #␣
↪needs sage.groups
[(0, 1), (0, 2), (1, 2)]
>>> sorted(detR.codomain())                                                 #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.groups
[0, 1, 2]
```

class sage.modules.tensor_operations.**VectorCollection**(*vector_collection*, *base_ring*, *dim*)

Bases: *FreeModule_ambient_field*

An ordered collection of generators of a vector space.

This is like a list of vectors, but with extra argument checking.

Warning

This class is only used as a base class for filtered vector spaces. You should not use it yourself.

INPUT:

- `dim` – integer; the dimension of the ambient vector space
- `base_ring` – a field; the base field of the ambient vector space
- `rays` – any list/iterable of things than can be converted into vectors of the ambient vector space. These will be used to span the subspaces of the filtration. Must span the ambient vector space.

EXAMPLES:

```
sage: from sage.modules.tensor_operations import VectorCollection
sage: R = VectorCollection([(1,0), (0,1), (1,2)], QQ, 2);  R
Vector space of dimension 2 over Rational Field
```

```
>>> from sage.all import *
>>> from sage.modules.tensor_operations import VectorCollection
>>> R = VectorCollection([(Integer(1),Integer(0)), (Integer(0),Integer(1)),␣
↪(Integer(1),Integer(2))], QQ, Integer(2));  R
Vector space of dimension 2 over Rational Field
```

n_vectors()

Return the number of vectors.

OUTPUT: integer

EXAMPLES:

```
sage: from sage.modules.tensor_operations import VectorCollection
sage: V = VectorCollection([(1,0), (0,1), (1,2)], QQ, 2)
sage: V.n_vectors()
3
```

```
>>> from sage.all import *
>>> from sage.modules.tensor_operations import VectorCollection
>>> V = VectorCollection([(Integer(1),Integer(0)), (Integer(0),Integer(1)),␣
↪(Integer(1),Integer(2))], QQ, Integer(2))
>>> V.n_vectors()
3
```

vectors()

Return the collection of vectors.

OUTPUT:

A tuple of vectors. The vectors that were specified in the constructor, in the same order.

EXAMPLES:

```
sage: from sage.modules.tensor_operations import VectorCollection
sage: V = VectorCollection([(1,0), (0,1), (1,2)], QQ, 2)
sage: V.vectors()
((1, 0), (0, 1), (1, 2))
```

```
>>> from sage.all import *
>>> from sage.modules.tensor_operations import VectorCollection
>>> V = VectorCollection([(Integer(1),Integer(0)), (Integer(0),Integer(1)),␣
↪(Integer(1),Integer(2))], QQ, Integer(2))
>>> V.vectors()
((1, 0), (0, 1), (1, 2))
```

sage.modules.tensor_operations.**antisymmetrized_coordinate_sums**(*dim*, *n*)

Return formal anti-symmetrized sum of multi-indices.

INPUT:

- dim – integer; the dimension (range of each index)
- n – integer; the total number of indices

OUTPUT:

An anti-symmetrized formal sum of multi-indices (tuples of integers)

EXAMPLES:

```
sage: from sage.modules.tensor_operations import antisymmetrized_coordinate_sums
sage: antisymmetrized_coordinate_sums(3, 2)                                    #␣
↪needs sage.groups
((0, 1) - (1, 0), (0, 2) - (2, 0), (1, 2) - (2, 1))
```

```
>>> from sage.all import *
>>> from sage.modules.tensor_operations import antisymmetrized_coordinate_sums
>>> antisymmetrized_coordinate_sums(Integer(3), Integer(2))                    ␣
↪              # needs sage.groups
((0, 1) - (1, 0), (0, 2) - (2, 0), (1, 2) - (2, 1))
```

sage.modules.tensor_operations.**symmetrized_coordinate_sums**(*dim*, *n*)

Return formal symmetrized sum of multi-indices.

INPUT:

- dim – integer; the dimension (range of each index)
- n – integer; the total number of indices

OUTPUT:

A symmetrized formal sum of multi-indices (tuples of integers)

EXAMPLES:

```
sage: from sage.modules.tensor_operations import symmetrized_coordinate_sums
sage: symmetrized_coordinate_sums(2, 2)
((0, 0), (0, 1) + (1, 0), (1, 1))
```

```
>>> from sage.all import *
>>> from sage.modules.tensor_operations import symmetrized_coordinate_sums
>>> symmetrized_coordinate_sums(Integer(2), Integer(2))
((0, 0), (0, 1) + (1, 0), (1, 1))
```

10.3 Iterators over finite submodules of a Z-module

We iterate over the elements of a finite **Z**-module. The action of **Z** must be the natural one.

This class is intended to provide optimizations for the `sage.free_module.FreeModule_generic:__iter__()` method.

AUTHORS:

- Thomas Feulner (2012-08-31): initial version
- Punarbasu Purkayastha (2012-11-09): replaced the loop with recursion
- Thomas Feulner (2012-11-09): added functionality to enumerate cosets, FiniteFieldsubspace_projPoint_iterator

EXAMPLES:

```
sage: from sage.modules.finite_submodule_iter import FiniteZZsubmodule_iterator
sage: F.<x,y,z> = FreeAlgebra(GF(3),3)
sage: iter = FiniteZZsubmodule_iterator([x,y], [3,3])
sage: list(iter)
[0, x, 2*x, y, x + y, 2*x + y, 2*y, x + 2*y, 2*x + 2*y]
```

```
>>> from sage.all import *
>>> from sage.modules.finite_submodule_iter import FiniteZZsubmodule_iterator
>>> F = FreeAlgebra(GF(Integer(3)),Integer(3), names=('x', 'y', 'z',)); (x, y, z,) =␣
↪F._first_ngens(3)
>>> iter = FiniteZZsubmodule_iterator([x,y], [Integer(3),Integer(3)])
>>> list(iter)
[0, x, 2*x, y, x + y, 2*x + y, 2*y, x + 2*y, 2*x + 2*y]
```

There is a specialization for subspaces over finite fields:

```
sage: from sage.modules.finite_submodule_iter import FiniteFieldsubspace_iterator
sage: A = random_matrix(GF(4, 'a'), 5, 100)
sage: iter = FiniteFieldsubspace_iterator(A)
sage: len(list(iter))
1024
```

```
>>> from sage.all import *
>>> from sage.modules.finite_submodule_iter import FiniteFieldsubspace_iterator
>>> A = random_matrix(GF(Integer(4), 'a'), Integer(5), Integer(100))
>>> iter = FiniteFieldsubspace_iterator(A)
>>> len(list(iter))
1024
```

The module also allows the iteration over cosets:

```
sage: from sage.modules.finite_submodule_iter import FiniteFieldsubspace_iterator
sage: A = random_matrix(GF(4, 'a'), 5, 100)
sage: v = random_vector(GF(4, 'a'), 100)
sage: iter = FiniteFieldsubspace_iterator(A, v)
sage: len(list(iter))
1024
```

```
>>> from sage.all import *
>>> from sage.modules.finite_submodule_iter import FiniteFieldsubspace_iterator
>>> A = random_matrix(GF(Integer(4), 'a'), Integer(5), Integer(100))
>>> v = random_vector(GF(Integer(4), 'a'), Integer(100))
>>> iter = FiniteFieldsubspace_iterator(A, v)
>>> len(list(iter))
1024
```

class sage.modules.finite_submodule_iter.**FiniteFieldsubspace_iterator**

Bases: *FiniteZZsubmodule_iterator*

This class implements an iterator over the subspace of a vector space over a finite field. The subspace is generated by `basis`.

INPUT:

- `basis` – list of vectors or a matrix with elements over a finite field. If a matrix is provided then it is not checked whether the matrix is full ranked. Similarly, if a list of vectors is provided, then the linear independence of the vectors is not checked.
- `coset_rep` – (optional) a vector in the same ambient space, if one aims to compute a coset of the vector space given by `basis`
- `immutable` – boolean (default: `False`); set it to `True` to return immutable vectors

EXAMPLES:

```
sage: from sage.modules.finite_submodule_iter import FiniteFieldsubspace_iterator
sage: A = random_matrix(GF(2), 10, 100)
sage: iter = FiniteFieldsubspace_iterator(A)
sage: len(list(iter))
1024
sage: X = random_matrix(GF(4, 'a'), 7, 100).row_space()
sage: s = list(X)  # long time (5s on sage.math, 2013)
sage: t = list(FiniteFieldsubspace_iterator(X.basis()))  # takes 0.31s
sage: sorted(t) == sorted(s)  # long time
True
```

```
>>> from sage.all import *
>>> from sage.modules.finite_submodule_iter import FiniteFieldsubspace_iterator
>>> A = random_matrix(GF(Integer(2)), Integer(10), Integer(100))
>>> iter = FiniteFieldsubspace_iterator(A)
>>> len(list(iter))
1024
>>> X = random_matrix(GF(Integer(4), 'a'), Integer(7), Integer(100)).row_space()
>>> s = list(X)  # long time (5s on sage.math, 2013)
>>> t = list(FiniteFieldsubspace_iterator(X.basis()))  # takes 0.31s
```

(continues on next page)

(continued from previous page)

```
>>> sorted(t) == sorted(s)  # long time
True
```

class sage.modules.finite_submodule_iter.**FiniteFieldsubspace_projPoint_iterator**

Bases: object

This class implements an iterator over the projective points of a vector space over a finite field. The vector space is generated by `basis` and need not to be equal to the full ambient space.

A projective point (= one dimensional subspace) P will be represented by a generator p. To ensure that all p will be normalized you can set the optional argument `normalize` to `True`.

INPUT:

- `basis` – list of vectors or a matrix with elements over a finite field. If a matrix is provided then it is not checked whether the matrix is full ranked. Similarly, if a list of vectors is provided, then the linear independence of the vectors is not checked.
- `normalize` – boolean (default: `False`); whether the returned vectors should be normalized, i.e. the first nonzero coordinate is equal to 1
- `immutable` – boolean (default: `False`); set it to `True` to return immutable vectors

EXAMPLES:

```
sage: from sage.modules.finite_submodule_iter import FiniteFieldsubspace_iterator,
↪ FiniteFieldsubspace_projPoint_iterator
sage: A = random_matrix(GF(4, 'a'), 5, 100)
sage: a = len(list(FiniteFieldsubspace_iterator(A)))
sage: b = len(list(FiniteFieldsubspace_projPoint_iterator(A)))
sage: b == (a-1)/3
True
```

```
>>> from sage.all import *
>>> from sage.modules.finite_submodule_iter import FiniteFieldsubspace_iterator,␣
↪FiniteFieldsubspace_projPoint_iterator
>>> A = random_matrix(GF(Integer(4), 'a'), Integer(5), Integer(100))
>>> a = len(list(FiniteFieldsubspace_iterator(A)))
>>> b = len(list(FiniteFieldsubspace_projPoint_iterator(A)))
>>> b == (a-Integer(1))/Integer(3)
True
```

Prove that the option `normalize == True` will only return normalized vectors.

sage: all(x.monic() == x for x in FiniteFieldsubspace_projPoint_iterator(A, True)) True

class sage.modules.finite_submodule_iter.**FiniteZZsubmodule_iterator**

Bases: object

Let G be an abelian group and suppose that $(g_0, \ldots, g_n)$ is a list of elements of G, whose additive orders are equal to m_i and $\sum_{i=0}^{n} x_i g_i = 0$ for $x_i \in \mathbf{Z}_{m_i}$ for $i \in \{0, \ldots, n\}$ implies $x_i = 0$ for all i.

This class implements an iterator over the $\mathbf{Z}$-submodule $M = \{\sum_{i=0}^{n} x_i g_i\}$. If the independence condition from above is not fulfilled, we can still use this iterator to run over the elements. In this case the elements will occur multiple times.

Getting from one element of the submodule to another is performed by one single addition in G.

INPUT:

- `basis` – the elements $(g_0, \ldots, g_n)$
- `order` – (optional) the additive_orders m_i of g_i
- `coset_rep` – (optional) an element of g, if one aims to compute a coset of the **Z**-submodule M
- `immutable` – boolean (default: `False`); set it to `True` to return immutable elements. Setting this to `True` makes sense if the elements are vectors. See *`FiniteFieldsubspace_iterator`* for examples.

EXAMPLES:

```
sage: from sage.modules.finite_submodule_iter import FiniteZZsubmodule_iterator
sage: F.<x,y,z> = FreeAlgebra(GF(3),3)
sage: iter = FiniteZZsubmodule_iterator([x,y], [3,3])
sage: list(iter)
[0, x, 2*x, y, x + y, 2*x + y, 2*y, x + 2*y, 2*x + 2*y]
sage: iter = FiniteZZsubmodule_iterator([x,y], [3,3], z)
sage: list(iter)
[z, x + z, 2*x + z, y + z, x + y + z, 2*x + y + z, 2*y + z, x + 2*y + z, 2*x +␣
↪2*y + z]
```

```
>>> from sage.all import *
>>> from sage.modules.finite_submodule_iter import FiniteZZsubmodule_iterator
>>> F = FreeAlgebra(GF(Integer(3)),Integer(3), names=('x', 'y', 'z',)); (x, y, z,
↪) = F._first_ngens(3)
>>> iter = FiniteZZsubmodule_iterator([x,y], [Integer(3),Integer(3)])
>>> list(iter)
[0, x, 2*x, y, x + y, 2*x + y, 2*y, x + 2*y, 2*x + 2*y]
>>> iter = FiniteZZsubmodule_iterator([x,y], [Integer(3),Integer(3)], z)
>>> list(iter)
[z, x + z, 2*x + z, y + z, x + y + z, 2*x + y + z, 2*y + z, x + 2*y + z, 2*x +␣
↪2*y + z]
```

10.4 Miscellaneous module-related functions

AUTHORS:

- William Stein (2007-11-18)

`sage.modules.misc.`**`gram_schmidt`**(B)

Return the Gram-Schmidt orthogonalization of the entries in the list B of vectors, along with the matrix mu of Gram-Schmidt coefficients.

Note that the output vectors need not have unit length. We do this to avoid having to extract square roots.

Note

Use of this function is discouraged. It fails on linearly dependent input and its output format is not as natural as it could be. Instead, see `sage.matrix.matrix2.Matrix2.gram_schmidt()` which is safer and more general-purpose.

EXAMPLES:

```
sage: B = [vector([1,2,1/5]), vector([1,2,3]), vector([-1,0,0])]
sage: from sage.modules.misc import gram_schmidt
sage: G, mu = gram_schmidt(B)
sage: G
[(1, 2, 1/5), (-1/9, -2/9, 25/9), (-4/5, 2/5, 0)]
sage: G[0] * G[1]
0
sage: G[0] * G[2]
0
sage: G[1] * G[2]
0
sage: mu
[      0       0       0]
[   10/9       0       0]
[-25/126    1/70       0]
sage: a = matrix([])
sage: a.gram_schmidt()
([], [])
sage: a = matrix([[],[],[],[]])
sage: a.gram_schmidt()
 ([], [])
```

```
>>> from sage.all import *
>>> B = [vector([Integer(1),Integer(2),Integer(1)/Integer(5)]),␣
↪vector([Integer(1),Integer(2),Integer(3)]), vector([-Integer(1),Integer(0),
↪Integer(0)])]
>>> from sage.modules.misc import gram_schmidt
>>> G, mu = gram_schmidt(B)
>>> G
[(1, 2, 1/5), (-1/9, -2/9, 25/9), (-4/5, 2/5, 0)]
>>> G[Integer(0)] * G[Integer(1)]
0
>>> G[Integer(0)] * G[Integer(2)]
0
>>> G[Integer(1)] * G[Integer(2)]
0
>>> mu
[      0       0       0]
[   10/9       0       0]
[-25/126    1/70       0]
>>> a = matrix([])
>>> a.gram_schmidt()
([], [])
>>> a = matrix([[],[],[],[]])
>>> a.gram_schmidt()
 ([], [])
```

Linearly dependent input leads to a zero dot product in a denominator. This shows that Issue #10791 is fixed.

```
sage: from sage.modules.misc import gram_schmidt
sage: V = [vector(ZZ,[1,1]), vector(ZZ,[2,2]), vector(ZZ,[1,2])]
sage: gram_schmidt(V)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: linearly dependent input for module version of Gram-Schmidt
```

