

---

# **Knot Theory**

***Release 10.6***

**The Sage Development Team**

**Jun 27, 2025**



# CONTENTS

|          |                                                               |            |
|----------|---------------------------------------------------------------|------------|
| <b>1</b> | <b>Knots</b>                                                  | <b>1</b>   |
| <b>2</b> | <b>Links</b>                                                  | <b>11</b>  |
| <b>3</b> | <b>Access to the KnotInfo database</b>                        | <b>77</b>  |
| <b>4</b> | <b>Free monoid generated by prime knots available via the</b> | <b>125</b> |
| <b>5</b> | <b>Indices and Tables</b>                                     | <b>129</b> |
|          | <b>Python Module Index</b>                                    | <b>131</b> |
|          | <b>Index</b>                                                  | <b>133</b> |



## KNOTS

### AUTHORS:

- Miguel Angel Marco Buzunariz
- Amit Jamadagni

**class** sage.knots.knot.**Knot** (data, check=True)

Bases: [Link](#), [Element](#)

A knot.

A knot is defined as embedding of the circle  $\mathbb{S}^1$  in the 3-dimensional sphere  $\mathbb{S}^3$ , considered up to ambient isotopy. They represent the physical idea of a knotted rope, but with the particularity that the rope is closed. That is, the ends of the rope are joined.

#### See also

[Link](#)

### INPUT:

- data – see [Link](#) for the allowable inputs
- check – boolean (default: True); if True, make sure that the data define a knot, not a link

### EXAMPLES:

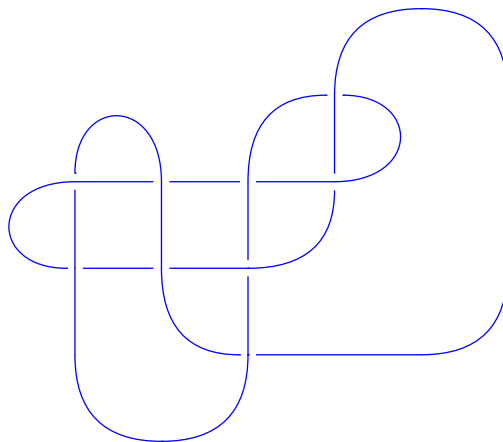
We construct the knot  $8_{14}$  and compute some invariants:

```
sage: B = BraidGroup(4)
sage: K = Knot(B([1, 1, 1, 2, -1, 2, -3, 2, -3]))
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> K = Knot(B([Integer(1), Integer(1), Integer(1), Integer(2), -Integer(1),
↪ Integer(2), -Integer(3), Integer(2), -Integer(3)]))
```

```
sage: K.alexander_polynomial()
-2*t^-2 + 8*t^-1 - 11 + 8*t - 2*t^2
sage: K.jones_polynomial()
t^7 - 3*t^6 + 4*t^5 - 5*t^4 + 6*t^3 - 5*t^2 + 4*t + 1/t - 2
sage: K.determinant()
31
```

(continues on next page)



(continued from previous page)

```
sage: K.signature()
-2
sage: K.colored_jones_polynomial(2) # long time
q^-1 - 2 + 4*q - 5*q^2 + 6*q^3 - 5*q^4 + 4*q^5 - 3*q^6 + q^7
```

```
>>> from sage.all import *
>>> K.alexander_polynomial()
-2*t^-2 + 8*t^-1 - 11 + 8*t - 2*t^2
>>> K.jones_polynomial()
t^7 - 3*t^6 + 4*t^5 - 5*t^4 + 6*t^3 - 5*t^2 + 4*t + 1/t - 2
>>> K.determinant()
31
>>> K.signature()
-2
>>> K.colored_jones_polynomial(Integer(2)) # long time
q^-1 - 2 + 4*q - 5*q^2 + 6*q^3 - 5*q^4 + 4*q^5 - 3*q^6 + q^7
```

## REFERENCES:

- [Wikipedia article Knot\\_\(mathematics\)](#)

**arf\_invariant()**

Return the Arf invariant.

## EXAMPLES:

```
sage: B = BraidGroup(4)
sage: K = Knot(B([-1, 2, 1, 2]))
sage: K.arf_invariant()
0
sage: B = BraidGroup(8)
sage: K = Knot(B([-2, 3, 1, 2, 1, 4]))
sage: K.arf_invariant()
0
sage: K = Knot(B([1, 2, 1, 2]))
sage: K.arf_invariant()
1
```

```

>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> K = Knot(B([-Integer(1), Integer(2), Integer(1), Integer(2)]))
>>> K.arf_invariant()
0
>>> B = BraidGroup(Integer(8))
>>> K = Knot(B([-Integer(2), Integer(3), Integer(1), Integer(2), Integer(1),
↳ Integer(4)]))
>>> K.arf_invariant()
0
>>> K = Knot(B([Integer(1), Integer(2), Integer(1), Integer(2)]))
>>> K.arf_invariant()
1

```

**colored\_jones\_polynomial** ( $N$ , *variab*=None, *try\_inverse*=True)

Return the colored Jones polynomial of the trace closure of the braid.

INPUT:

- $N$  – integer; the number of colors
- *variab* – (default:  $q$ ) the variable in the resulting Laurent polynomial
- *try\_inverse* – boolean (default: True); if True, attempt a faster calculation by using the inverse of the braid

ALGORITHM:

The algorithm used is described in [HL2018] for the corresponding braid representation. We follow their notation, but work in a suitable free algebra over a Laurent polynomial ring in one variable to simplify book-keeping.

EXAMPLES:

```

sage: W = Knots()
sage: K = W.from_dowker_code([-4,-6,-2])
sage: K.colored_jones_polynomial(2)
-q^-4 + q^-3 + q^-1
sage: K.colored_jones_polynomial(2, 't')
-t^-4 + t^-3 + t^-1
sage: R.<t> = LaurentPolynomialRing(ZZ)
sage: K.colored_jones_polynomial(2, -t)
-t^-4 - t^-3 - t^-1

sage: R.<t> = ZZ[]
sage: K.colored_jones_polynomial(2, t+1)
(t^3 + 3*t^2 + 4*t + 1)/(t^4 + 4*t^3 + 6*t^2 + 4*t + 1)

```

```

>>> from sage.all import *
>>> W = Knots()
>>> K = W.from_dowker_code([-Integer(4), -Integer(6), -Integer(2)])
>>> K.colored_jones_polynomial(Integer(2))
-q^-4 + q^-3 + q^-1
>>> K.colored_jones_polynomial(Integer(2), 't')
-t^-4 + t^-3 + t^-1
>>> R = LaurentPolynomialRing(ZZ, names=('t',)); (t,) = R._first_ngens(1)

```

(continues on next page)

(continued from previous page)

```

>>> K.colored_jones_polynomial(Integer(2), -t)
-t^4 - t^3 - t^1

>>> R = ZZ['t']; (t,) = R._first_ngens(1)
>>> K.colored_jones_polynomial(Integer(2), t+Integer(1))
(t^3 + 3*t^2 + 4*t + 1)/(t^4 + 4*t^3 + 6*t^2 + 4*t + 1)

```

**connected\_sum(*other*)**Return the oriented connected sum of *self* and *other*.**Note**

We give the knots an orientation based upon the braid representation.

**INPUT:**

- *other* – a knot

**OUTPUT:** a knot equivalent to the connected sum of *self* and *other***EXAMPLES:**

```

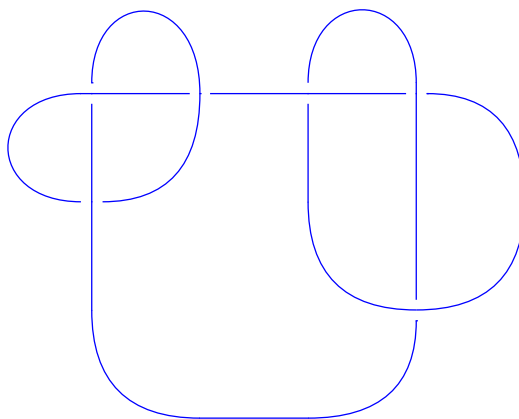
sage: B = BraidGroup(2)
sage: trefoil = Knot(B([1,1,1]))
sage: K = trefoil.connected_sum(trefoil); K
Knot represented by 6 crossings
sage: K.braid()
s0^3*s1^-1*s0^3*s1

```

```

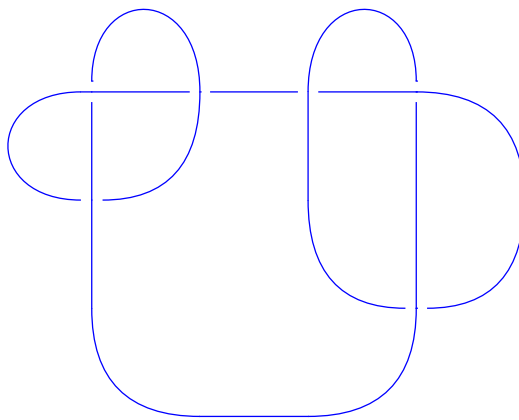
>>> from sage.all import *
>>> B = BraidGroup(Integer(2))
>>> trefoil = Knot(B([Integer(1),Integer(1),Integer(1)]))
>>> K = trefoil.connected_sum(trefoil); K
Knot represented by 6 crossings
>>> K.braid()
s0^3*s1^-1*s0^3*s1

```



```
sage: rev_trefoil = Knot(B([-1,-1,-1]))
sage: K2 = trefoil.connected_sum(rev_trefoil); K2
Knot represented by 6 crossings
sage: K2.braid()
s0^3*s1^-1*s0^-3*s1
```

```
>>> from sage.all import *
>>> rev_trefoil = Knot(B([-Integer(1),-Integer(1),-Integer(1)]))
>>> K2 = trefoil.connected_sum(rev_trefoil); K2
Knot represented by 6 crossings
>>> K2.braid()
s0^3*s1^-1*s0^-3*s1
```



Observe that both knots have according `dowker_notation` (showing that the constructing from DT-code may not be unique for non prime knots, see `from_dowker_code()`):

```
sage: K.dowker_notation()
[(4, 1), (2, 5), (6, 3), (10, 7), (8, 11), (12, 9)]
sage: K2.dowker_notation()
[(4, 1), (2, 5), (6, 3), (7, 10), (11, 8), (9, 12)]
sage: K.homfly_polynomial() == K2.homfly_polynomial()
False
```

```
>>> from sage.all import *
>>> K.dowker_notation()
[(4, 1), (2, 5), (6, 3), (10, 7), (8, 11), (12, 9)]
>>> K2.dowker_notation()
[(4, 1), (2, 5), (6, 3), (7, 10), (11, 8), (9, 12)]
>>> K.homfly_polynomial() == K2.homfly_polynomial()
False
```

#### REFERENCES:

- [Wikipedia article Connected\\_sum](#)

#### `dt_code()`

Return the DT code of `self`.

#### ALGORITHM:

The DT code is generated by the following way:

Start moving along the knot, as we encounter the crossings we start numbering them, so every crossing has two numbers assigned to it once we have traced the entire knot. Now we take the even number associated with every crossing.

The following sign convention is to be followed:

Take the even number with a negative sign if it is an overcrossing that we are encountering.

OUTPUT: DT code representation of the knot

EXAMPLES:

```
sage: K = Knot([[1, 5, 2, 4], [5, 3, 6, 2], [3, 1, 4, 6]])
sage: K.dt_code()
[4, 6, 2]

sage: B = BraidGroup(4)
sage: K = Knot(B([1, 2, 1, 2]))
sage: K.dt_code()
[4, -6, 8, -2]

sage: K = Knot([[1, -2, 3, -4, 5, -1, 2, -3, 4, -5]],
....:          [1, 1, 1, 1, 1])
sage: K.dt_code()
[6, 8, 10, 2, 4]
```

```
>>> from sage.all import *
>>> K = Knot([[Integer(1), Integer(5), Integer(2), Integer(4)], [Integer(5),
↪ Integer(3), Integer(6), Integer(2)], [Integer(3), Integer(1), Integer(4),
↪ Integer(6)]])
>>> K.dt_code()
[4, 6, 2]

>>> B = BraidGroup(Integer(4))
>>> K = Knot(B([Integer(1), Integer(2), Integer(1), Integer(2)]))
>>> K.dt_code()
[4, -6, 8, -2]

>>> K = Knot([[[Integer(1), -Integer(2), Integer(3), -Integer(4), Integer(5),
↪ -Integer(1), Integer(2), -Integer(3), Integer(4), -Integer(5)]],
...           [Integer(1), Integer(1), Integer(1), Integer(1), Integer(1)]])
>>> K.dt_code()
[6, 8, 10, 2, 4]
```

```
class sage.knots.knot.Knots
```

Bases: `Singleton`, `Parent`

The set for all knots, as a monoid for the connected sum.

**Element**

alias of `Knot`

**an\_element()**

Return the trefoil knot.

EXAMPLES:

```
sage: Knots().an_element()
Knot represented by 3 crossings
```

```
>>> from sage.all import *
>>> Knots().an_element()
Knot represented by 3 crossings
```

**from\_dowker\_code**(code)

Build a knot from a Dowker-Thistlethwaite code.

The Dowker-Thistlethwaite code of a knot diagram is defined as follows.

Start following the knot diagram at some regular point. Label the crossings by a number (starting from number 1) in the order in which they are met. At the end, every crossing gets numbered twice, once by an even number and once by an odd number. When meeting an over-crossing with even number, use instead the negative of this even number as label.

Then the set of crossings gives a set of pairs (odd, even). Sort this set according to the odd component, and then keep only the even components in the same order. This is the Dowker-Thistlethwaite code.

INPUT:

- code – list of signed even numbers; the Dowker-Thistlethwaite code of a knot

OUTPUT: a knot

### Warning

In general the Dowker-Thistlethwaite code does not describe a knot uniquely. It is not only insensitive on mirror images, but may also mix up non prime knots. For example `[4, 6, 2, 10, 12, 8]` describes the connected sum of two trefoil knots, as well as the connected sum of a trefoil with its mirror (see the corresponding example in the documentation of `connected_sum()`).

EXAMPLES:

```
sage: W = Knots()
sage: K1 = W.from_dowker_code([8,10,2,12,4,6])
sage: K1.dowker_notation()
[(5, 2), (9, 4), (11, 6), (1, 8), (3, 10), (7, 12)]

sage: W.from_dowker_code([6,10,14,12,16,2,18,4,8])
Knot represented by 9 crossings

sage: W.from_dowker_code([4,8,10,-14,2,-16,-18,-6,-12])
Knot represented by 9 crossings

sage: K3 = W.from_dowker_code([6,-12,2,8,-4,-10]); K3
Knot represented by 6 crossings
sage: K3.dowker_notation()
[(5, 2), (4, 9), (1, 6), (7, 8), (10, 11), (12, 3)]
```

```
>>> from sage.all import *
>>> W = Knots()
>>> K1 = W.from_dowker_code([Integer(8),Integer(10),Integer(2),Integer(12),
```

(continues on next page)

(continued from previous page)

```

↪Integer(4),Integer(6)])
>>> K1.dowker_notation()
[(5, 2), (9, 4), (11, 6), (1, 8), (3, 10), (7, 12)]

>>> W.from_dowker_code([Integer(6),Integer(10),Integer(14),Integer(12),
↪Integer(16),Integer(2),Integer(18),Integer(4),Integer(8)])
Knot represented by 9 crossings

>>> W.from_dowker_code([Integer(4),Integer(8),Integer(10),-Integer(14),
↪Integer(2),-Integer(16),-Integer(18),-Integer(6),-Integer(12)])
Knot represented by 9 crossings

>>> K3 = W.from_dowker_code([Integer(6),-Integer(12),Integer(2),Integer(8),-
↪Integer(4),-Integer(10)]); K3
Knot represented by 6 crossings
>>> K3.dowker_notation()
[(5, 2), (4, 9), (1, 6), (7, 8), (10, 11), (12, 3)]

```

**See also**

dowker\_notation()

**REFERENCES:**

- [Wikipedia article Dowker\\_notation](#)
- [http://katlas.org/wiki/DT\\_\(Dowker-Thistlethwaite\)\\_Codes](http://katlas.org/wiki/DT_(Dowker-Thistlethwaite)_Codes)

**from\_gauss\_code** (*gauss*)

Build a knot from a signed Gauss code.

This makes some arbitrary choice of orientation.

INPUT:

- *gauss* – a signed Gauss code

OUTPUT: a knot

EXAMPLES:

```

sage: W = Knots()
sage: K1 = W.from_gauss_code([2, -1, 3, -2, 1, -3])
sage: K1.alexander_polynomial()
t^-1 - 1 + t

```

```

>>> from sage.all import *
>>> W = Knots()
>>> K1 = W.from_gauss_code([Integer(2), -Integer(1), Integer(3), -Integer(2),
↪Integer(1), -Integer(3)])
>>> K1.alexander_polynomial()
t^-1 - 1 + t

```

**from\_table** (*n, k*)

Return a knot from its index in the Rolfsen table.

INPUT:

- $n$  – the crossing number
- $k$  – positive integer

OUTPUT: the knot  $K_{n,k}$  in the Rolfsen table

EXAMPLES:

```
sage: K1 = Knots().from_table(6,3); K1
Knot represented by 6 crossings
sage: K1.alexander_polynomial()
t^-2 - 3*t^-1 + 5 - 3*t + t^2

sage: K2 = Knots().from_table(8,4); K2
Knot represented by 9 crossings
sage: K2.determinant()
19
sage: K2.signature()
2

sage: K3 = Knots().from_table(10,56); K3
Knot represented by 11 crossings
sage: K3.jones_polynomial()
t^10 - 3*t^9 + 6*t^8 - 9*t^7 + 10*t^6 - 11*t^5 + 10*t^4 - 7*t^3
+ 5*t^2 - 2*t + 1

sage: K4 = Knots().from_table(10,100)
sage: K4.genus()
4
```

```
>>> from sage.all import *
>>> K1 = Knots().from_table(Integer(6),Integer(3)); K1
Knot represented by 6 crossings
>>> K1.alexander_polynomial()
t^-2 - 3*t^-1 + 5 - 3*t + t^2

>>> K2 = Knots().from_table(Integer(8),Integer(4)); K2
Knot represented by 9 crossings
>>> K2.determinant()
19
>>> K2.signature()
2

>>> K3 = Knots().from_table(Integer(10),Integer(56)); K3
Knot represented by 11 crossings
>>> K3.jones_polynomial()
t^10 - 3*t^9 + 6*t^8 - 9*t^7 + 10*t^6 - 11*t^5 + 10*t^4 - 7*t^3
+ 5*t^2 - 2*t + 1

>>> K4 = Knots().from_table(Integer(10),Integer(100))
>>> K4.genus()
4
```

REFERENCES:

- KnotAtlas, [http://katlas.math.toronto.edu/wiki/The\\_Rolfsen\\_Knot\\_Table](http://katlas.math.toronto.edu/wiki/The_Rolfsen_Knot_Table)

**one()**

Return the unit of the monoid.

This is the trivial knot.

EXAMPLES:

```
sage: Knots().one()  
Knot represented by 0 crossings
```

```
>>> from sage.all import *  
>>> Knots().one()  
Knot represented by 0 crossings
```

## LINKS

A knot is defined as embedding of the circle  $\mathbb{S}^1$  in the 3-dimensional sphere  $\mathbb{S}^3$ , considered up to ambient isotopy. They represent the physical idea of a knotted rope, but with the particularity that the rope is closed. That is, the ends of the rope are joined.

A link is an embedding of one or more copies of  $\mathbb{S}^1$  in  $\mathbb{S}^3$ , considered up to ambient isotopy. That is, a link represents the idea of one or more tied ropes. Every knot is a link, but not every link is a knot.

Generically, the projection of a link on  $\mathbf{R}^2$  is a curve with crossings. The crossings are represented to show which strand goes over the other. This curve is called a planar diagram of the link. If we remove the crossings, the resulting connected components are segments. These segments are called the edges of the diagram.