```
>>> from sage.all import *
>>> from sage.modules.misc import gram_schmidt
>>> V = [vector(ZZ,[Integer(1),Integer(1)]), vector(ZZ,[Integer(2),Integer(2)]),␣
↪vector(ZZ,[Integer(1),Integer(2)])]
>>> gram_schmidt(V)
Traceback (most recent call last):
...
ValueError: linearly dependent input for module version of Gram-Schmidt
```

CHAPTER

ELEVEN

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

m

sage.modules.complex_double_vector, 753
sage.modules.diamond_cutting, 755
sage.modules.fg_pid.fgp_element, 366
sage.modules.fg_pid.fgp_module, 333
sage.modules.fg_pid.fgp_morphism, 370
sage.modules.filtered_vector_space, 576
sage.modules.finite_submodule_iter, 766
sage.modules.fp_graded.element, 430
sage.modules.fp_graded.free_element, 397
sage.modules.fp_graded.free_homspace, 405
sage.modules.fp_graded.free_module, 377
sage.modules.fp_graded.free_morphism, 402
sage.modules.fp_graded.homspace, 460
sage.modules.fp_graded.module, 406
sage.modules.fp_graded.morphism, 435
sage.modules.fp_graded.steenrod.module, 464
sage.modules.fp_graded.steenrod.morphism, 470
sage.modules.free_module, 16
sage.modules.free_module_element, 136
sage.modules.free_module_homspace, 638
sage.modules.free_module_integer, 477
sage.modules.free_module_morphism, 644
sage.modules.free_module_pseudohomspace, 697
sage.modules.free_module_pseudomorphism, 699
sage.modules.free_quadratic_module, 493
sage.modules.free_quadratic_module_integer_symmetric, 514
sage.modules.matrix_morphism, 663
sage.modules.misc, 769
sage.modules.module, 11
sage.modules.multi_filtered_vector_space, 596
sage.modules.quotient_module, 236
sage.modules.real_double_vector, 754
sage.modules.submodule, 232
sage.modules.tensor_operations, 758
sage.modules.torsion_quadratic_module, 555
sage.modules.tutorial_free_modules, 3
sage.modules.vector_callable_symbolic_dense, 737
sage.modules.vector_complex_double_dense, 752
sage.modules.vector_double_dense, 738
sage.modules.vector_integer_dense, 707
sage.modules.vector_integer_sparse, 708
sage.modules.vector_mod2_dense, 708
sage.modules.vector_modn_dense, 710
sage.modules.vector_modn_sparse, 713
sage.modules.vector_numpy_dense, 749
sage.modules.vector_numpy_integer_dense, 751
sage.modules.vector_rational_dense, 713
sage.modules.vector_rational_sparse, 714
sage.modules.vector_real_double_dense, 747
sage.modules.vector_space_homspace, 611
sage.modules.vector_space_morphism, 617
sage.modules.vector_symbolic_dense, 714
sage.modules.vector_symbolic_sparse, 725
sage.modules.with_basis.all, 249
sage.modules.with_basis.cell_module, 249
sage.modules.with_basis.indexed_element, 253
sage.modules.with_basis.invariant, 256
sage.modules.with_basis.morphism, 271
sage.modules.with_basis.representation, 305
sage.modules.with_basis.subquotient, 295

INDEX

A

additive_order() (*sage.modules.fg_pid.fgp_element.FGP_Element method*), 367
additive_order() (*sage.modules.free_module_element.FreeModuleElement method*), 139
all_submodules() (*sage.modules.torsion_quadratic_module.TorsionQuadraticModule method*), 557
ambient() (*sage.modules.free_module.FreeModule_submodule_with_basis_pid method*), 101
ambient() (*sage.modules.with_basis.subquotient.QuotientModuleWithBasis method*), 295
ambient() (*sage.modules.with_basis.subquotient.SubmoduleWithBasis method*), 297
ambient_module() (*sage.modules.free_module.FreeModule_ambient method*), 25
ambient_module() (*sage.modules.free_module.FreeModule_submodule_with_basis_pid method*), 103
ambient_module() (*sage.modules.free_quadratic_module.FreeQuadraticModule_generic method*), 497
ambient_module() (*sage.modules.quotient_module.QuotientModule_free_ambient method*), 244
ambient_module() (*sage.modules.submodule.Submodule_free_ambient method*), 234
ambient_vector_space() (*sage.modules.filtered_vector_space.FilteredVectorSpace_class method*), 581
ambient_vector_space() (*sage.modules.free_module.FreeModule_ambient_domain method*), 34
ambient_vector_space() (*sage.modules.free_module.FreeModule_ambient_field method*), 36
ambient_vector_space() (*sage.modules.free_module.FreeModule_submodule_with_basis_pid method*), 103
ambient_vector_space() (*sage.modules.free_quadratic_module.FreeQuadraticModule_ambient_domain method*), 495
ambient_vector_space() (*sage.modules.multi_filtered_vector_space.MultiFilteredVectorSpace_class method*), 598
an_element() (*sage.modules.fp_graded.free_module.FreeGradedModule method*), 387
an_element() (*sage.modules.fp_graded.homspace.FPModuleHomspace method*), 460
an_element() (*sage.modules.fp_graded.module.FPModule method*), 408
annihilator() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 340
antisymmetrized_coordinate_sums() (*in module sage.modules.tensor_operations*), 765
apply_map() (*in module sage.modules.vector_symbolic_dense*), 725
apply_map() (*in module sage.modules.vector_symbolic_sparse*), 736
apply_map() (*sage.modules.free_module_element.FreeModuleElement method*), 139
approximate_closest_vector() (*sage.modules.free_module_integer.FreeModule_submodule_with_basis_integer method*), 481
are_linearly_dependent() (*sage.modules.free_module.FreeModule_generic method*), 38
automorphisms() (*sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method*), 517

B

b() (*sage.modules.torsion_quadratic_module.TorsionQuadraticModuleElement method*), 573
babai() (*sage.modules.free_module_integer.FreeModule_submodule_with_basis_integer method*), 483
base_extend() (*sage.modules.module.Module method*), 13
base_field() (*sage.modules.free_module.FreeModule_ambient_field method*), 36
base_field() (*sage.modules.free_module.FreeModule_generic method*), 39
base_ring() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 341
base_ring() (*sage.modules.matrix_morphism.MatrixMorphism_abstract method*), 668

BaseIsomorphism1D (*class in sage.modules.free_module_morphism*), 644
BaseIsomorphism1D_from_FM (*class in sage.modules.free_module_morphism*), 645
BaseIsomorphism1D_to_FM (*class in sage.modules.free_module_morphism*), 646
basis() (*sage.modules.free_module_homspace.FreeModuleHomspace method*), 640
basis() (*sage.modules.free_module_pseudo_homspace.FreeModulePseudoHomspace method*), 697
basis() (*sage.modules.free_module.FreeModule_ambient method*), 25
basis() (*sage.modules.free_module.FreeModule_generic method*), 39
basis() (*sage.modules.free_module.FreeModule_submodule_with_basis_pid method*), 105
basis_elements() (*sage.modules.fp_graded.free_module.FreeGradedModule method*), 388
basis_elements() (*sage.modules.fp_graded.homspace.FPModuleHomspace method*), 460
basis_elements() (*sage.modules.fp_graded.module.FPModule method*), 409
basis_matrix() (*sage.modules.free_module.FreeModule_generic method*), 39
basis_seq() (*in module sage.modules.free_module*), 131
bilinear_form() (*sage.modules.with_basis.cell_module.CellModule method*), 250
bilinear_form_matrix() (*sage.modules.with_basis.cell_module.CellModule method*), 250
BKZ() (*sage.modules.free_module_integer.FreeModule_submodule_with_basis_integer method*), 478
brauer_character() (*sage.modules.with_basis.representation.Representation_abstract method*), 311
brown_invariant() (*sage.modules.torsion_quadratic_module.TorsionQuadraticModule method*), 558

C

calculate_voronoi_cell() (*in module sage.modules.diamond_cutting*), 755
canonicalize_radical() (*sage.modules.vector_symbolic_dense.Vector_symbolic_dense method*), 715
canonicalize_radical() (*sage.modules.vector_symbolic_sparse.Vector_symbolic_sparse method*), 726
cardinality() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 341
cardinality() (*sage.modules.free_module.FreeModule_generic method*), 41
CellModule (*class in sage.modules.with_basis.cell_module*), 249
CellModule.Element (*class in sage.modules.with_basis.cell_module*), 250
cellular_algebra() (*sage.modules.with_basis.cell_module.CellModule method*), 250
change_ring() (*sage.modules.filtered_vector_space.FilteredVectorSpace_class method*), 581
change_ring() (*sage.modules.fp_graded.free_module.FreeGradedModule method*), 389
change_ring() (*sage.modules.fp_graded.module.FPModule method*), 410
change_ring() (*sage.modules.fp_graded.morphism.FPModuleMorphism method*), 435
change_ring() (*sage.modules.free_module_element.FreeModuleElement method*), 142
change_ring() (*sage.modules.free_module_morphism.FreeModuleMorphism method*), 647
change_ring() (*sage.modules.free_module.FreeModule_ambient method*), 26
change_ring() (*sage.modules.free_module.FreeModule_submodule_with_basis_pid method*), 105
change_ring() (*sage.modules.free_quadratic_module.FreeQuadraticModule_submodule_with_basis_pid method*), 510
change_ring() (*sage.modules.module.Module method*), 14
change_ring() (*sage.modules.multi_filtered_vector_space.MultiFilteredVectorSpace_class method*), 598
character() (*sage.modules.with_basis.representation.Representation_abstract method*), 312
characteristic_polynomial() (*sage.modules.matrix_morphism.MatrixMorphism_abstract method*), 668
charpoly() (*sage.modules.matrix_morphism.MatrixMorphism_abstract method*), 669
closest_vector() (*sage.modules.free_module_integer.FreeModule_submodule_with_basis_integer method*), 483
codimension() (*sage.modules.free_module.FreeModule_generic method*), 42
codomain() (*sage.modules.tensor_operations.TensorOperation method*), 761
cokernel_basis_indices() (*sage.modules.with_basis.morphism.TriangularModuleMorphism method*), 285
cokernel_projection() (*sage.modules.fp_graded.morphism.FPModuleMorphism method*), 436
cokernel_projection() (*sage.modules.fp_graded.steenrod.morphism.SteenrodFPModuleMorphism method*), 470
cokernel_projection() (*sage.modules.with_ba-*

sis.morphism.TriangularModuleMorphism method), 287
column() (*sage.modules.free_module_element.FreeModuleElement method*), 142
complement() (*sage.modules.free_module.FreeModule_generic_field method*), 65
complex_vector() (*sage.modules.vector_double_dense.Vector_double_dense method*), 739
ComplexDoubleVectorSpace_class (*class in sage.modules.free_module*), 19
composition_factors() (*sage.modules.with_basis.representation.Representation_abstract method*), 313
composition_series() (*sage.modules.with_basis.representation.Representation_abstract method*), 315
concatenate() (*sage.modules.free_module_element.FreeModuleElement method*), 144
conjugate() (*sage.modules.free_module_element.FreeModuleElement method*), 146
connectivity() (*sage.modules.fp_graded.free_module.FreeGradedModule method*), 389
connectivity() (*sage.modules.fp_graded.module.FPModule method*), 411
construct_from_dim_degree() (*in module sage.modules.filtered_vector_space*), 593
construct_from_generators() (*in module sage.modules.filtered_vector_space*), 594
construct_from_generators_indices() (*in module sage.modules.filtered_vector_space*), 594
construction() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 342
construction() (*sage.modules.free_module.FreeModule_generic method*), 42
construction() (*sage.modules.free_module.FreeModule_submodule_with_basis_pid method*), 106
construction() (*sage.modules.with_basis.invariant.FiniteDimensionalInvariantModule method*), 262
coordinate_module() (*sage.modules.free_module.FreeModule_generic method*), 43
coordinate_ring() (*sage.modules.free_module_element.FreeModuleElement method*), 148
coordinate_ring() (*sage.modules.free_module.FreeModule_generic method*), 44
coordinate_ring() (*sage.modules.free_module.Module_free_ambient method*), 117
coordinate_vector() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 343
coordinate_vector() (*sage.modules.free_module.FreeModule_ambient method*), 26
coordinate_vector() (*sage.modules.free_module.FreeModule_ambient_domain method*), 35
coordinate_vector() (*sage.modules.free_module.FreeModule_generic method*), 45
coordinate_vector() (*sage.modules.free_module.FreeModule_submodule_field method*), 94
coordinate_vector() (*sage.modules.free_module.FreeModule_submodule_pid method*), 97
coordinate_vector() (*sage.modules.free_module.FreeModule_submodule_with_basis_pid method*), 107
coordinates() (*sage.modules.free_module.ComplexDoubleVectorSpace_class method*), 19
coordinates() (*sage.modules.free_module.FreeModule_generic method*), 46
coordinates() (*sage.modules.free_module.RealDoubleVectorSpace_class method*), 130
coreduced() (*sage.modules.with_basis.morphism.TriangularModuleMorphism method*), 287
cover() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 344
cover() (*sage.modules.quotient_module.FreeModule_ambient_field_quotient method*), 240
cover() (*sage.modules.quotient_module.QuotientModule_free_ambient method*), 244
create_key() (*sage.modules.free_module.FreeModuleFactory method*), 25
create_object() (*sage.modules.free_module.FreeModuleFactory method*), 25
cross_product() (*sage.modules.free_module_element.FreeModuleElement method*), 148
cross_product_matrix() (*sage.modules.free_module_element.FreeModuleElement method*), 150
curl() (*sage.modules.free_module_element.FreeModuleElement method*), 151

D

decomposition() (*sage.modules.matrix_morphism.MatrixMorphism_abstract method*), 670
defining_homomorphism() (*sage.modules.fp_graded.module.FPModule method*), 412
degree() (*sage.modules.fp_graded.element.FPElement method*), 430
degree() (*sage.modules.fp_graded.free_element.FreeGradedModuleElement method*), 398
degree() (*sage.modules.fp_graded.free_morphism.FreeGradedModuleMorphism method*), 403
degree() (*sage.modules.fp_graded.morphism.FPModuleMorphism method*), 437
degree() (*sage.modules.free_module_element.FreeModuleElement method*), 153
degree() (*sage.modules.free_module.Module_free_ambient method*), 117

denominator() (*sage.modules.free_module_element.FreeModuleElement method*), 153
denominator() (*sage.modules.free_module_element.FreeModuleElement_generic_sparse method*), 198
denominator() (*sage.modules.free_module.FreeModule_generic_pid method*), 81
dense_coefficient_list() (*sage.modules.fp_graded.element.FPElement method*), 431
dense_coefficient_list() (*sage.modules.fp_graded.free_element.FreeGradedModuleElement method*), 399
dense_module() (*sage.modules.free_module.FreeModule_generic method*), 46
dense_vector() (*sage.modules.free_module_element.FreeModuleElement method*), 154
derivative() (*sage.modules.free_module_element.FreeModuleElement method*), 154
det() (*sage.modules.matrix_morphism.MatrixMorphism_abstract method*), 670
determinant() (*sage.modules.free_quadratic_module.FreeQuadraticModule_generic method*), 497
determinant() (*sage.modules.matrix_morphism.MatrixMorphism_abstract method*), 671
DiagonalModuleMorphism (*class in sage.modules.with_basis.morphism*), 274
diamond_cut() (*in module sage.modules.diamond_cutting*), 755
dict() (*sage.modules.free_module_element.FreeModuleElement method*), 155
dict() (*sage.modules.free_module_element.FreeModuleElement_generic_sparse method*), 198
diff() (*sage.modules.free_module_element.FreeModuleElement method*), 156
dimension() (*sage.modules.free_module.FreeModule_generic method*), 47
direct_sum() (*sage.modules.filtered_vector_space.FilteredVectorSpace_class method*), 581
direct_sum() (*sage.modules.free_module.FreeModule_generic method*), 48
direct_sum() (*sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method*), 520
direct_sum() (*sage.modules.multi_filtered_vector_space.MultiFilteredVectorSpace_class method*), 599
discriminant() (*sage.modules.free_module_integer.FreeModule_submodule_with_basis_integer method*), 483
discriminant() (*sage.modules.free_module.FreeModule_generic method*), 49
discriminant() (*sage.modules.free_quadratic_module.FreeQuadraticModule_generic method*), 498
discriminant_group() (*sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method*), 521
div() (*sage.modules.free_module_element.FreeModuleElement method*), 156
dot_product() (*sage.modules.free_module_element.FreeModuleElement method*), 158
dual() (*sage.modules.filtered_vector_space.FilteredVectorSpace_class method*), 582
dual() (*sage.modules.multi_filtered_vector_space.MultiFilteredVectorSpace_class method*), 600
dual_lattice() (*sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method*), 521

E

echelon_coordinate_vector() (*sage.modules.free_module.FreeModule_ambient method*), 27
echelon_coordinate_vector() (*sage.modules.free_module.FreeModule_submodule_with_basis_pid method*), 107
echelon_coordinates() (*sage.modules.free_module.FreeModule_ambient method*), 27
echelon_coordinates() (*sage.modules.free_module.FreeModule_submodule_field method*), 95
echelon_coordinates() (*sage.modules.free_module.FreeModule_submodule_with_basis_pid method*), 108
echelon_to_user_matrix() (*sage.modules.free_module.FreeModule_submodule_with_basis_pid method*), 109
echelonized_basis() (*sage.modules.free_module.FreeModule_ambient method*), 29
echelonized_basis() (*sage.modules.free_module.FreeModule_submodule_with_basis_pid method*), 110
echelonized_basis_matrix() (*sage.modules.free_module.FreeModule_ambient method*), 29
echelonized_basis_matrix() (*sage.modules.free_module.FreeModule_generic method*), 49
echelonized_basis_matrix() (*sage.modules.free_module.FreeModule_generic_field method*), 67
echelonized_basis_matrix() (*sage.modules.free_module.FreeModule_submodule_with_basis_pid method*), 110

EchelonMatrixKey (*class in sage.modules.free_module*), 19
eigenspaces() (*sage.modules.free_module_morphism.FreeModuleMorphism method*), 649
eigenvalues() (*sage.modules.free_module_morphism.FreeModuleMorphism method*), 651
eigenvectors() (*sage.modules.free_module_morphism.FreeModuleMorphism method*), 652
Element (*sage.modules.fg_pid.fgp_module.FGP_Module_class attribute*), 339
Element (*sage.modules.fg_pid.fgp_morphism.FGP_Homset_class attribute*), 370
Element (*sage.modules.fp_graded.free_homspace.FreeGradedModuleHomspace attribute*), 406
Element (*sage.modules.fp_graded.free_module.FreeGradedModule attribute*), 387
Element (*sage.modules.fp_graded.homspace.FPModuleHomspace attribute*), 460
Element (*sage.modules.fp_graded.module.FPModule attribute*), 408
Element (*sage.modules.free_module_pseudohomspace.FreeModulePseudoHomspace attribute*), 697
Element (*sage.modules.torsion_quadratic_module.TorsionQuadraticModule attribute*), 557
Element (*sage.modules.with_basis.representation.QuotientRepresentation attribute*), 305
Element (*sage.modules.with_basis.representation.SchurFunctorRepresentation attribute*), 330
element() (*sage.modules.free_module_element.FreeModuleElement method*), 159
element_class() (*in module sage.modules.free_module*), 132
element_from_coordinates() (*sage.modules.fp_graded.free_module.FreeGradedModule method*), 390
element_from_coordinates() (*sage.modules.fp_graded.module.FPModule method*), 412
endomorphism_ring() (*sage.modules.module.Module method*), 14
enumerate_close_vectors() (*sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method*), 522
enumerate_short_vectors() (*sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method*), 523
export_module_definition() (*sage.modules.fp_graded.steenrod.module.SteenrodModuleMixin method*), 468
exterior_power() (*sage.modules.filtered_vector_space.FilteredVectorSpace_class method*), 583
exterior_power() (*sage.modules.multi_filtered_vector_space.MultiFilteredVectorSpace_class method*), 600
exterior_power() (*sage.modules.with_basis.representation.Representation_abstract method*), 316

F

fcp() (*sage.modules.matrix_morphism.MatrixMorphism_abstract method*), 672
fft() (*sage.modules.vector_double_dense.Vector_double_dense method*), 739
FGP_Element (*class in sage.modules.fg_pid.fgp_element*), 366
FGP_Homset() (*in module sage.modules.fg_pid.fgp_morphism*), 370
FGP_Homset_class (*class in sage.modules.fg_pid.fgp_morphism*), 370
FGP_Module() (*in module sage.modules.fg_pid.fgp_module*), 338
FGP_Module_class (*class in sage.modules.fg_pid.fgp_module*), 338
FGP_Morphism (*class in sage.modules.fg_pid.fgp_morphism*), 371
FilteredVectorSpace() (*in module sage.modules.filtered_vector_space*), 579
FilteredVectorSpace_class (*class in sage.modules.filtered_vector_space*), 580
find_subrepresentation() (*sage.modules.with_basis.representation.Representation_abstract method*), 316
FiniteDimensionalInvariantModule (*class in sage.modules.with_basis.invariant*), 256
FiniteDimensionalInvariantModule.Element (*class in sage.modules.with_basis.invariant*), 262
FiniteDimensionalTwistedInvariantModule (*class in sage.modules.with_basis.invariant*), 263
FiniteDimensionalTwistedInvariantModule.Element (*class in sage.modules.with_basis.invariant*), 268
FiniteFieldsubspace_iterator (*class in sage.modules.finite_submodule_iter*), 767
FiniteFieldsubspace_projPoint_iterator (*class in sage.modules.finite_submodule_iter*), 768
FiniteZZsubmodule_iterator (*class in sage.modules.finite_submodule_iter*), 768
fp_module() (*sage.modules.fp_graded.free_morphism.FreeGradedModuleMorphism method*), 404
fp_module() (*sage.modules.fp_graded.morphism.FPModuleMorphism method*), 438
FPElement (*class in sage.modules.fp_graded.element*), 430

FPModule (*class in sage.modules.fp_graded.module*), 406
FPModuleHomspace (*class in sage.modules.fp_graded.homspace*), 460
FPModuleMorphism (*class in sage.modules.fp_graded.morphism*), 435
free_cover() (*sage.modules.quotient_module.QuotientModule_free_ambient method*), 244
free_module() (*sage.modules.free_module.FreeModule_generic method*), 49
free_module_element() (*in module sage.modules.free_module_element*), 202
free_relations() (*sage.modules.quotient_module.QuotientModule_free_ambient method*), 245
free_resolution() (*sage.modules.free_module.Module_free_ambient method*), 118
FreeGradedModule (*class in sage.modules.fp_graded.free_module*), 385
FreeGradedModuleElement (*class in sage.modules.fp_graded.free_element*), 397
FreeGradedModuleHomspace (*class in sage.modules.fp_graded.free_homspace*), 406
FreeGradedModuleMorphism (*class in sage.modules.fp_graded.free_morphism*), 402
FreeModule() (*in module sage.modules.free_module*), 20
FreeModule_ambient (*class in sage.modules.free_module*), 25
FreeModule_ambient_domain (*class in sage.modules.free_module*), 33
FreeModule_ambient_field (*class in sage.modules.free_module*), 36
FreeModule_ambient_field_quotient (*class in sage.modules.quotient_module*), 236
FreeModule_ambient_pid (*class in sage.modules.free_module*), 36
FreeModule_generic (*class in sage.modules.free_module*), 37
FreeModule_generic_domain (*class in sage.modules.free_module*), 65
FreeModule_generic_field (*class in sage.modules.free_module*), 65
FreeModule_generic_pid (*class in sage.modules.free_module*), 81
FreeModule_submodule_field (*class in sage.modules.free_module*), 93
FreeModule_submodule_pid (*class in sage.modules.free_module*), 96
FreeModule_submodule_with_basis_field (*class in sage.modules.free_module*), 98
FreeModule_submodule_with_basis_integer (*class in sage.modules.free_module_integer*), 477
FreeModule_submodule_with_basis_pid (*class in sage.modules.free_module*), 100
FreeModuleElement (*class in sage.modules.free_module_element*), 138
FreeModuleElement_generic_dense (*class in sage.modules.free_module_element*), 196
FreeModuleElement_generic_sparse (*class in sage.modules.free_module_element*), 198
FreeModuleFactory (*class in sage.modules.free_module*), 25
FreeModuleHomspace (*class in sage.modules.free_module_homspace*), 640
FreeModuleMorphism (*class in sage.modules.free_module_morphism*), 647
FreeModulePseudoHomspace (*class in sage.modules.free_module_pseudohomspace*), 697
FreeModulePseudoMorphism (*class in sage.modules.free_module_pseudomorphism*), 699
FreeQuadraticModule() (*in module sage.modules.free_quadratic_module*), 494
FreeQuadraticModule_ambient (*class in sage.modules.free_quadratic_module*), 495
FreeQuadraticModule_ambient_domain (*class in sage.modules.free_quadratic_module*), 495
FreeQuadraticModule_ambient_field (*class in sage.modules.free_quadratic_module*), 496
FreeQuadraticModule_ambient_pid (*class in sage.modules.free_quadratic_module*), 496
FreeQuadraticModule_generic (*class in sage.modules.free_quadratic_module*), 496
FreeQuadraticModule_generic_field (*class in sage.modules.free_quadratic_module*), 501
FreeQuadraticModule_generic_pid (*class in sage.modules.free_quadratic_module*), 503
FreeQuadraticModule_integer_symmetric (*class in sage.modules.free_quadratic_module_integer_symmetric*), 515