### REFERENCES:

- [Wikipedia article Knot\\_\(mathematics\)](#)
- [Col2013]
- [KnotAtlas]

### See also

There are also tables of link and knot invariants at web-pages [KnotInfo](#) and [LinkInfo](#). These can be used inside Sage after installing the optional package `database_knotinfo` (type `sage -i database_knotinfo` in a command shell, see [knotinfo](#)).

### AUTHORS:

- Miguel Angel Marco Buzunariz
- Amit Jamadagni
- Sebastian Oehms (October 2020, add `get_knotinfo()` and `is_isotopic()`)
- Sebastian Oehms (May 2022): add `links_gould_polynomial()`
- Sebastian Oehms (May 2023): change the convention about the `pd_code` from clockwise to anti-clockwise (see [Issue #35665](#)).

**class** `sage.knots.link.Link` (*data*)

Bases: `SageObject`

A link.

A link is an embedding of one or more copies of  $\mathbb{S}^1$  in  $\mathbb{S}^3$ , considered up to ambient isotopy. That is, a link represents the idea of one or more tied ropes. Every knot is a link, but not every link is a knot.

A link can be created by using one of the conventions mentioned below:

Braid:

- The closure of a braid is a link:

```
sage: B = BraidGroup(8)
sage: L = Link(B([-1, -1, -1, -2, 1, -2, 3, -2, 3])); L
Link with 1 component represented by 9 crossings
sage: L = Link(B([1, 2, 1, -2, -1])); L
Link with 2 components represented by 5 crossings
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(8))
>>> L = Link(B([-Integer(1), -Integer(1), -Integer(1), -Integer(2), ↵
↵Integer(1), -Integer(2), Integer(3), -Integer(2), Integer(3)])); L
Link with 1 component represented by 9 crossings
>>> L = Link(B([Integer(1), Integer(2), Integer(1), -Integer(2), -
↵Integer(1)])); L
Link with 2 components represented by 5 crossings
```

#### Note

The strands of the braid that have no crossings at all are removed.

- Oriented Gauss Code:

Label the crossings from 1 to  $n$  (where  $n$  is the number of crossings) and start moving along the link. Trace every component of the link, by starting at a particular point on one component of the link and writing down each of the crossings that you encounter until returning to the starting point. The crossings are written with sign depending on whether we cross them as over or undercrossing. Each component is then represented as a list whose elements are the crossing numbers. A second list of  $+1$  and  $-1$ 's keeps track of the orientation of each crossing:

```
sage: L = Link([[-1, 2, 3, -4, 5, -6, 7, 8, -2, -5, 6, 1, -8, -3, 4, -7]],
....:          [-1, -1, -1, -1, 1, 1, -1, 1]])
sage: L
Link with 1 component represented by 8 crossings
```

```
>>> from sage.all import *
>>> L = Link([[-Integer(1), Integer(2), Integer(3), -Integer(4), Integer(5), ↵
↵Integer(6), Integer(7), Integer(8), -Integer(2), -Integer(5), Integer(6), ↵
↵Integer(1), -Integer(8), -Integer(3), Integer(4), -Integer(7)]],
...         [-Integer(1), -Integer(1), -Integer(1), -Integer(1), Integer(1),
↵Integer(1), -Integer(1), Integer(1)])
>>> L
Link with 1 component represented by 8 crossings
```

For links there may be more than one component and the input is as follows:

```
sage: L = Link([[-1, 2], [-3, 4], [1, 3, -4, -2]], [-1, -1, 1, 1])
sage: L
Link with 3 components represented by 4 crossings
```

```
>>> from sage.all import *
>>> L = Link([[[-Integer(1), Integer(2)], [-Integer(3), Integer(4)],
↳ [Integer(1), Integer(3), -Integer(4), -Integer(2)], [-Integer(1), -
↳ Integer(1), Integer(1), Integer(1)]]])
>>> L
Link with 3 components represented by 4 crossings
```

- Planar Diagram (PD) Code:

The diagram of the link is formed by segments that are adjacent to the crossings. Label each one of this segments with a positive number, and for each crossing, write down the four incident segments. The order of these segments is anti-clockwise, starting with the incoming undercrossing.

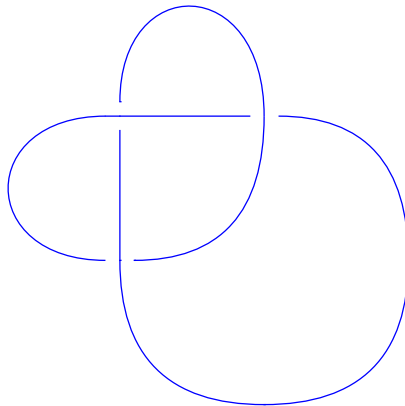
There is no particular distinction between knots and links for this input.

#### EXAMPLES:

One of the representations of the trefoil knot:

```
sage: L = Link([[1, 5, 2, 4], [5, 3, 6, 2], [3, 1, 4, 6]])
sage: L
Link with 1 component represented by 3 crossings
```

```
>>> from sage.all import *
>>> L = Link([[Integer(1), Integer(5), Integer(2), Integer(4)], [Integer(5),
↳ Integer(3), Integer(6), Integer(2)], [Integer(3), Integer(1), Integer(4),
↳ Integer(6)]]])
>>> L
Link with 1 component represented by 3 crossings
```



One of the representations of the Hopf link:

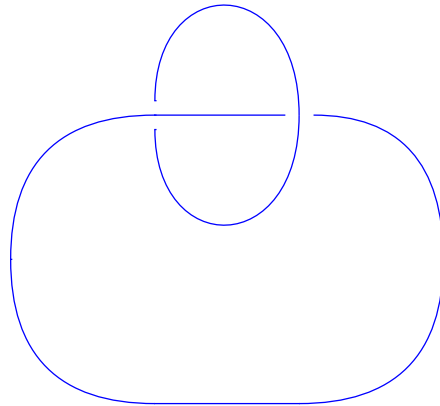
```
sage: L = Link([[1, 4, 2, 3], [4, 1, 3, 2]])
sage: L
Link with 2 components represented by 2 crossings
```

```
>>> from sage.all import *
>>> L = Link([[Integer(1), Integer(4), Integer(2), Integer(3)], [Integer(4),
↳ Integer(1), Integer(3), Integer(2)]]])
```

(continues on next page)

(continued from previous page)

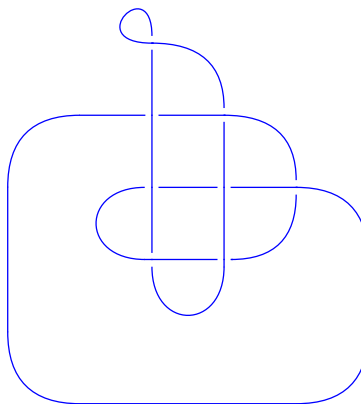
```
>>> L
Link with 2 components represented by 2 crossings
```



We can construct links from the braid group:

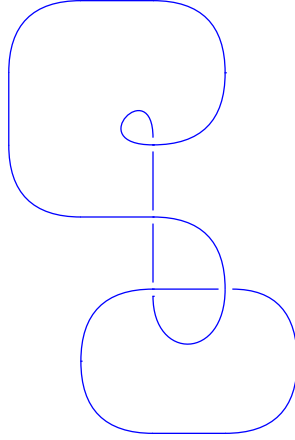
```
sage: B = BraidGroup(4)
sage: L = Link(B([-1, -1, -1, -2, 1, -2, 3, -2])); L
Link with 2 components represented by 8 crossings
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> L = Link(B([-Integer(1), -Integer(1), -Integer(1), -Integer(2), Integer(1), -
Integer(2), Integer(3), -Integer(2)])); L
Link with 2 components represented by 8 crossings
```



```
sage: L = Link(B([1, 2, 1, 3])); L
Link with 2 components represented by 4 crossings
```

```
>>> from sage.all import *
>>> L = Link(B([Integer(1), Integer(2), Integer(1), Integer(3)])); L
Link with 2 components represented by 4 crossings
```



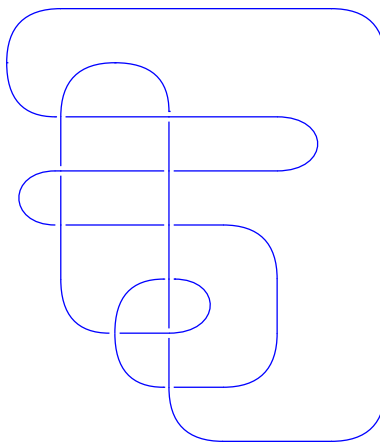
We construct the “monster” unknot using a planar code, and then construct the oriented Gauss code and braid representation:

```
sage: L = Link([[3,4,2,1], [8,7,1,9], [5,3,7,6], [4,5,6,18],
.....:         [17,18,8,19], [9,14,11,10], [10,11,13,12],
.....:         [12,13,15,19], [20,15,14,16], [16,2,17,20]])
sage: L.oriented_gauss_code()
[[[1, -4, 3, -1, 10, -9, 6, -7, 8, 5, 4, -3, 2, -6, 7, -8, 9, -10, -5, -2]],
 [1, -1, 1, 1, 1, -1, -1, -1, -1, -1]]
sage: L.braid()
s0*s1^-3*s2^-1*s1*s3*s2^2*s1^-1*s0^-1*s2*s1^-1*s3^-1*s2*s1^-1
```