FreeQuadraticModule_submodule_field (*class in sage.modules.free_quadratic_module*), 505
FreeQuadraticModule_submodule_pid (*class in sage.modules.free_quadratic_module*), 506
FreeQuadraticModule_submodule_with_basis_field (*class in sage.modules.free_quadratic_module*), 507
FreeQuadraticModule_submodule_with_basis_pid (*class in sage.modules.free_quadratic_module*), 509
function() (*sage.modules.free_module_element.FreeModuleElement_generic_dense method*), 196

G

gen() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 344
gen() (*sage.modules.fp_graded.free_module.FreeGradedModule method*), 391

gen() (*sage.modules.fp_graded.module.FPModule method*), 413
gen() (*sage.modules.free_module.FreeModule_ambient method*), 30
gen() (*sage.modules.free_module.FreeModule_generic method*), 50
gen() (*sage.modules.quotient_module.QuotientModule_free_ambient method*), 246
gen() (*sage.modules.submodule.Submodule_free_ambient method*), 234
generator() (*sage.modules.fp_graded.free_module.FreeGradedModule method*), 391
generator() (*sage.modules.fp_graded.module.FPModule method*), 414
generator_degrees() (*sage.modules.fp_graded.free_module.FreeGradedModule method*), 392
generator_degrees() (*sage.modules.fp_graded.module.FPModule method*), 414
generators() (*sage.modules.fp_graded.free_module.FreeGradedModule method*), 392
generators() (*sage.modules.fp_graded.module.FPModule method*), 415
generators_matrix() (*sage.modules.submodule.Submodule_free_ambient method*), 235
gens() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 345
gens() (*sage.modules.fp_graded.module.FPModule method*), 416
gens() (*sage.modules.free_module.FreeModule_generic method*), 50
gens() (*sage.modules.quotient_module.QuotientModule_free_ambient method*), 246
gens() (*sage.modules.submodule.Submodule_free_ambient method*), 235
gens() (*sage.modules.torsion_quadratic_module.TorsionQuadraticModule method*), 559
gens_to_smith() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 346
gens_vector() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 347
genus() (*sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method*), 524
genus() (*sage.modules.torsion_quadratic_module.TorsionQuadraticModule method*), 559
get() (*sage.modules.free_module_element.FreeModuleElement method*), 160
get_degree() (*sage.modules.filtered_vector_space.FilteredVectorSpace_class method*), 584
get_degree() (*sage.modules.multi_filtered_vector_space.MultiFilteredVectorSpace_class method*), 601
get_filtration() (*sage.modules.multi_filtered_vector_space.MultiFilteredVectorSpace_class method*), 601
graded() (*sage.modules.filtered_vector_space.FilteredVectorSpace_class method*), 585
graded() (*sage.modules.multi_filtered_vector_space.MultiFilteredVectorSpace_class method*), 602
graded_free_resolution() (*sage.modules.free_module.Module_free_ambient method*), 118
gram_matrix() (*sage.modules.free_module.FreeModule_generic method*), 51
gram_matrix() (*sage.modules.free_quadratic_module.FreeQuadraticModule_generic method*), 498
gram_matrix_bilinear() (*sage.modules.torsion_quadratic_module.TorsionQuadraticModule method*), 561
gram_matrix_quadratic() (*sage.modules.torsion_quadratic_module.TorsionQuadraticModule method*), 562
gram_schmidt() (*in module sage.modules.misc*), 769

H

hamming_weight() (*sage.modules.free_module_element.FreeModuleElement method*), 160
hamming_weight() (*sage.modules.free_module_element.FreeModuleElement_generic_sparse method*), 199
hamming_weight() (*sage.modules.vector_mod2_dense.Vector_mod2_dense method*), 709
has_canonical_map_to() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 349
has_relations() (*sage.modules.fp_graded.free_module.FreeGradedModule method*), 392
has_relations() (*sage.modules.fp_graded.module.FPModule method*), 416
has_user_basis() (*sage.modules.free_module.FreeModule_generic method*), 51
has_user_basis() (*sage.modules.free_module.FreeModule_submodule_field method*), 96
has_user_basis() (*sage.modules.free_module.FreeModule_submodule_pid method*), 97
has_user_basis() (*sage.modules.free_module.FreeModule_submodule_with_basis_pid method*), 111
hermitian_inner_product() (*sage.modules.free_module_element.FreeModuleElement method*), 160
HKZ() (*sage.modules.free_module_integer.FreeModule_submodule_with_basis_integer method*), 479

hom() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 350
hom() (*sage.modules.free_module.FreeModule_generic method*), 52
homology() (*sage.modules.fp_graded.morphism.FPModuleMorphism method*), 439

I

identity() (*sage.modules.fp_graded.homspace.FPModuleHomspace method*), 461
identity() (*sage.modules.free_module_homspace.FreeModuleHomspace method*), 641
im_gens() (*sage.modules.fg_pid.fgp_morphism.FGP_Morphism method*), 372
image() (*sage.modules.fg_pid.fgp_morphism.FGP_Morphism method*), 372
image() (*sage.modules.fp_graded.morphism.FPModuleMorphism method*), 440
image() (*sage.modules.fp_graded.steenrod.morphism.SteenrodFPModuleMorphism method*), 471
image() (*sage.modules.matrix_morphism.MatrixMorphism_abstract method*), 672
index_in() (*sage.modules.free_module.FreeModule_generic_pid method*), 82
index_in_saturation() (*sage.modules.free_module.FreeModule_generic_pid method*), 83
index_map() (*sage.modules.tensor_operations.TensorOperation method*), 761
index_set() (*sage.modules.multi_filtered_vector_space.MultiFilteredVectorSpace_class method*), 602
IndexedFreeModuleElement (*class in sage.modules.with_basis.indexed_element*), 253
inner_product() (*sage.modules.free_module_element.FreeModuleElement method*), 162
inner_product() (*sage.modules.torsion_quadratic_module.TorsionQuadraticModuleElement method*), 574
inner_product_matrix() (*sage.modules.free_module.FreeModule_generic method*), 52
inner_product_matrix() (*sage.modules.free_quadratic_module.FreeQuadraticModule_generic method*), 499
InnerProductSpace() (*in module sage.modules.free_quadratic_module*), 511
IntegerLattice() (*in module sage.modules.free_module_integer*), 488
integral() (*sage.modules.free_module_element.FreeModuleElement method*), 164
IntegralLattice() (*in module sage.modules.free_quadratic_module_integer_symmetric*), 539
IntegralLatticeDirectSum() (*in module sage.modules.free_quadratic_module_integer_symmetric*), 544
IntegralLatticeGluing() (*in module sage.modules.free_quadratic_module_integer_symmetric*), 546
integrate() (*sage.modules.free_module_element.FreeModuleElement method*), 165
intersection() (*sage.modules.free_module.FreeModule_generic_field method*), 68
intersection() (*sage.modules.free_module.FreeModule_generic_pid method*), 83
intersection() (*sage.modules.with_basis.subquotient.SubmoduleWithBasis method*), 298
inv_fft() (*sage.modules.vector_double_dense.Vector_double_dense method*), 741
invariant_module() (*sage.modules.with_basis.representation.Representation_abstract method*), 317
invariants() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 353
inverse() (*sage.modules.matrix_morphism.MatrixMorphism_abstract method*), 674
inverse_image() (*sage.modules.fg_pid.fgp_morphism.FGP_Morphism method*), 373
inverse_image() (*sage.modules.free_module_morphism.FreeModuleMorphism method*), 653
is_ambient() (*sage.modules.free_module.FreeModule_ambient method*), 30
is_ambient() (*sage.modules.free_module.FreeModule_generic method*), 53
is_ambient() (*sage.modules.free_module.FreeModule_submodule_with_basis_field method*), 100
is_bijective() (*sage.modules.matrix_morphism.MatrixMorphism_abstract method*), 678
is_constant() (*sage.modules.filtered_vector_space.FilteredVectorSpace_class method*), 585
is_constant() (*sage.modules.multi_filtered_vector_space.MultiFilteredVectorSpace_class method*), 603
is_dense() (*sage.modules.free_module_element.FreeModuleElement method*), 165
is_dense() (*sage.modules.free_module.FreeModule_generic method*), 54
is_equal_function() (*sage.modules.matrix_morphism.MatrixMorphism_abstract method*), 679
is_equal_subspace() (*sage.modules.with_basis.subquotient.SubmoduleWithBasis method*), 298
is_even() (*sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method*), 524
is_exact() (*sage.modules.free_module.Module_free_ambient method*), 119
is_exhaustive() (*sage.modules.filtered_vec-*

tor_space.FilteredVectorSpace_class method), 586
is_exhaustive() (sage.modules.multi_filtered_vector_space.MultiFilteredVectorSpace_class method), 603
is_FGP_Module() (in module sage.modules.fg_pid.fgp_module), 363
is_FilteredVectorSpace() (in module sage.modules.filtered_vector_space), 594
is_finite() (sage.modules.fg_pid.fgp_module.FGP_Module_class method), 354
is_finite() (sage.modules.free_module.FreeModule_generic method), 54
is_FreeModule() (in module sage.modules.free_module), 133
is_FreeModuleElement() (in module sage.modules.free_module_element), 212
is_FreeModuleHomspace() (in module sage.modules.free_module_homspace), 642
is_FreeModuleMorphism() (in module sage.modules.free_module_morphism), 662
is_FreeQuadraticModule() (in module sage.modules.free_quadratic_module), 513
is_full() (sage.modules.free_module.FreeModule_generic method), 54
is_genus() (sage.modules.torsion_quadratic_module.TorsionQuadraticModule method), 562
is_identity() (sage.modules.fp_graded.morphism.FPModuleMorphism method), 442
is_identity() (sage.modules.matrix_morphism.MatrixMorphism_abstract method), 681
is_injective() (sage.modules.fp_graded.morphism.FPModuleMorphism method), 444
is_injective() (sage.modules.fp_graded.steenrod.morphism.SteenrodFPModuleMorphism method), 473
is_injective() (sage.modules.free_module_morphism.BaseIsomorphism1D method), 644
is_injective() (sage.modules.matrix_morphism.MatrixMorphism method), 664
is_invertible() (sage.modules.vector_space_morphism.VectorSpaceMorphism method), 629
is_irreducible() (sage.modules.with_basis.representation.Representation_abstract method), 318
is_MatrixMorphism() (in module sage.modules.matrix_morphism), 695
is_Module() (in module sage.modules.module), 15
is_primitive() (sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method), 525
is_separating() (sage.modules.filtered_vector_space.FilteredVectorSpace_class method), 587
is_separating() (sage.modules.multi_filtered_vector_space.MultiFilteredVectorSpace_class method), 604
is_sparse() (sage.modules.free_module_element.FreeModuleElement method), 166
is_sparse() (sage.modules.free_module.Module_free_ambient method), 119
is_submodule() (sage.modules.fg_pid.fgp_module.FGP_Module_class method), 354
is_submodule() (sage.modules.free_module.FreeModule_generic method), 55
is_submodule() (sage.modules.free_module.Module_free_ambient method), 120
is_submodule() (sage.modules.with_basis.subquotient.SubmoduleWithBasis method), 300
is_subspace() (sage.modules.free_module.FreeModule_generic_field method), 69
is_surjective() (sage.modules.fp_graded.morphism.FPModuleMorphism method), 444
is_surjective() (sage.modules.free_module_morphism.BaseIsomorphism1D method), 645
is_surjective() (sage.modules.matrix_morphism.MatrixMorphism method), 665
is_trivial() (sage.modules.fp_graded.free_module.FreeGradedModule method), 393
is_trivial() (sage.modules.fp_graded.module.FPModule method), 417
is_unimodular() (sage.modules.free_module_integer.FreeModule_submodule_with_basis_integer method), 483
is_vector() (sage.modules.free_module_element.FreeModuleElement method), 166
is_VectorSpace() (in module sage.modules.module), 15
is_VectorSpaceHomspace() (in module sage.modules.vector_space_homspace), 616
is_VectorSpaceMorphism() (in module sage.modules.vector_space_morphism), 630
is_zero() (sage.modules.fp_graded.morphism.FPModuleMorphism method), 445
is_zero() (sage.modules.matrix_morphism.MatrixMorphism_abstract method), 683
items() (sage.modules.free_module_element.FreeModuleElement method), 166
items() (sage.modules.free_module_element.FreeModuleElement_generic_sparse method), 199
iteritems() (sage.modules.free_module_element.FreeModuleElement method), 167
iteritems() (sage.modules.free_module_element.FreeModuleElement_generic_sparse method), 200

J

jacobi() (in module sage.modules.diamond_cutting), 756

K

kernel() (*sage.modules.fg_pid.fgp_morphism.FGP_Morphism method*), 374