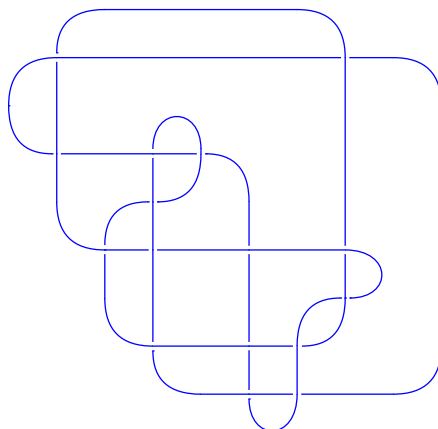
```
>>> from sage.all import *
>>> L = Link([Integer(3),Integer(4),Integer(2),Integer(1)], [Integer(8),
↪Integer(7),Integer(1),Integer(9)], [Integer(5),Integer(3),Integer(7),
↪Integer(6)], [Integer(4),Integer(5),Integer(6),Integer(18)],
...      [Integer(17),Integer(18),Integer(8),Integer(19)], [Integer(9),
↪Integer(14),Integer(11),Integer(10)], [Integer(10),Integer(11),Integer(13),
↪Integer(12)],
...      [Integer(12),Integer(13),Integer(15),Integer(19)], [Integer(20),
↪Integer(15),Integer(14),Integer(16)], [Integer(16),Integer(2),Integer(17),
↪Integer(20)])
>>> L.oriented_gauss_code()
[[[1, -4, 3, -1, 10, -9, 6, -7, 8, 5, 4, -3, 2, -6, 7, -8, 9, -10, -5, -2]],
 [1, -1, 1, 1, 1, -1, -1, -1, -1, -1]]
>>> L.braid()
s0*s1^-3*s2^-1*s1*s3*s2^2*s1^-1*s0^-1*s2*s1^-1*s3^-1*s2*s1^-1
```

We construct the Ochiai unknot by using an oriented Gauss code:

```
sage: L = Link([[1,-2,-3,-8,-12,13,-14,15,-7,-1,2,-4,10,11,-13,12,
.....:         -11,-16,4,3,-5,6,-9,7,-15,14,16,-10,8,9,-6,5]],
.....:         [-1,-1,1,1,1,1,-1,1,1,-1,1,-1,-1,-1,-1,-1,-1,-1,-1,-1]])
sage: L.pd_code()
[[10, 1, 11, 2], [2, 11, 3, 12], [3, 21, 4, 20], [12, 20, 13, 19],
 [21, 1, 22, 32], [31, 23, 32, 22], [9, 24, 10, 25], [4, 30, 5, 29],
 [23, 31, 24, 30], [28, 13, 29, 14], [17, 15, 18, 14], [5, 16, 6, 17],
 [15, 6, 16, 7], [7, 26, 8, 27], [25, 8, 26, 9], [18, 27, 19, 28]]
```



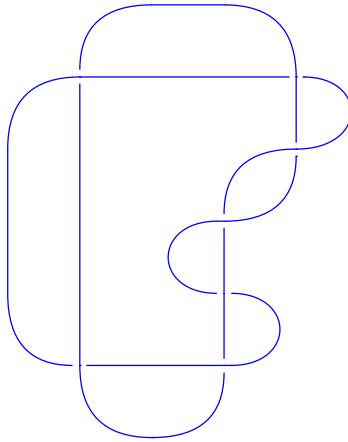
```
>>> from sage.all import *
>>> L = Link([[Integer(1), -Integer(2), -Integer(3), -Integer(8), -Integer(12),
↪ Integer(13), -Integer(14), Integer(15), -Integer(7), -Integer(1), Integer(2), -
↪ Integer(4), Integer(10), Integer(11), -Integer(13), Integer(12),
...      -Integer(11), -Integer(16), Integer(4), Integer(3), -Integer(5),
↪ Integer(6), -Integer(9), Integer(7), -Integer(15), Integer(14), Integer(16), -
↪ Integer(10), Integer(8), Integer(9), -Integer(6), Integer(5)]],
...      [-Integer(1), -Integer(1), Integer(1), Integer(1), Integer(1),
↪ Integer(1), -Integer(1), Integer(1), Integer(1), -Integer(1), Integer(1), -Integer(1),
↪ -Integer(1), -Integer(1), -Integer(1), -Integer(1)])
>>> L.pd_code()
[[10, 1, 11, 2], [2, 11, 3, 12], [3, 21, 4, 20], [12, 20, 13, 19],
[21, 1, 22, 32], [31, 23, 32, 22], [9, 24, 10, 25], [4, 30, 5, 29],
[23, 31, 24, 30], [28, 13, 29, 14], [17, 15, 18, 14], [5, 16, 6, 17],
[15, 6, 16, 7], [7, 26, 8, 27], [25, 8, 26, 9], [18, 27, 19, 28]]
```



We construct the knot  $7_1$  and compute some invariants:

```
sage: B = BraidGroup(2)
sage: L = Link(B([1]*7))
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(2))
>>> L = Link(B([Integer(1)]*Integer(7)))
```



```
sage: L.alexander_polynomial()
t^-3 - t^-2 + t^-1 - 1 + t - t^2 + t^3
sage: L.jones_polynomial()
-t^10 + t^9 - t^8 + t^7 - t^6 + t^5 + t^3
sage: L.determinant()
7
sage: L.signature()
-6
```

```
>>> from sage.all import *
>>> L.alexander_polynomial()
t^-3 - t^-2 + t^-1 - 1 + t - t^2 + t^3
>>> L.jones_polynomial()
-t^10 + t^9 - t^8 + t^7 - t^6 + t^5 + t^3
>>> L.determinant()
7
>>> L.signature()
-6
```

The links here have removed components in which no strand is used:

```
sage: B = BraidGroup(8)
sage: b = B([1])
sage: L = Link(b)
sage: b.components_in_closure()
7
sage: L.number_of_components()
1
sage: L.braid().components_in_closure()
1
sage: L.braid().parent()
Braid group on 2 strands
```

```

>>> from sage.all import *
>>> B = BraidGroup(Integer(8))
>>> b = B([Integer(1)])
>>> L = Link(b)
>>> b.components_in_closure()
7
>>> L.number_of_components()
1
>>> L.braid().components_in_closure()
1
>>> L.braid().parent()
Braid group on 2 strands

```

**Warning**

Equality of knots is done by comparing the corresponding braids, which may give false negatives.

**Note**

The behavior of removing unused strands from an element of a braid group may change without notice in the future. Do not rely on this feature.

**Todo**

Implement methods to creating new links from previously created links.

**alexander\_polynomial** (*var='t'*)

Return the Alexander polynomial of *self*.

INPUT:

- *var* – (default: 't') the variable in the polynomial

EXAMPLES:

We begin by computing the Alexander polynomial for the figure-eight knot:

```

sage: # needs sage.modules
sage: B = BraidGroup(3)
sage: L = Link(B([1, -2, 1, -2]))
sage: L.alexander_polynomial()
-t^-1 + 3 - t

```

```

>>> from sage.all import *
>>> # needs sage.modules
>>> B = BraidGroup(Integer(3))
>>> L = Link(B([Integer(1), -Integer(2), Integer(1), -Integer(2)]))
>>> L.alexander_polynomial()
-t^-1 + 3 - t

```

The “monster” unknot:

```

sage: L = Link([[3,1,2,4],[8,9,1,7],[5,6,7,3],[4,18,6,5],
.....:         [17,19,8,18],[9,10,11,14],[10,12,13,11],
.....:         [12,19,15,13],[20,16,14,15],[16,20,17,2]])
sage: L.alexander_polynomial()
↳needs sage.modules
1

```

```

>>> from sage.all import *
>>> L = Link([[Integer(3),Integer(1),Integer(2),Integer(4)],[Integer(8),
↳Integer(9),Integer(1),Integer(7)],[Integer(5),Integer(6),Integer(7),
↳Integer(3)],[Integer(4),Integer(18),Integer(6),Integer(5)],
...          [Integer(17),Integer(19),Integer(8),Integer(18)],[Integer(9),
↳Integer(10),Integer(11),Integer(14)],[Integer(10),Integer(12),Integer(13),
↳Integer(11)],
...          [Integer(12),Integer(19),Integer(15),Integer(13)],[Integer(20),
↳Integer(16),Integer(14),Integer(15)],[Integer(16),Integer(20),Integer(17),
↳Integer(2)]]])
>>> L.alexander_polynomial()
↳needs sage.modules
1

```

Some additional examples:

```

sage: # needs sage.modules
sage: B = BraidGroup(2)
sage: L = Link(B([1]))
sage: L.alexander_polynomial()
1
sage: L = Link(B.one())
sage: L.alexander_polynomial()
1
sage: B = BraidGroup(3)
sage: L = Link(B([1, 2, 1, 2]))
sage: L.alexander_polynomial()
t^-1 - 1 + t

```

```

>>> from sage.all import *
>>> # needs sage.modules
>>> B = BraidGroup(Integer(2))
>>> L = Link(B([Integer(1)]))
>>> L.alexander_polynomial()
1
>>> L = Link(B.one())
>>> L.alexander_polynomial()
1
>>> B = BraidGroup(Integer(3))
>>> L = Link(B([Integer(1), Integer(2), Integer(1), Integer(2)]))
>>> L.alexander_polynomial()
t^-1 - 1 + t

```

When the Seifert surface is disconnected, the Alexander polynomial is defined to be 0:

```
sage: # needs sage.modules
sage: B = BraidGroup(4)
sage: L = Link(B([1, 3]))
sage: L.alexander_polynomial()
0
```

```
>>> from sage.all import *
>>> # needs sage.modules
>>> B = BraidGroup(Integer(4))
>>> L = Link(B([Integer(1), Integer(3)]))
>>> L.alexander_polynomial()
0
```

### See also

`conway_polynomial()`

**arcs** (*presentation='pd'*)

Return the arcs of `self`.

Arcs are the connected components of the planar diagram.

INPUT:

- `presentation` – one of the following:
  - 'pd' – the arcs are returned as lists of parts in the PD code
  - 'gauss\_code' – the arcs are returned as pieces of the Gauss code that start with a negative number, and end with the following negative one; if there exist a closed arc, it is returned as a list of positive numbers only

OUTPUT: list of lists representing the arcs based upon `presentation`

EXAMPLES:

```
sage: K = Knot([[1, -2, 3, -1, 2, -3]], [1, 1, 1])
sage: K.arcs()
[[1, 2], [3, 4], [5, 6]]
sage: K.arcs(presentation='gauss_code')
[[-3, 1, -2], [-2, 3, -1], [-1, 2, -3]]
```

```
>>> from sage.all import *
>>> K = Knot([Integer(1), -Integer(2), Integer(3), -Integer(1), Integer(2), -Integer(3)], [Integer(1), Integer(1), Integer(1)])
>>> K.arcs()
[[1, 2], [3, 4], [5, 6]]
>>> K.arcs(presentation='gauss_code')
[[-3, 1, -2], [-2, 3, -1], [-1, 2, -3]]
```

```
sage: L = Link([1, 2, 3, 4], [3, 2, 1, 4])
sage: L.arcs()
[[2, 4], [1], [3]]
sage: L.arcs(presentation='gauss_code')
```

(continues on next page)

(continued from previous page)

```

[[-2, -1], [-1, -2], [2, 1]]
sage: L.gauss_code()
[[-1, -2], [2, 1]]

```

```

>>> from sage.all import *
>>> L = Link([[Integer(1), Integer(2), Integer(3), Integer(4)], [Integer(3), Integer(2), Integer(1), Integer(4)]])
>>> L.arcs()
[[2, 4], [1], [3]]
>>> L.arcs(presentation='gauss_code')
[[-2, -1], [-1, -2], [2, 1]]
>>> L.gauss_code()
[[-1, -2], [2, 1]]

```

**braid** (*remove\_loops=False*)

Return a braid representation of *self*.

INPUT:

- *remove\_loops* – boolean (default: *False*); if set to *True* loops will be removed first. This can reduce the number of strands needed for an ambient isotopic braid closure. However, this can lead to a loss of the regular isotopy.

OUTPUT: an element in the braid group

### Warning

For the unknot with no crossings, this returns the identity of the braid group with 2 strands because this disregards strands with no crossings.

EXAMPLES:

```

sage: L = Link([[2, 4, 1, 3], [4, 2, 3, 1]])
sage: L.braid()
s^2
sage: L = Link([[-1, 2, -3, 1, -2, 3]], [-1, -1, -1])
sage: L.braid()
s^-3
sage: L = Link([[1, 7, 2, 8], [8, 5, 9, 4], [3, 10, 4, 9], [10, 6, 7, 1], [5, 2, 6, 3]])
sage: L.braid()
(s0*s1^-1)^2*s1^-1

```

```

>>> from sage.all import *
>>> L = Link([[Integer(2), Integer(4), Integer(1), Integer(3)], [Integer(4), Integer(2), Integer(3), Integer(1)]])
>>> L.braid()
s^2
>>> L = Link([[-Integer(1), Integer(2), -Integer(3), Integer(1), -Integer(2), Integer(3)], [-Integer(1), -Integer(1), -Integer(1)]])
>>> L.braid()
s^-3
>>> L = Link([[Integer(1), Integer(7), Integer(2), Integer(8)], [Integer(8),

```

(continues on next page)

(continued from previous page)

```

↪Integer(5), Integer(9), Integer(4)], [Integer(3), Integer(10), Integer(4),
↪Integer(9)], [Integer(10), Integer(6), Integer(7), Integer(1)], [Integer(5),
↪Integer(2), Integer(6), Integer(3)]]
>>> L.braid()
(s0*s1^-1)^2*s1^-1

```

using remove\_loops=True:

```

sage: L = Link([[2, 7, 1, 1], [7, 3, 9, 2], [4, 11, 3, 9], [11, 5, 5, 4]])
sage: L.braid()
s0*s1^-1*s2*s3^-1
sage: L.braid(remove_loops=True)
1

```

```

>>> from sage.all import *
>>> L = Link([[Integer(2), Integer(7), Integer(1), Integer(1)], [Integer(7), ↪
↪Integer(3), Integer(9), Integer(2)], [Integer(4), Integer(11), Integer(3), ↪
↪Integer(9)], [Integer(11), Integer(5), Integer(5), Integer(4)]]
>>> L.braid()
s0*s1^-1*s2*s3^-1
>>> L.braid(remove_loops=True)
1

```

**coloring\_maps** (*n=None*, *finitely\_presented=False*)

Return the  $n$ -coloring maps of `self`. These are group homomorphisms from the fundamental group of `self` to the  $n$ -th dihedral group.

INPUT:

- $n$  – the number of colors to consider (if omitted the value of the determinant of `self` will be taken). Note that there are no coloring maps if  $n$  is coprime to the determinant of `self`
- `finitely_presented` – boolean (default: `False`); whether to choose the dihedral groups as finitely presented groups. If not set to `True` they are represented as permutation groups.

OUTPUT:

a list of group homomorphisms from the fundamental group of `self` to the  $n$ -th dihedral group (represented according to the key argument `finitely_presented`).

EXAMPLES:

```

sage: L5a1_1 = Link([[8, 2, 9, 1], [10, 7, 5, 8], [4, 10, 1, 9],
....:               [2, 5, 3, 6], [6, 3, 7, 4]])
sage: L5a1_1.determinant()
8
sage: L5a1_1.coloring_maps(2)
[Group morphism:
  From: Finitely presented group < x0, x1, x2, x3, x4 | x4*x1*x0^-1*x1^-1, ↪
↪x0*x4^-1*x3^-1*x4, x2*x0*x1^-1*x0^-1, x1*x3^-1*x2^-1*x3, x3*x2^-1*x4^-1*x2 >
  To:   Dihedral group of order 4 as a permutation group,
Group morphism:
  From: Finitely presented group < x0, x1, x2, x3, x4 | x4*x1*x0^-1*x1^-1, ↪
↪x0*x4^-1*x3^-1*x4, x2*x0*x1^-1*x0^-1, x1*x3^-1*x2^-1*x3, x3*x2^-1*x4^-1*x2 >
  To:   Dihedral group of order 4 as a permutation group]

```

(continues on next page)

(continued from previous page)

```

sage: col_maps = L5a1_1.coloring_maps(4); len(col_maps)
12
sage: col_maps = L5a1_1.coloring_maps(5); len(col_maps)
0
sage: col_maps = L5a1_1.coloring_maps(12); len(col_maps)
36
sage: col_maps = L5a1_1.coloring_maps(); len(col_maps)
56

```

```

>>> from sage.all import *
>>> L5a1_1 = Link([[Integer(8), Integer(2), Integer(9), Integer(1)],
↳ [Integer(10), Integer(7), Integer(5), Integer(8)], [Integer(4), Integer(10),
↳ Integer(1), Integer(9)],
...               [Integer(2), Integer(5), Integer(3), Integer(6)],
↳ [Integer(6), Integer(3), Integer(7), Integer(4)])
>>> L5a1_1.determinant()
8
>>> L5a1_1.coloring_maps(Integer(2))
[Group morphism:
  From: Finitely presented group < x0, x1, x2, x3, x4 | x4*x1*x0^-1*x1^-1,
↳ x0*x4^-1*x3^-1*x4, x2*x0*x1^-1*x0^-1, x1*x3^-1*x2^-1*x3, x3*x2^-1*x4^-1*x2 >
  To:   Dihedral group of order 4 as a permutation group,
Group morphism:
  From: Finitely presented group < x0, x1, x2, x3, x4 | x4*x1*x0^-1*x1^-1,
↳ x0*x4^-1*x3^-1*x4, x2*x0*x1^-1*x0^-1, x1*x3^-1*x2^-1*x3, x3*x2^-1*x4^-1*x2 >
  To:   Dihedral group of order 4 as a permutation group]
>>> col_maps = L5a1_1.coloring_maps(Integer(4)); len(col_maps)
12
>>> col_maps = L5a1_1.coloring_maps(Integer(5)); len(col_maps)
0
>>> col_maps = L5a1_1.coloring_maps(Integer(12)); len(col_maps)
36
>>> col_maps = L5a1_1.coloring_maps(); len(col_maps)
56

```

applying the map:

```

sage: cm1 = col_maps[0]
sage: gs = L5a1_1.fundamental_group().gens()
sage: d = cm1(gs[0]); d
(1,8) (2,7) (3,6) (4,5)
sage: d.parent()
Dihedral group of order 16 as a permutation group

```

```

>>> from sage.all import *
>>> cm1 = col_maps[Integer(0)]
>>> gs = L5a1_1.fundamental_group().gens()
>>> d = cm1(gs[Integer(0)]); d
(1,8) (2,7) (3,6) (4,5)
>>> d.parent()
Dihedral group of order 16 as a permutation group

```

using the finitely presented dihedral group:

```
sage: col_maps = L5a1_1.coloring_maps(2, finitely_presented=True)
sage: d = col_maps[0](gs[1]); d
b*a
sage: d.parent()
Finitely presented group < a, b | a^2, b^2, (a*b)^2 >
```

```
>>> from sage.all import *
>>> col_maps = L5a1_1.coloring_maps(Integer(2), finitely_presented=True)
>>> d = col_maps[Integer(0)](gs[Integer(1)]); d
b*a
>>> d.parent()
Finitely presented group < a, b | a^2, b^2, (a*b)^2 >
```

#### REFERENCES:

- [Wikipedia article Fox\\_n-coloring](#)
- Chapter 3 of [Liv1993]

#### See also

`is_colorable()` and `colorings()`

**colorings** (*n=None*)

Return the *n*-colorings of *self*.

INPUT:

- *n* – the number of colors to consider (if omitted the value of the determinant of *self* will be taken). Note that there are no colorings if *n* is coprime to the determinant of *self*

OUTPUT:

a list with the colorings. Each coloring is represented as a dictionary that maps a tuple of the edges forming each arc (as in the PD code) to the index of the corresponding color.