kernel() (*sage.modules.matrix_morphism.MatrixMorphism_abstract method*), 684
kernel_inclusion() (*sage.modules.fp_graded.morphism.FPModuleMorphism method*), 446
kernel_inclusion() (*sage.modules.fp_graded.steenrod.morphism.SteenrodFPModuleMorphism method*), 474

L

lift() (*sage.modules.fg_pid.fgp_element.FGP_Element method*), 368
lift() (*sage.modules.fg_pid.fgp_morphism.FGP_Morphism method*), 375
lift() (*sage.modules.fp_graded.morphism.FPModuleMorphism method*), 448
lift() (*sage.modules.free_module_element.FreeModuleElement method*), 167
lift() (*sage.modules.free_module_morphism.FreeModuleMorphism method*), 657
lift() (*sage.modules.free_module.FreeModule_submodule_with_basis_pid method*), 111
lift() (*sage.modules.quotient_module.FreeModule_ambient_field_quotient method*), 240
lift() (*sage.modules.with_basis.subquotient.QuotientModuleWithBasis method*), 296
lift() (*sage.modules.with_basis.subquotient.SubmoduleWithBasis method*), 302
lift_centered() (*sage.modules.free_module_element.FreeModuleElement method*), 168
lift_map() (*sage.modules.quotient_module.FreeModule_ambient_field_quotient method*), 241
lift_to_free() (*sage.modules.fp_graded.element.FPElement method*), 431
lift_to_free() (*sage.modules.fp_graded.free_element.FreeGradedModuleElement method*), 399
linear_combination_of_basis() (*sage.modules.free_module.FreeModule_ambient method*), 31
linear_combination_of_basis() (*sage.modules.free_module.FreeModule_submodule_with_basis_pid method*), 112
linear_combination_of_smith_form_gens() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 355
linear_dependence() (*sage.modules.free_module.FreeModule_generic_field method*), 69
linear_transformation() (*in module sage.modules.vector_space_morphism*), 631
list() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 356
list() (*sage.modules.free_module_element.FreeModuleElement method*), 169
list() (*sage.modules.free_module_element.FreeModuleElement_generic_dense method*), 197
list() (*sage.modules.free_module_element.FreeModuleElement_generic_sparse method*), 200
list() (*sage.modules.vector_integer_dense.Vector_integer_dense method*), 708
list() (*sage.modules.vector_mod2_dense.Vector_mod2_dense method*), 710
list() (*sage.modules.vector_rational_dense.Vector_rational_dense method*), 714
list_from_positions() (*sage.modules.free_module_element.FreeModuleElement method*), 169
LLL() (*sage.modules.free_module_integer.FreeModule_submodule_with_basis_integer method*), 480
LLL() (*sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method*), 516
lll() (*sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method*), 526
local_modification() (*in module sage.modules.free_quadratic_module_integer_symmetric*), 554

M

make_FreeModuleElement_generic_dense() (*in module sage.modules.free_module_element*), 212
make_FreeModuleElement_generic_dense_v1() (*in module sage.modules.free_module_element*), 213
make_FreeModuleElement_generic_sparse() (*in module sage.modules.free_module_element*), 213
make_FreeModuleElement_generic_sparse_v1() (*in module sage.modules.free_module_element*), 213
matrix() (*sage.modules.free_module_pseudomorphism.FreeModulePseudoMorphism method*), 700
matrix() (*sage.modules.free_module.FreeModule_generic method*), 56
matrix() (*sage.modules.matrix_morphism.MatrixMorphism method*), 666
matrix() (*sage.modules.matrix_morphism.MatrixMorphism_abstract method*), 685
matrix() (*sage.modules.submodule.Submodule_free_ambient method*), 235
matrix_space() (*sage.modules.free_module_pseudohomspace.FreeModulePseudoHomspace method*), 698

MatrixMorphism (*class in sage.modules.matrix_morphism*), 664
MatrixMorphism_abstract (*class in sage.modules.matrix_morphism*), 667
max() (*sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method*), 526
max_degree() (*sage.modules.filtered_vector_space.FilteredVectorSpace_class method*), 587
max_degree() (*sage.modules.multi_filtered_vector_space.MultiFilteredVectorSpace_class method*), 604
maximal_overlattice() (*sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method*), 527
maximum() (*sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method*), 527
mean() (*sage.modules.vector_double_dense.Vector_double_dense method*), 741
min() (*sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method*), 528
min_degree() (*sage.modules.filtered_vector_space.FilteredVectorSpace_class method*), 587
min_degree() (*sage.modules.multi_filtered_vector_space.MultiFilteredVectorSpace_class method*), 604
minimal_presentation() (*sage.modules.fp_graded.free_module.FreeGradedModule method*), 393
minimal_presentation() (*sage.modules.fp_graded.module.FPModule method*), 418
minimum() (*sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method*), 528
Mod() (*sage.modules.free_module_element.FreeModuleElement method*), 138
module
    sage.modules.complex_double_vector, 753
    sage.modules.diamond_cutting, 755
    sage.modules.fg_pid.fgp_element, 366
    sage.modules.fg_pid.fgp_module, 333
    sage.modules.fg_pid.fgp_morphism, 370
    sage.modules.filtered_vector_space, 576
    sage.modules.finite_submodule_iter, 766
    sage.modules.fp_graded.element, 430
    sage.modules.fp_graded.free_element, 397
    sage.modules.fp_graded.free_homspace, 405
    sage.modules.fp_graded.free_module, 377
    sage.modules.fp_graded.free_morphism, 402
    sage.modules.fp_graded.homspace, 460
    sage.modules.fp_graded.module, 406
    sage.modules.fp_graded.morphism, 435
    sage.modules.fp_graded.steenrod.module, 464
    sage.modules.fp_graded.steenrod.morphism, 470
    sage.modules.free_module, 16
    sage.modules.free_module_element, 136
    sage.modules.free_module_homspace, 638
    sage.modules.free_module_integer, 477
    sage.modules.free_module_morphism, 644
    sage.modules.free_module_pseudo_homspace, 697
    sage.modules.free_module_pseudomorphism, 699
    sage.modules.free_quadratic_module, 493
    sage.modules.free_quadratic_module_integer_symmetric, 514
    sage.modules.matrix_morphism, 663
    sage.modules.misc, 769
    sage.modules.module, 11
    sage.modules.multi_filtered_vector_space, 596
    sage.modules.quotient_module, 236
    sage.modules.real_double_vector, 754
    sage.modules.submodule, 232
    sage.modules.tensor_operations, 758
    sage.modules.torsion_quadratic_module, 555
    sage.modules.tutorial_free_modules, 3
    sage.modules.vector_callable_symbolic_dense, 737
    sage.modules.vector_complex_double_dense, 752
    sage.modules.vector_double_dense, 738
    sage.modules.vector_integer_dense, 707
    sage.modules.vector_integer_sparse, 708
    sage.modules.vector_mod2_dense, 708
    sage.modules.vector_modn_dense, 710
    sage.modules.vector_modn_sparse, 713
    sage.modules.vector_numpy_dense, 749
    sage.modules.vector_numpy_integer_dense, 751
    sage.modules.vector_rational_dense, 713
    sage.modules.vector_rational_sparse, 714
    sage.modules.vector_real_double_dense, 747
    sage.modules.vector_space_homspace, 611
    sage.modules.vector_space_morphism, 617
    sage.modules.vector_symbolic_dense, 714
    sage.modules.vector_symbolic_sparse, 725
    sage.modules.with_basis.all, 249

sage.modules.with_basis.cell_module, 249
sage.modules.with_basis.indexed_element, 253
sage.modules.with_basis.invariant, 256
sage.modules.with_basis.morphism, 271
sage.modules.with_basis.representation, 305
sage.modules.with_basis.subquotient, 295
Module (*class in sage.modules.module*), 12
Module_free_ambient (*class in sage.modules.free_module*), 116
ModuleMorphism (*class in sage.modules.with_basis.morphism*), 274
ModuleMorphismByLinearity (*class in sage.modules.with_basis.morphism*), 275
ModuleMorphismFromFunction (*class in sage.modules.with_basis.morphism*), 276
ModuleMorphismFromMatrix (*class in sage.modules.with_basis.morphism*), 277
monic() (*sage.modules.free_module_element.FreeModuleElement method*), 170
monomial() (*sage.modules.fp_graded.module.FPModule method*), 419
monomial_coefficients() (*sage.modules.free_module_element.FreeModuleElement method*), 170
monomial_coefficients() (*sage.modules.with_basis.indexed_element.IndexedFreeModuleElement method*), 253
MultiFilteredVectorSpace() (*in module sage.modules.multi_filtered_vector_space*), 596
MultiFilteredVectorSpace_class (*class in sage.modules.multi_filtered_vector_space*), 597

N

n_vectors() (*sage.modules.tensor_operations.VectorCollection method*), 764
NaturalMatrixRepresentation (*class in sage.modules.with_basis.representation*), 305
ngens() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 356
ngens() (*sage.modules.free_module.FreeModule_generic method*), 57
nintegral() (*sage.modules.free_module_element.FreeModuleElement method*), 171
nintegrate() (*sage.modules.free_module_element.FreeModuleElement method*), 172
nonembedded_free_module() (*sage.modules.free_module.FreeModule_generic method*), 57
nonzero_bilinear_form() (*sage.modules.with_basis.cell_module.CellModule method*), 251
nonzero_positions() (*sage.modules.free_module_element.FreeModuleElement method*), 173
nonzero_positions() (*sage.modules.free_module_element.FreeModuleElement_generic_sparse method*), 201
norm() (*sage.modules.free_module_element.FreeModuleElement method*), 173
norm() (*sage.modules.vector_double_dense.Vector_double_dense method*), 741
normal_form() (*sage.modules.torsion_quadratic_module.TorsionQuadraticModule method*), 563
normalize() (*sage.modules.fp_graded.element.FPElement method*), 432
normalize_degree() (*in module sage.modules.filtered_vector_space*), 595
normalized() (*sage.modules.free_module_element.FreeModuleElement method*), 175
nullity() (*sage.modules.matrix_morphism.MatrixMorphism_abstract method*), 686
numerical_approx() (*sage.modules.free_module_element.FreeModuleElement method*), 176
numerical_approx() (*sage.modules.free_module_element.FreeModuleElement_generic_sparse method*), 201
numpy() (*sage.modules.free_module_element.FreeModuleElement method*), 180
numpy() (*sage.modules.vector_numpy_dense.Vector_numpy_dense method*), 750

O

on_basis() (*sage.modules.with_basis.morphism.ModuleMorphismByLinearity method*), 276
one_basis() (*sage.modules.with_basis.representation.Representation_ExteriorAlgebra method*), 309
optimized() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 356
ore_module() (*sage.modules.free_module_pseudomorphism.FreeModulePseudoMorphism method*), 701
ore_ring() (*sage.modules.free_module_pseudohomspace.FreeModulePseudoHomspace method*), 699
orthogonal_complement() (*sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method*), 529
orthogonal_group() (*sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method*), 530
orthogonal_group() (*sage.modules.torsion_quadratic_module.TorsionQuadraticModule method*), 566
orthogonal_submodule_to() (*sage.modules.torsion_quadratic_module.TorsionQuadraticMod-*

ule method), 568
outer_product() (*sage.modules.free_module_element.FreeModuleElement method*), 182
overlattice() (*sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method*), 533

P

pairwise_product() (*sage.modules.free_module_element.FreeModuleElement method*), 186
plane_inequality() (*in module sage.modules.diamond_cutting*), 758
plot() (*sage.modules.free_module_element.FreeModuleElement method*), 187
plot_step() (*sage.modules.free_module_element.FreeModuleElement method*), 190
pointwise_inverse() (*sage.modules.with_basis.morphism.PointwiseInverseFunction method*), 279
pointwise_inverse_function() (*in module sage.modules.with_basis.morphism*), 294
PointwiseInverseFunction (*class in sage.modules.with_basis.morphism*), 278
preimage() (*sage.modules.tensor_operations.TensorOperation method*), 763
preimage() (*sage.modules.with_basis.morphism.TriangularModuleMorphism method*), 290
preimage_representative() (*sage.modules.free_module_morphism.FreeModuleMorphism method*), 659
prepare() (*in module sage.modules.free_module_element*), 214
presentation() (*sage.modules.filtered_vector_space.FilteredVectorSpace_class method*), 588
primary_part() (*sage.modules.torsion_quadratic_module.TorsionQuadraticModule method*), 569