EXAMPLES:

```
sage: K = Link([[1, -2, 3, -1, 2, -3]], [1, 1, 1])
sage: K.colorings(3)
↳needs sage.libs.pari sage.modules
[{(1, 2): 0, (3, 4): 1, (5, 6): 2},
 {(1, 2): 0, (3, 4): 2, (5, 6): 1},
 {(1, 2): 1, (3, 4): 0, (5, 6): 2},
 {(1, 2): 1, (3, 4): 2, (5, 6): 0},
 {(1, 2): 2, (3, 4): 0, (5, 6): 1},
 {(1, 2): 2, (3, 4): 1, (5, 6): 0}]
sage: K.pd_code()
[[4, 2, 5, 1], [2, 6, 3, 5], [6, 4, 1, 3]]
sage: K.arcs('pd')
[[1, 2], [3, 4], [5, 6]]
```

```

>>> from sage.all import *
>>> K = Link([[Integer(1), -Integer(2), Integer(3), -Integer(1), Integer(2),
↳ -Integer(3)]], [Integer(1), Integer(1), Integer(1)])
>>> K.colorings(Integer(3))
↳ # needs sage.libs.pari sage.modules
[{(1, 2): 0, (3, 4): 1, (5, 6): 2},
 {(1, 2): 0, (3, 4): 2, (5, 6): 1},
 {(1, 2): 1, (3, 4): 0, (5, 6): 2},
 {(1, 2): 1, (3, 4): 2, (5, 6): 0},
 {(1, 2): 2, (3, 4): 0, (5, 6): 1},
 {(1, 2): 2, (3, 4): 1, (5, 6): 0}]
>>> K.pd_code()
[[4, 2, 5, 1], [2, 6, 3, 5], [6, 4, 1, 3]]
>>> K.arcs('pd')
[[1, 2], [3, 4], [5, 6]]

```

Note that  $n$  is not the number of different colors to be used. It can be looked upon the size of the color palette:

```

sage: K = Knots().from_table(9, 15)
sage: cols = K.colorings(13); len(cols)
156
sage: max(cols[0].values())
12
sage: max(cols[13].values())
9

```

```

>>> from sage.all import *
>>> K = Knots().from_table(Integer(9), Integer(15))
>>> cols = K.colorings(Integer(13)); len(cols)
156
>>> max(cols[Integer(0)].values())
12
>>> max(cols[Integer(13)].values())
9

```

#### REFERENCES:

- [Wikipedia article Fox\\_n-coloring](#)
- Chapter 3 of [Liv1993]

#### See also

`is_colorable()` and `coloring_maps()`

#### `conway_polynomial()`

Return the Conway polynomial of `self`.

This is closely related to the Alexander polynomial.

See [Wikipedia article Alexander\\_polynomial](#) for the definition.

#### EXAMPLES:

```

sage: # needs sage.modules
sage: B = BraidGroup(3)
sage: L = Link(B([1, -2, 1, -2]))
sage: L.conway_polynomial()
-t^2 + 1
sage: Link([[1, 5, 2, 4], [3, 9, 4, 8], [5, 1, 6, 10],
.....:      [7, 3, 8, 2], [9, 7, 10, 6]])
Link with 1 component represented by 5 crossings
sage: _.conway_polynomial()
2*t^2 + 1
sage: B = BraidGroup(4)
sage: L = Link(B([1, 3]))
sage: L.conway_polynomial()
0

```

```

>>> from sage.all import *
>>> # needs sage.modules
>>> B = BraidGroup(Integer(3))
>>> L = Link(B([Integer(1), -Integer(2), Integer(1), -Integer(2)]))
>>> L.conway_polynomial()
-t^2 + 1
>>> Link([[Integer(1), Integer(5), Integer(2), Integer(4)], [Integer(3),
↪Integer(9), Integer(4), Integer(8)], [Integer(5), Integer(1), Integer(6),
↪Integer(10)],
...      [Integer(7), Integer(3), Integer(8), Integer(2)], [Integer(9),
↪Integer(7), Integer(10), Integer(6)]])
Link with 1 component represented by 5 crossings
>>> _.conway_polynomial()
2*t^2 + 1
>>> B = BraidGroup(Integer(4))
>>> L = Link(B([Integer(1), Integer(3)]))
>>> L.conway_polynomial()
0

```

**See also**

`alexander_polynomial()`

**determinant()**

Return the determinant of `self`.

**EXAMPLES:**

```

sage: # needs sage.modules
sage: B = BraidGroup(4)
sage: L = Link(B([-1, 2, 1, 2]))
sage: L.determinant()
1
sage: B = BraidGroup(8)
sage: L = Link(B([2, 4, 2, 3, 1, 2]))
sage: L.determinant()
3

```

(continues on next page)

(continued from previous page)

```

sage: L = Link(B([1]*16 + [2,1,2,1,2,2,2,2,2,2,1,2,1,2,-1,2,-2]))
sage: L.determinant()
65
sage: B = BraidGroup(3)
sage: Link(B([1, 2, 1, 1, 2])).determinant()
4

```

```

>>> from sage.all import *
>>> # needs sage.modules
>>> B = BraidGroup(Integer(4))
>>> L = Link(B([-Integer(1), Integer(2), Integer(1), Integer(2)]))
>>> L.determinant()
1
>>> B = BraidGroup(Integer(8))
>>> L = Link(B([Integer(2), Integer(4), Integer(2), Integer(3), Integer(1), ↵
↵Integer(2)]))
>>> L.determinant()
3
>>> L = Link(B([Integer(1)]*Integer(16) + [Integer(2), Integer(1), Integer(2),
↵Integer(1), Integer(2), Integer(2), Integer(2), Integer(2),
↵Integer(2), Integer(2), Integer(1), Integer(2), Integer(1), Integer(2), -
↵Integer(1), Integer(2), -Integer(2)]))
>>> L.determinant()
65
>>> B = BraidGroup(Integer(3))
>>> Link(B([Integer(1), Integer(2), Integer(1), Integer(1), Integer(2)]))
↵determinant()
4

```

## REFERENCES:

- Definition 6.6.3 in [Cro2004]

**dowker\_notation()**

Return the Dowker notation of `self`.

Similar to the PD code we number the components, so every crossing is represented by four numbers. We focus on the incoming entities of the under and the overcrossing. It is the pair of incoming undercrossing and the incoming overcrossing. This information at every crossing gives the Dowker notation.

## OUTPUT:

A list containing the pair of incoming under cross and the incoming over cross.

## EXAMPLES:

```

sage: L = Link([[-1, 2, -3, 4, 5, 1, -2, 6, 7, 3, -4, -7, -6, -5]],
....:          [-1, -1, -1, -1, 1, -1, 1]])
sage: L.dowker_notation()
[(1, 6), (7, 2), (3, 10), (11, 4), (14, 5), (13, 8), (12, 9)]

sage: B = BraidGroup(4)
sage: L = Link(B([1, 2, 1, 2]))
sage: L.dowker_notation()
[(2, 1), (3, 5), (6, 4), (7, 9)]

```

(continues on next page)

(continued from previous page)

```
sage: L = Link([[1, 4, 2, 3], [4, 1, 3, 2]])
sage: L.dowker_notation()
[(1, 3), (4, 2)]
```

```
>>> from sage.all import *
>>> L = Link([[-Integer(1), Integer(2), -Integer(3), Integer(4), Integer(5),
↳ Integer(1), -Integer(2), Integer(6), Integer(7), Integer(3), -Integer(4), -
↳ Integer(7), -Integer(6), -Integer(5)]],
...          [-Integer(1), -Integer(1), -Integer(1), -Integer(1), Integer(1),
↳ -Integer(1), Integer(1)])
>>> L.dowker_notation()
[(1, 6), (7, 2), (3, 10), (11, 4), (14, 5), (13, 8), (12, 9)]

>>> B = BraidGroup(Integer(4))
>>> L = Link(B([Integer(1), Integer(2), Integer(1), Integer(2)]))
>>> L.dowker_notation()
[(2, 1), (3, 5), (6, 4), (7, 9)]
>>> L = Link([[Integer(1), Integer(4), Integer(2), Integer(3)], [Integer(4),
↳ Integer(1), Integer(3), Integer(2)]])
>>> L.dowker_notation()
[(1, 3), (4, 2)]
```

**fundamental\_group** (*presentation='wirtinger'*)

Return the fundamental group of the complement of self.

INPUT:

- *presentation* – string; one of the following:
  - 'wirtinger' – (default) the Wirtinger presentation (see [Wikipedia article Link\\_group](#))
  - 'braid' – the presentation is given by the braid action on the free group (see chapter 2 of [Bir1975])

OUTPUT: a finitely presented group

EXAMPLES:

```
sage: L = Link([[1, 4, 3, 2], [3, 4, 1, 2]])
sage: L.fundamental_group()
Finitely presented group < x0, x1, x2 | x1*x0^-1*x2^-1*x0, x2*x0*x1^-1*x0^-1 >
sage: L.fundamental_group('braid')
Finitely presented group < x0, x1 | 1, 1 >
```

```
>>> from sage.all import *
>>> L = Link([[Integer(1), Integer(4), Integer(3), Integer(2)], [Integer(3),
↳ Integer(4), Integer(1), Integer(2)]])
>>> L.fundamental_group()
Finitely presented group < x0, x1, x2 | x1*x0^-1*x2^-1*x0, x2*x0*x1^-1*x0^-1 >
>>> L.fundamental_group('braid')
Finitely presented group < x0, x1 | 1, 1 >
```

We can see, for instance, that the two presentations of the group of the figure eight knot correspond to isomorphic groups:

```

sage: K8 = Knot([[1, -2, 4, -3, 2, -1, 3, -4]], [1, 1, -1, -1])
sage: GA = K8.fundamental_group(); GA
Finitely presented group < x0, x1, x2, x3 |
  x2*x0*x3^-1*x0^-1, x0*x2*x1^-1*x2^-1,
  x1*x3^-1*x2^-1*x3, x3*x1^-1*x0^-1*x1 >
sage: GB = K8.fundamental_group(presentation='braid'); GB
Finitely presented group
< x0, x1, x2 | x1*x2^-1*x1^-1*x0*x1*x2*x1*x2^-1*x1^-1*x0^-1*x1*x2*x1^-1*x0^-
↪1,
                x1*x2^-1*x1^-1*x0*x1*x2*x1^-1*x2^-1*x1^-1*x0^-1*x1*x2*x1^-
↪1*x0*x1*x2*x1*x2^-1*x1^-1*x0^-1*x1*x2*x1^-2,
                x1*x2^-1*x1^-1*x0*x1*x2*x1^-1*x2^-1 >
sage: GA.simplified()
Finitely presented group
< x0, x1 | x1^-1*x0*x1*x0^-1*x1*x0*x1^-1*x0^-1*x1*x0^-1 >
sage: GB.simplified()
Finitely presented group
< x0, x2 | x2^-1*x0*x2^-1*x0^-1*x2*x0*x2^-1*x0*x2*x0^-1 >

```

```

>>> from sage.all import *
>>> K8 = Knot([[[Integer(1), -Integer(2), Integer(4), -Integer(3), Integer(2),
↪ -Integer(1), Integer(3), -Integer(4)]]], [Integer(1), Integer(1), -
↪Integer(1), -Integer(1)]]])
>>> GA = K8.fundamental_group(); GA
Finitely presented group < x0, x1, x2, x3 |
  x2*x0*x3^-1*x0^-1, x0*x2*x1^-1*x2^-1,
  x1*x3^-1*x2^-1*x3, x3*x1^-1*x0^-1*x1 >
>>> GB = K8.fundamental_group(presentation='braid'); GB
Finitely presented group
< x0, x1, x2 | x1*x2^-1*x1^-1*x0*x1*x2*x1*x2^-1*x1^-1*x0^-1*x1*x2*x1^-1*x0^-
↪1,
                x1*x2^-1*x1^-1*x0*x1*x2*x1^-1*x2^-1*x1^-1*x0^-1*x1*x2*x1^-
↪1*x0*x1*x2*x1*x2^-1*x1^-1*x0^-1*x1*x2*x1^-2,
                x1*x2^-1*x1^-1*x0*x1*x2*x1^-1*x2^-1 >
>>> GA.simplified()
Finitely presented group
< x0, x1 | x1^-1*x0*x1*x0^-1*x1*x0*x1^-1*x0^-1*x1*x0^-1 >
>>> GB.simplified()
Finitely presented group
< x0, x2 | x2^-1*x0*x2^-1*x0^-1*x2*x0*x2^-1*x0*x2*x0^-1 >

```

#### **gauss\_code()**

Return the Gauss code of `self`.