prod() (*sage.modules.vector_double_dense.Vector_double_dense method*), 744
product_by_coercion() (*sage.modules.with_basis.representation.Representation method*), 308
product_on_basis() (*sage.modules.with_basis.representation.Representation_ExteriorAlgebra method*), 310
profile() (*sage.modules.fp_graded.steenrod.module.SteenrodModuleMixin method*), 469
profile() (*sage.modules.fp_graded.steenrod.morphism.SteenrodFPModuleMorphism method*), 475
project() (*sage.modules.with_basis.invariant.FiniteDimensionalTwistedInvariantModule method*), 269
project_ambient() (*sage.modules.with_basis.invariant.FiniteDimensionalTwistedInvariantModule method*), 269
projection_matrix() (*sage.modules.with_basis.invariant.FiniteDimensionalTwistedInvariantModule method*), 270
pseudoHom() (*sage.modules.free_module.FreeModule_generic method*), 57
pseudohom() (*sage.modules.free_module.FreeModule_generic method*), 58
pushforward() (*sage.modules.free_module_morphism.FreeModuleMorphism method*), 661

Q

q() (*sage.modules.torsion_quadratic_module.TorsionQuadraticModuleElement method*), 575
quadratic_form() (*sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method*), 534
quadratic_product() (*sage.modules.torsion_quadratic_module.TorsionQuadraticModuleElement method*), 575
QuadraticSpace() (*in module sage.modules.free_quadratic_module*), 512
quotient_abstract() (*sage.modules.free_module.FreeModule_generic_field method*), 72
quotient_map() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 358
quotient_map() (*sage.modules.quotient_module.FreeModule_ambient_field_quotient method*), 241
quotient_module() (*sage.modules.free_module.FreeModule_generic_field method*), 73
quotient_module() (*sage.modules.free_module.FreeModule_generic_pid method*), 85
quotient_module() (*sage.modules.free_module.Module_free_ambient method*), 121
quotient_representation() (*sage.modules.with_basis.representation.Representation_abstract method*), 319
QuotientModule_free_ambient (*class in sage.modules.quotient_module*), 242
QuotientModuleWithBasis (*class in sage.modules.with_basis.subquotient*), 295
QuotientRepresentation (*class in sage.modules.with_basis.representation*), 305

R

radical() (*sage.modules.with_basis.cell_module.CellModule method*), 251
radical_basis() (*sage.modules.with_basis.cell_module.CellModule method*), 252
random_deformation() (*sage.modules.filtered_vector_space.FilteredVectorSpace_class method*), 588
random_deformation() (*sage.modules.multi_filtered_vector_space.MultiFilteredVec-*

torSpace_class method), 605
random_element() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 358
random_element() (*sage.modules.free_module.FreeModule_ambient method*), 32
random_element() (*sage.modules.free_module.FreeModule_generic method*), 61
random_fgp_module() (*in module sage.modules.fg_pid.fgp_module*), 364
random_fgp_morphism_0() (*in module sage.modules.fg_pid.fgp_module*), 365
random_vector() (*in module sage.modules.free_module_element*), 216
rank() (*sage.modules.free_module.FreeModule_generic method*), 62
rank() (*sage.modules.matrix_morphism.MatrixMorphism_abstract method*), 687
RealDoubleVectorSpace_class (*class in sage.modules.free_module*), 130
reduce() (*sage.modules.with_basis.subquotient.SubmoduleWithBasis method*), 302
reduced_basis (*sage.modules.free_module_integer.FreeModule_submodule_with_basis_integer property*), 484
ReflectionRepresentation (*class in sage.modules.with_basis.representation*), 305
RegularRepresentation (*class in sage.modules.with_basis.representation*), 306
relation() (*sage.modules.fp_graded.module.FPModule method*), 420
relations() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 359
relations() (*sage.modules.fp_graded.free_module.FreeGradedModule method*), 394
relations() (*sage.modules.fp_graded.module.FPModule method*), 420
relations() (*sage.modules.free_module.FreeModule_generic method*), 62
relations() (*sage.modules.free_module.FreeModule_submodule_with_basis_pid method*), 113
relations() (*sage.modules.quotient_module.FreeModule_ambient_field_quotient method*), 242
relations() (*sage.modules.quotient_module.QuotientModule_free_ambient method*), 247
relations() (*sage.modules.submodule.Submodule_free_ambient method*), 236
relations_matrix() (*sage.modules.free_module.Module_free_ambient method*), 121
Representation (*class in sage.modules.with_basis.representation*), 306
Representation_abstract (*class in sage.modules.with_basis.representation*), 310
Representation_abstract.Element (*class in sage.modules.with_basis.representation*), 311
Representation_Exterior (*class in sage.modules.with_basis.representation*), 309
Representation_ExteriorAlgebra (*class in sage.modules.with_basis.representation*), 309
representation_matrix() (*sage.modules.with_basis.representation.Representation_abstract method*), 320
Representation_Symmetric (*class in sage.modules.with_basis.representation*), 310
Representation_Tensor (*class in sage.modules.with_basis.representation*), 310
Representation_Tensor.Element (*class in sage.modules.with_basis.representation*), 310
resolution() (*sage.modules.fp_graded.free_module.FreeGradedModule method*), 394
resolution() (*sage.modules.fp_graded.module.FPModule method*), 421
resolution() (*sage.modules.fp_graded.steenrod.module.SteenrodFPModule method*), 465
restrict() (*sage.modules.matrix_morphism.MatrixMorphism_abstract method*), 687
restrict_codomain() (*sage.modules.matrix_morphism.MatrixMorphism_abstract method*), 690
restrict_domain() (*sage.modules.matrix_morphism.MatrixMorphism_abstract method*), 692
retract() (*sage.modules.free_module.FreeModule_submodule_with_basis_pid method*), 114
retract() (*sage.modules.with_basis.subquotient.QuotientModuleWithBasis method*), 296
retract() (*sage.modules.with_basis.subquotient.SubmoduleWithBasis method*), 303
row() (*sage.modules.free_module_element.FreeModuleElement method*), 190

S

sage.modules.complex_double_vector module, 753
sage.modules.diamond_cutting module, 755
sage.modules.fg_pid.fgp_element module, 366
sage.modules.fg_pid.fgp_module module, 333
sage.modules.fg_pid.fgp_morphism module, 370
sage.modules.filtered_vector_space module, 576
sage.modules.finite_submodule_iter module, 766
sage.modules.fp_graded.element module, 430
sage.modules.fp_graded.free_element module, 397
sage.modules.fp_graded.free_homspace

module, 405
sage.modules.fp_graded.free_module
module, 377
sage.modules.fp_graded.free_morphism
module, 402
sage.modules.fp_graded.homspace
module, 460
sage.modules.fp_graded.module
module, 406
sage.modules.fp_graded.morphism
module, 435
sage.modules.fp_graded.steenrod.module
module, 464
sage.modules.fp_graded.steenrod.morphism
module, 470
sage.modules.free_module
module, 16
sage.modules.free_module_element
module, 136
sage.modules.free_module_homspace
module, 638
sage.modules.free_module_integer
module, 477
sage.modules.free_module_morphism
module, 644
sage.modules.free_module_pseudohomspace
module, 697
sage.modules.free_module_pseudomorphism
module, 699
sage.modules.free_quadratic_module
module, 493
sage.modules.free_quadratic_module_integer_symmetric
module, 514
sage.modules.matrix_morphism
module, 663
sage.modules.misc
module, 769
sage.modules.module
module, 11
sage.modules.multi_filtered_vector_space
module, 596
sage.modules.quotient_module
module, 236
sage.modules.real_double_vector
module, 754
sage.modules.submodule
module, 232
sage.modules.tensor_operations
module, 758
sage.modules.torsion_quadratic_module
module, 555
sage.modules.tutorial_free_modules
module, 3
sage.modules.vector_callable_symbolic_dense
module, 737
sage.modules.vector_complex_double_dense
module, 752
sage.modules.vector_double_dense
module, 738
sage.modules.vector_integer_dense
module, 707
sage.modules.vector_integer_sparse
module, 708
sage.modules.vector_mod2_dense
module, 708
sage.modules.vector_modn_dense
module, 710
sage.modules.vector_modn_sparse
module, 713
sage.modules.vector_numpy_dense
module, 749
sage.modules.vector_numpy_integer_dense
module, 751
sage.modules.vector_rational_dense
module, 713
sage.modules.vector_rational_sparse
module, 714
sage.modules.vector_real_double_dense
module, 747
sage.modules.vector_space_homspace
module, 611
sage.modules.vector_space_morphism
module, 617
sage.modules.vector_symbolic_dense
module, 714
sage.modules.vector_symbolic_sparse
module, 725
sage.modules.with_basis.all
module, 249
sage.modules.with_basis.cell_module
module, 249
sage.modules.with_basis.indexed_element
module, 253
sage.modules.with_basis.invariant
module, 256
sage.modules.with_basis.morphism
module, 271
sage.modules.with_basis.representation
module, 305
sage.modules.with_basis.subquotient
module, 295
saturation() (*sage.modules.free_module.FreeModule_generic_pid method*), 86
scale() (*sage.modules.free_module.FreeModule_generic method*), 62

scale() (*sage.modules.free_module.FreeModule_generic_field method*), 74
schur_functor() (*sage.modules.with_basis.representation.Representation_abstract method*), 323
SchurFunctorRepresentation (*class in sage.modules.with_basis.representation*), 326
section() (*sage.modules.with_basis.morphism.TriangularModuleMorphism method*), 292
semigroup() (*sage.modules.with_basis.invariant.FiniteDimensionalInvariantModule method*), 262
semigroup() (*sage.modules.with_basis.representation.Representation_abstract method*), 323
semigroup_algebra() (*sage.modules.with_basis.representation.Representation_abstract method*), 323
semigroup_representation() (*sage.modules.with_basis.invariant.FiniteDimensionalInvariantModule method*), 263
set() (*sage.modules.free_module_element.FreeModuleElement method*), 192
shift() (*sage.modules.filtered_vector_space.FilteredVectorSpace_class method*), 589
shift() (*sage.modules.multi_filtered_vector_space.MultiFilteredVectorSpace_class method*), 606
short_vectors() (*sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method*), 534
shortest_vector() (*sage.modules.free_module_integer.FreeModule_submodule_with_basis_integer method*), 484
side() (*sage.modules.free_module_pseudomorphism.FreeModulePseudoMorphism method*), 703
side() (*sage.modules.matrix_morphism.MatrixMorphism_abstract method*), 693
side() (*sage.modules.with_basis.representation.Representation_abstract method*), 324
side_switch() (*sage.modules.free_module_pseudomorphism.FreeModulePseudoMorphism method*), 703
side_switch() (*sage.modules.matrix_morphism.MatrixMorphism_abstract method*), 693
signature() (*sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method*), 535
signature_pair() (*sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method*), 535
SignRepresentation_abstract (*class in sage.modules.with_basis.representation*), 331
SignRepresentationCoxeterGroup (*class in sage.modules.with_basis.representation*), 330
SignRepresentationMatrixGroup (*class in sage.modules.with_basis.representation*), 330
SignRepresentationPermgroup (*class in sage.modules.with_basis.representation*), 331
simple_module() (*sage.modules.with_basis.cell_module.CellModule method*), 252
SimpleModule (*class in sage.modules.with_basis.cell_module*), 252
SimpleModule.Element (*class in sage.modules.with_basis.cell_module*), 253
simplify() (*sage.modules.vector_symbolic_dense.Vector_symbolic_dense method*), 716
simplify() (*sage.modules.vector_symbolic_sparse.Vector_symbolic_sparse method*), 727
simplify_factorial() (*sage.modules.vector_symbolic_dense.Vector_symbolic_dense method*), 717
simplify_factorial() (*sage.modules.vector_symbolic_sparse.Vector_symbolic_sparse method*), 728
simplify_full() (*sage.modules.vector_symbolic_dense.Vector_symbolic_dense method*), 718
simplify_full() (*sage.modules.vector_symbolic_sparse.Vector_symbolic_sparse method*), 729
simplify_log() (*sage.modules.vector_symbolic_dense.Vector_symbolic_dense method*), 719
simplify_log() (*sage.modules.vector_symbolic_sparse.Vector_symbolic_sparse method*), 731
simplify_rational() (*sage.modules.vector_symbolic_dense.Vector_symbolic_dense method*), 720
simplify_rational() (*sage.modules.vector_symbolic_sparse.Vector_symbolic_sparse method*), 732
simplify_trig() (*sage.modules.vector_symbolic_dense.Vector_symbolic_dense method*), 722
simplify_trig() (*sage.modules.vector_symbolic_sparse.Vector_symbolic_sparse method*), 733
smith_form_gen() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 359
smith_form_gens() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 360
smith_to_gens() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 360
solve() (*sage.modules.fp_graded.morphism.FPModuleMorphism method*), 451
some_elements() (*sage.modules.free_module.Module_free_ambient method*), 122
span() (*in module sage.modules.free_module*), 133
span() (*sage.modules.free_module.Module_free_ambient method*), 124

span() (*sage.modules.free_quadratic_module.FreeQuadraticModule_generic_field method*), 501
span() (*sage.modules.free_quadratic_module.FreeQuadraticModule_generic_pid method*), 503
span_of_basis() (*sage.modules.free_module.FreeModule_generic_field method*), 75
span_of_basis() (*sage.modules.free_module.FreeModule_generic_pid method*), 87
span_of_basis() (*sage.modules.free_quadratic_module.FreeQuadraticModule_generic_field method*), 502
span_of_basis() (*sage.modules.free_quadratic_module.FreeQuadraticModule_generic_pid method*), 503
sparse_module() (*sage.modules.free_module.FreeModule_generic method*), 63
sparse_vector() (*sage.modules.free_module_element.FreeModuleElement method*), 192
split() (*sage.modules.fp_graded.morphism.FPModuleMorphism method*), 452
standard_deviation() (*sage.modules.vector_double_dense.Vector_double_dense method*), 744
stats_kurtosis() (*sage.modules.vector_double_dense.Vector_double_dense method*), 745
stats_skew() (*sage.modules.vector_real_double_dense.Vector_real_double_dense method*), 748
SteenrodFPModule (*class in sage.modules.fp_graded.steenrod.module*), 464
SteenrodFPModuleMorphism (*class in sage.modules.fp_graded.steenrod.morphism*), 470
SteenrodFreeModule (*class in sage.modules.fp_graded.steenrod.module*), 467
SteenrodFreeModuleMorphism (*class in sage.modules.fp_graded.steenrod.morphism*), 476
SteenrodModuleMixin (*class in sage.modules.fp_graded.steenrod.module*), 468
sublattice() (*sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method*), 535
submodule() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 362
submodule() (*sage.modules.free_module.Module_free_ambient method*), 127
Submodule_free_ambient (*class in sage.modules.submodule*), 233
submodule_inclusion() (*sage.modules.fp_graded.module.FPModule method*), 427
submodule_with_basis() (*sage.modules.free_module.FreeModule_generic_pid method*), 88
submodule_with_gens() (*sage.modules.torsion_quadratic_module.TorsionQuadraticModule method*), 570
SubmoduleWithBasis (*class in sage.modules.with_basis.subquotient*), 297
Subrepresentation (*class in sage.modules.with_basis.representation*), 331
subrepresentation() (*sage.modules.with_basis.representation.Representation_abstract method*), 324
Subrepresentation.Element (*class in sage.modules.with_basis.representation*), 331
subs() (*sage.modules.free_module_element.FreeModuleElement method*), 192
subspace() (*sage.modules.free_module.FreeModule_generic_field method*), 76
subspace() (*sage.modules.with_basis.subquotient.SubmoduleWithBasis method*), 303
subspace_sum() (*sage.modules.with_basis.subquotient.SubmoduleWithBasis method*), 304
subspace_with_basis() (*sage.modules.free_module.FreeModule_generic_field method*), 77
subspaces() (*sage.modules.free_module.FreeModule_generic_field method*), 79
sum() (*sage.modules.vector_double_dense.Vector_double_dense method*), 746
support() (*sage.modules.filtered_vector_space.FilteredVectorSpace_class method*), 590
support() (*sage.modules.free_module_element.FreeModuleElement method*), 193
support() (*sage.modules.multi_filtered_vector_space.MultiFilteredVectorSpace_class method*), 606
suspension() (*sage.modules.fp_graded.free_module.FreeGradedModule method*), 396
suspension() (*sage.modules.fp_graded.module.FPModule method*), 428
suspension() (*sage.modules.fp_graded.morphism.FPModuleMorphism method*), 453
symmetric_power() (*sage.modules.filtered_vector_space.FilteredVectorSpace_class method*), 590
symmetric_power() (*sage.modules.multi_filtered_vector_space.MultiFilteredVectorSpace_class method*), 607
symmetric_power() (*sage.modules.with_basis.representation.Representation_abstract method*), 325
symmetrized_coordinate_sums() (*in module sage.modules.tensor_operations*), 765

T

Tensor (*sage.modules.with_basis.representation.Representation_abstract attribute*), 311
tensor_product() (*sage.modules.filtered_vector_space.FilteredVectorSpace_class method*), 591

tensor_product() (*sage.modules.free_module_element.FreeModuleElement method*), 193
tensor_product() (*sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method*), 536
tensor_product() (*sage.modules.multi_filtered_vector_space.MultiFilteredVectorSpace_class method*), 607
TensorOperation (*class in sage.modules.tensor_operations*), 760
to_vector() (*sage.modules.with_basis.indexed_element.IndexedFreeModuleElement method*), 254
TorsionQuadraticForm() (*in module sage.modules.torsion_quadratic_module*), 555
TorsionQuadraticModule (*class in sage.modules.torsion_quadratic_module*), 556
TorsionQuadraticModuleElement (*class in sage.modules.torsion_quadratic_module*), 573
trace() (*sage.modules.matrix_morphism.MatrixMorphism_abstract method*), 694
TriangularModuleMorphism (*class in sage.modules.with_basis.morphism*), 279
TriangularModuleMorphismByLinearity (*class in sage.modules.with_basis.morphism*), 293
TriangularModuleMorphismFromFunction (*class in sage.modules.with_basis.morphism*), 294
trig_expand() (*sage.modules.vector_symbolic_dense.Vector_symbolic_dense method*), 723
trig_expand() (*sage.modules.vector_symbolic_sparse.Vector_symbolic_sparse method*), 734
trig_reduce() (*sage.modules.vector_symbolic_dense.Vector_symbolic_dense method*), 724
trig_reduce() (*sage.modules.vector_symbolic_sparse.Vector_symbolic_sparse method*), 735
TrivialRepresentation (*class in sage.modules.with_basis.representation*), 332
TrivialRepresentation.Element (*class in sage.modules.with_basis.representation*), 332
twist() (*sage.modules.free_quadratic_module_integer_symmetric.FreeQuadraticModule_integer_symmetric method*), 538
twist() (*sage.modules.torsion_quadratic_module.TorsionQuadraticModule method*), 571
twisted_invariant_module() (*sage.modules.with_basis.representation.Representation_abstract method*), 325
twisting_derivation() (*sage.modules.free_module_pseudomorphism.FreeModulePseudoMorphism method*), 705
twisting_morphism() (*sage.modules.free_module_pseudomorphism.FreeModulePseudoMorphism method*), 705

U

unpickle_v0() (*in module sage.modules.vector_complex_double_dense*), 753
unpickle_v0() (*in module sage.modules.vector_integer_dense*), 708
unpickle_v0() (*in module sage.modules.vector_mod2_dense*), 710
unpickle_v0() (*in module sage.modules.vector_modn_dense*), 712
unpickle_v0() (*in module sage.modules.vector_rational_dense*), 714
unpickle_v0() (*in module sage.modules.vector_real_double_dense*), 749
unpickle_v1() (*in module sage.modules.vector_complex_double_dense*), 753
unpickle_v1() (*in module sage.modules.vector_integer_dense*), 708
unpickle_v1() (*in module sage.modules.vector_modn_dense*), 712
unpickle_v1() (*in module sage.modules.vector_rational_dense*), 714
unpickle_v1() (*in module sage.modules.vector_real_double_dense*), 749
update_reduced_basis() (*sage.modules.free_module_integer.FreeModule_submodule_with_basis_integer method*), 486
user_to_echelon_matrix() (*sage.modules.free_module.FreeModule_submodule_with_basis_pid method*), 115
uses_ambient_inner_product() (*sage.modules.free_module.FreeModule_generic method*), 64

V

V() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 339
V() (*sage.modules.quotient_module.FreeModule_ambient_field_quotient method*), 239
V() (*sage.modules.quotient_module.QuotientModule_free_ambient method*), 243
value_module() (*sage.modules.torsion_quadratic_module.TorsionQuadraticModule method*), 572
value_module_qf() (*sage.modules.torsion_quadratic_module.TorsionQuadraticModule method*), 572
values() (*sage.modules.fp_graded.morphism.FPModuleMorphism method*), 455
variance() (*sage.modules.vector_double_dense.Vector_double_dense method*), 746

vector() (*in module sage.modules.free_module_element*), 220
vector() (*sage.modules.fg_pid.fgp_element.FGP_Element method*), 369
Vector_callable_symbolic_dense (*class in sage.modules.vector_callable_symbolic_dense*), 737
Vector_complex_double_dense (*class in sage.modules.vector_complex_double_dense*), 752
Vector_double_dense (*class in sage.modules.vector_double_dense*), 738
Vector_integer_dense (*class in sage.modules.vector_integer_dense*), 708
Vector_mod2_dense (*class in sage.modules.vector_mod2_dense*), 709
Vector_modn_dense (*class in sage.modules.vector_modn_dense*), 712
Vector_numpy_dense (*class in sage.modules.vector_numpy_dense*), 750
Vector_numpy_integer_dense (*class in sage.modules.vector_numpy_integer_dense*), 751
vector_presentation() (*sage.modules.fp_graded.element.FPElement method*), 433
vector_presentation() (*sage.modules.fp_graded.free_element.FreeGradedModuleElement method*), 400
vector_presentation() (*sage.modules.fp_graded.free_module.FreeGradedModule method*), 396
vector_presentation() (*sage.modules.fp_graded.module.FPModule method*), 429
vector_presentation() (*sage.modules.fp_graded.morphism.FPModuleMorphism method*), 456
Vector_rational_dense (*class in sage.modules.vector_rational_dense*), 714
Vector_real_double_dense (*class in sage.modules.vector_real_double_dense*), 748
vector_space() (*sage.modules.free_module.FreeModule_ambient_domain method*), 35
vector_space() (*sage.modules.free_module.FreeModule_generic_field method*), 80
vector_space() (*sage.modules.free_module.FreeModule_submodule_with_basis_pid method*), 115
vector_space_span() (*sage.modules.free_module.FreeModule_generic_pid method*), 90
vector_space_span_of_basis() (*sage.modules.free_module.FreeModule_generic_pid method*), 92
Vector_symbolic_dense (*class in sage.modules.vector_symbolic_dense*), 715
Vector_symbolic_sparse (*class in sage.modules.vector_symbolic_sparse*), 726
VectorCollection (*class in sage.modules.tensor_operations*), 764
vectors() (*sage.modules.tensor_operations.VectorCollection method*), 764
VectorSpace() (*in module sage.modules.free_module*), 130
VectorSpaceHomspace (*class in sage.modules.vector_space_homspace*), 616
VectorSpaceMorphism (*class in sage.modules.vector_space_morphism*), 627
volume() (*sage.modules.free_module_integer.FreeModule_submodule_with_basis_integer method*), 486
voronoi_cell() (*sage.modules.free_module_integer.FreeModule_submodule_with_basis_integer method*), 486
voronoi_relevant_vectors() (*sage.modules.free_module_integer.FreeModule_submodule_with_basis_integer method*), 488

W

W() (*sage.modules.fg_pid.fgp_module.FGP_Module_class method*), 340
W() (*sage.modules.quotient_module.FreeModule_ambient_field_quotient method*), 239
W() (*sage.modules.quotient_module.QuotientModule_free_ambient method*), 243
wedge() (*sage.modules.filtered_vector_space.FilteredVectorSpace_class method*), 592
wedge() (*sage.modules.multi_filtered_vector_space.MultiFilteredVectorSpace_class method*), 608

Z

zero() (*sage.modules.fp_graded.homspace.FPModuleHomspace method*), 463
zero() (*sage.modules.fp_graded.module.FPModule method*), 429
zero() (*sage.modules.free_module_homspace.FreeModuleHomspace method*), 641
zero() (*sage.modules.free_module.Module_free_ambient method*), 129
zero_at() (*sage.modules.vector_double_dense.Vector_double_dense method*), 746
zero_submodule() (*sage.modules.free_module.FreeModule_generic_field method*), 80
zero_submodule() (*sage.modules.free_module.FreeModule_generic_pid method*), 92
zero_submodule() (*sage.modules.free_module.Module_free_ambient method*), 129
zero_submodule() (*sage.modules.free_quadratic_module.FreeQuadraticModule_generic_pid method*), 504
zero_subspace() (*sage.modules.free_module.FreeModule_generic_field method*), 81

zero_vector() (*in module sage.modules.free_module_element*), 230
zero_vector() (*sage.modules.free_module.Module_free_ambient method*), 130