The Gauss code is generated by the following procedure:

- Number the crossings from 1 to  $n$ .
- Select a point on the knot and start moving along the component.
- At each crossing, take the number of the crossing, along with sign, which is  $-$  if it is an undercrossing and  $+$  if it is an overcrossing.

EXAMPLES:

```

sage: L = Link([[1, 4, 2, 3], [4, 1, 3, 2]])
sage: L.gauss_code()
[[-1, 2], [1, -2]]

sage: B = BraidGroup(8)
sage: L = Link(B([1, -2, 1, -2, -2]))
sage: L.gauss_code()
[[-1, 3, -4, 5], [1, -2, 4, -5, 2, -3]]

sage: L = Link([[-1, 2], [-3, 4], [1, 3, -4, -2]], [-1, -1, 1, 1])
sage: L.gauss_code()
[[-1, 2], [-3, 4], [1, 3, -4, -2]]

```

```

>>> from sage.all import *
>>> L = Link([[Integer(1), Integer(4), Integer(2), Integer(3)], [Integer(4), -
↳ Integer(1), Integer(3), Integer(2)]])
>>> L.gauss_code()
[[-1, 2], [1, -2]]

>>> B = BraidGroup(Integer(8))
>>> L = Link(B([Integer(1), -Integer(2), Integer(1), -Integer(2), -
↳ Integer(2)]))
>>> L.gauss_code()
[[-1, 3, -4, 5], [1, -2, 4, -5, 2, -3]]

>>> L = Link([[-Integer(1), Integer(2)], [-Integer(3), Integer(4)], -
↳ [Integer(1), Integer(3), -Integer(4), -Integer(2)], [-Integer(1), -
↳ Integer(1), Integer(1)]]])
>>> L.gauss_code()
[[-1, 2], [-3, 4], [1, 3, -4, -2]]

```

**genus()**

Return the genus of self.

EXAMPLES:

```

sage: B = BraidGroup(4)
sage: L = Link(B([-1, 3, 1, 3]))
sage: L.genus()
0
sage: L = Link(B([1, 3]))
sage: L.genus()
0
sage: B = BraidGroup(8)
sage: L = Link(B([-2, 4, 1, 6, 1, 4]))
sage: L.genus()
0
sage: L = Link(B([1, 2, 1, 2]))
sage: L.genus()
1

```

```

>>> from sage.all import *
>>> B = BraidGroup(Integer(4))

```

(continues on next page)

(continued from previous page)

```

>>> L = Link(B([-Integer(1), Integer(3), Integer(1), Integer(3)]))
>>> L.genus()
0
>>> L = Link(B([Integer(1), Integer(3)]))
>>> L.genus()
0
>>> B = BraidGroup(Integer(8))
>>> L = Link(B([-Integer(2), Integer(4), Integer(1), Integer(6), Integer(1),
↪ Integer(4)]))
>>> L.genus()
0
>>> L = Link(B([Integer(1), Integer(2), Integer(1), Integer(2)]))
>>> L.genus()
1

```

**get\_knotinfo** (*mirror\_version=True, unique=True*)

Identify this link as an item of the KnotInfo database (if possible).

INPUT:

- *mirror\_version* – boolean (default: True); if set to False the result of the method will be just the instance of *KnotInfoBase* (by default the result is a tuple of the instance and an enum, see explanation of the output below)
- *unique* – boolean (default: True); this only affects the case where a unique identification is not possible. If set to False you can obtain a matching list (see explanation of the output below).

OUTPUT:

If *self* is a knot, then an element of the free monoid over prime knots constructed from the KnotInfo database is returned. More explicitly this is an element of *FreeKnotInfoMonoidElement*. Else a tuple (*K*, *m*) is returned where *K* is an instance of *KnotInfoBase* and *m* an instance of *SymmetryMutant* (for chiral links) specifying the symmetry mutant of *K* to which *self* is isotopic. The value of *m* is unknown if it cannot be determined uniquely and the keyword option *unique=False* is given.

For proper links, if the orientation mutant cannot be uniquely determined, *K* will be a series of links gathering all links having the same unoriented name, that is an instance of *KnotInfoSeries*.

If *mirror\_version* is set to False then the result is just *K* (that is: *m* is suppressed).

If it is not possible to determine a unique result a *NotImplementedError* will be raised. To avoid this you can set *unique* to False. You will get a list of matching candidates instead.

#### Note

The identification of proper links may fail to be unique due to the following fact: In opposite to the database for knots, there are pairs of oriented mutants of an unoriented link which are isotopic to each other. For example *L5a1\_0* and *L5a1\_1* is such a pair.

This is because all combinatorial possible oriented mutants are listed with individual names regardless whether they are pairwise non isotopic or not. In such a case the identification is not unique and therefore a series of the links will be returned which gathers all having the same unoriented name.

To obtain the individual oriented links being isotopic to *self* use the keyword *unique* (see the examples for *L2a1\_1* and *L5a1\_0* below).

EXAMPLES:

```

sage: # optional - database_knotinfo
sage: L = Link([[4,1,5,2], [10,4,11,3], [5,17,6,16], [7,13,8,12],
....:          [18,10,19,9], [2,12,3,11], [13,21,14,20], [15,7,16,6],
....:          [22,17,1,18], [8,20,9,19], [21,15,22,14]])
sage: L.get_knotinfo()
KnotInfo['K11n_121m']
sage: K = KnotInfo.K10_25
sage: l = K.link()
sage: l.get_knotinfo()
KnotInfo['K10_25']
sage: k11 = KnotInfo.K11n_82.link()
sage: k11m = k11.mirror_image()
sage: k11mr = k11m.reverse()
sage: k11mr.get_knotinfo()
KnotInfo['K11n_82m']
sage: k11r = k11.reverse()
sage: k11r.get_knotinfo()
KnotInfo['K11n_82']
sage: k11rm = k11r.mirror_image()
sage: k11rm.get_knotinfo()
KnotInfo['K11n_82m']

```

```

>>> from sage.all import *
>>> # optional - database_knotinfo
>>> L = Link([[Integer(4), Integer(1), Integer(5), Integer(2)], [Integer(10),
↪ Integer(4), Integer(11), Integer(3)], [Integer(5), Integer(17), Integer(6),
↪ Integer(16)], [Integer(7), Integer(13), Integer(8), Integer(12)],
...          [Integer(18), Integer(10), Integer(19), Integer(9)], [Integer(2),
↪ Integer(12), Integer(3), Integer(11)], [Integer(13), Integer(21), Integer(14),
↪ Integer(20)], [Integer(15), Integer(7), Integer(16), Integer(6)],
...          [Integer(22), Integer(17), Integer(1), Integer(18)], [Integer(8),
↪ Integer(20), Integer(9), Integer(19)], [Integer(21), Integer(15), Integer(22),
↪ Integer(14)]])
>>> L.get_knotinfo()
KnotInfo['K11n_121m']
>>> K = KnotInfo.K10_25
>>> l = K.link()
>>> l.get_knotinfo()
KnotInfo['K10_25']
>>> k11 = KnotInfo.K11n_82.link()
>>> k11m = k11.mirror_image()
>>> k11mr = k11m.reverse()
>>> k11mr.get_knotinfo()
KnotInfo['K11n_82m']
>>> k11r = k11.reverse()
>>> k11r.get_knotinfo()
KnotInfo['K11n_82']
>>> k11rm = k11r.mirror_image()
>>> k11rm.get_knotinfo()
KnotInfo['K11n_82m']

```

Knots with more than 13 and multi-component links having more than 11 crossings cannot be identified. In addition non prime multi-component links or even links whose HOMFLY-PT polynomial is not irreducible

cannot be identified:

```
sage: b, = BraidGroup(2).gens()
sage: Link(b**13).get_knotinfo()      # optional - database_knotinfo
KnotInfo['K13a_4878']
sage: Link(b**14).get_knotinfo()
Traceback (most recent call last):
...
NotImplementedError: this link having more than 11 crossings cannot be_
↳determined

sage: Link([[1, 4, 2, 5], [3, 8, 4, 1], [5, 2, 6, 3],
.....:      [6, 10, 7, 9], [10, 8, 9, 7]])
Link with 2 components represented by 5 crossings
sage: _.get_knotinfo()                #_
↳needs sage.libs.homfly
Traceback (most recent call last):
...
NotImplementedError: this (possibly non prime) link cannot be determined
```

```
>>> from sage.all import *
>>> b, = BraidGroup(Integer(2)).gens()
>>> Link(b**Integer(13)).get_knotinfo()      # optional - database_knotinfo
KnotInfo['K13a_4878']
>>> Link(b**Integer(14)).get_knotinfo()
Traceback (most recent call last):
...
NotImplementedError: this link having more than 11 crossings cannot be_
↳determined

>>> Link([Integer(1), Integer(4), Integer(2), Integer(5)], [Integer(3),_
↳Integer(8), Integer(4), Integer(1)], [Integer(5), Integer(2), Integer(6),_
↳Integer(3)],
...      [Integer(6), Integer(10), Integer(7), Integer(9)], [Integer(10),_
↳Integer(8), Integer(9), Integer(7)])
Link with 2 components represented by 5 crossings
>>> _.get_knotinfo()                #_
↳needs sage.libs.homfly
Traceback (most recent call last):
...
NotImplementedError: this (possibly non prime) link cannot be determined
```

Lets identify the monster unknot:

```
sage: L = Link([3,1,2,4], [8,9,1,7], [5,6,7,3], [4,18,6,5],
.....:          [17,19,8,18], [9,10,11,14], [10,12,13,11],
.....:          [12,19,15,13], [20,16,14,15], [16,20,17,2]])
sage: L.get_knotinfo()
KnotInfo['K0_1']
```

```
>>> from sage.all import *
>>> L = Link([Integer(3), Integer(1), Integer(2), Integer(4)], [Integer(8),
↳Integer(9), Integer(1), Integer(7)], [Integer(5), Integer(6), Integer(7),
```

(continues on next page)

(continued from previous page)

```

↪Integer(3)], [Integer(4),Integer(18),Integer(6),Integer(5)],
...      [Integer(17),Integer(19),Integer(8),Integer(18)], [Integer(9),
↪Integer(10),Integer(11),Integer(14)], [Integer(10),Integer(12),Integer(13),
↪Integer(11)],
...      [Integer(12),Integer(19),Integer(15),Integer(13)], [Integer(20),
↪Integer(16),Integer(14),Integer(15)], [Integer(16),Integer(20),Integer(17),
↪Integer(2)]]
>>> L.get_knotinfo()
KnotInfo['K0_1']

```

Usage of option `mirror_version`:

```

sage: L.get_knotinfo(mirror_version=False) == KnotInfo.K0_1
True

```

```

>>> from sage.all import *
>>> L.get_knotinfo(mirror_version=False) == KnotInfo.K0_1
True

```

Usage of option `unique`:

```

sage: # optional - database_knotinfo
sage: l = K.link(K.items.gauss_notation)
sage: l.get_knotinfo()
Traceback (most recent call last):
...
NotImplementedError: this link cannot be uniquely determined
use keyword argument `unique` to obtain more details
sage: l.get_knotinfo(unique=False)
[KnotInfo['K10_25'], KnotInfo['K10_56']]
sage: t = (1, -2, 1, 1, -2, 1, -2, -2)
sage: l8 = Link(BraidGroup(3)(t))
sage: l8.get_knotinfo()
Traceback (most recent call last):
...
NotImplementedError: this link cannot be uniquely determined
use keyword argument `unique` to obtain more details
sage: l8.get_knotinfo(unique=False)
[(<KnotInfo.L8a19_0_0: 'L8a19{0,0}>', <SymmetryMutant.itself: 's'>),
 (<KnotInfo.L8a19_1_1: 'L8a19{1,1}>', <SymmetryMutant.itself: 's'>)]
sage: t = (2, -3, -3, -2, 3, 3, -2, 3, 1, -2, -2, 1)
sage: l12 = Link(BraidGroup(5)(t))
sage: l12.get_knotinfo()
Traceback (most recent call last):
...
NotImplementedError: this link having more than 11 crossings
cannot be uniquely determined
use keyword argument `unique` to obtain more details
sage: l12.get_knotinfo(unique=False)
[(<KnotInfo.L10n36_0: 'L10n36{0}>', <SymmetryMutant.unknown: '?'>),
 (<KnotInfo.L10n36_1: 'L10n36{1}>', <SymmetryMutant.unknown: '?'>),
 (<KnotInfo.L10n59_0: 'L10n59{0}>', <SymmetryMutant.itself: 's'>),

```

(continues on next page)

(continued from previous page)

```

(<KnotInfo.L10n59_1: 'L10n59{1}'>, <SymmetryMutant.itself: 's'>)]

>>> from sage.all import *
>>> # optional - database_knotinfo
>>> l = K.link(K.items.gauss_notation)
>>> l.get_knotinfo()
Traceback (most recent call last):
...
NotImplementedError: this link cannot be uniquely determined
use keyword argument `unique` to obtain more details
>>> l.get_knotinfo(unique=False)
[KnotInfo['K10_25'], KnotInfo['K10_56']]
>>> t = (Integer(1), -Integer(2), Integer(1), Integer(1), -Integer(2),
Integer(1), -Integer(2), -Integer(2))
>>> l8 = Link(BraidGroup(Integer(3))(t))
>>> l8.get_knotinfo()
Traceback (most recent call last):
...
NotImplementedError: this link cannot be uniquely determined
use keyword argument `unique` to obtain more details
>>> l8.get_knotinfo(unique=False)
[(<KnotInfo.L8a19_0_0: 'L8a19{0,0}'>, <SymmetryMutant.itself: 's'>),
 (<KnotInfo.L8a19_1_1: 'L8a19{1,1}'>, <SymmetryMutant.itself: 's'>)]
>>> t = (Integer(2), -Integer(3), -Integer(3), -Integer(2), Integer(3),
Integer(3), -Integer(2), Integer(3), Integer(1), -Integer(2), -Integer(2),
Integer(1))
>>> l12 = Link(BraidGroup(Integer(5))(t))
>>> l12.get_knotinfo()
Traceback (most recent call last):
...
NotImplementedError: this link having more than 11 crossings
cannot be uniquely determined
use keyword argument `unique` to obtain more details
>>> l12.get_knotinfo(unique=False)
[(<KnotInfo.L10n36_0: 'L10n36{0}'>, <SymmetryMutant.unknown: '?'>),
 (<KnotInfo.L10n36_1: 'L10n36{1}'>, <SymmetryMutant.unknown: '?'>),
 (<KnotInfo.L10n59_0: 'L10n59{0}'>, <SymmetryMutant.itself: 's'>),
 (<KnotInfo.L10n59_1: 'L10n59{1}'>, <SymmetryMutant.itself: 's'>)]

```

Furthermore, if the result is a complete series of oriented links having the same unoriented name (according to the note above) the option can be used to achieve more detailed information:

```

sage: L2a1 = Link(b**2)
sage: L2a1.get_knotinfo()
(Series of links L2a1, <SymmetryMutant.mixed: 'x'>)
sage: L2a1.get_knotinfo(unique=False)
[(<KnotInfo.L2a1_0: 'L2a1{0}'>, <SymmetryMutant.mirror_image: 'm'>),
 (<KnotInfo.L2a1_1: 'L2a1{1}'>, <SymmetryMutant.itself: 's'>)]

sage: KnotInfo.L5a1_0.inject()
Defining L5a1_0
sage: l5 = Link(L5a1_0.braid())

```

(continues on next page)

(continued from previous page)

```

sage: l5.get_knotinfo()
(Series of links L5a1, <SymmetryMutant.itself: 's'>)
sage: _[0].inject()
Defining L5a1
sage: list(L5a1)
[<KnotInfo.L5a1_0: 'L5a1{0}'>, <KnotInfo.L5a1_1: 'L5a1{1}'>]
sage: l5.get_knotinfo(unique=False)
[(<KnotInfo.L5a1_0: 'L5a1{0}'>, <SymmetryMutant.itself: 's'>),
 (<KnotInfo.L5a1_1: 'L5a1{1}'>, <SymmetryMutant.itself: 's'>)]

```

```

>>> from sage.all import *
>>> L2a1 = Link(b**Integer(2))
>>> L2a1.get_knotinfo()
(Series of links L2a1, <SymmetryMutant.mixed: 'x'>)
>>> L2a1.get_knotinfo(unique=False)
[(<KnotInfo.L2a1_0: 'L2a1{0}'>, <SymmetryMutant.mirror_image: 'm'>),
 (<KnotInfo.L2a1_1: 'L2a1{1}'>, <SymmetryMutant.itself: 's'>)]

>>> KnotInfo.L5a1_0.inject()
Defining L5a1_0
>>> l5 = Link(L5a1_0.braid())
>>> l5.get_knotinfo()
(Series of links L5a1, <SymmetryMutant.itself: 's'>)
>>> _[Integer(0)].inject()
Defining L5a1
>>> list(L5a1)
[<KnotInfo.L5a1_0: 'L5a1{0}'>, <KnotInfo.L5a1_1: 'L5a1{1}'>]
>>> l5.get_knotinfo(unique=False)
[(<KnotInfo.L5a1_0: 'L5a1{0}'>, <SymmetryMutant.itself: 's'>),
 (<KnotInfo.L5a1_1: 'L5a1{1}'>, <SymmetryMutant.itself: 's'>)]

```

Clarifying the series around the Perko pair ([Wikipedia article Perko pair](#)):

```

sage: for i in range(160, 166): # optional - database_knotinfo
.....:     K = Knots().from_table(10, i)
.....:     print('%s_%s' % (10, i), '--->', K.get_knotinfo())
10_160 ---> KnotInfo['K10_160']
10_161 ---> KnotInfo['K10_161m']
10_162 ---> KnotInfo['K10_162']
10_163 ---> KnotInfo['K10_163']
10_164 ---> KnotInfo['K10_164']
10_165 ---> KnotInfo['K10_165m']

```

```

>>> from sage.all import *
>>> for i in range(Integer(160), Integer(166)): # optional -_
↳database_knotinfo
...     K = Knots().from_table(Integer(10), i)
...     print('%s_%s' % (Integer(10), i), '--->', K.get_knotinfo())
10_160 ---> KnotInfo['K10_160']
10_161 ---> KnotInfo['K10_161m']
10_162 ---> KnotInfo['K10_162']
10_163 ---> KnotInfo['K10_163']

```

(continues on next page)

(continued from previous page)

```
10_164 ---> KnotInfo['K10_164']
10_165 ---> KnotInfo['K10_165m']
```

Clarifying the Perko series against SnapPy:

```
sage: import snappy                                # optional - snappy
...

sage: # optional - database_knotinfo snappy
sage: from sage.knots.knotinfo import KnotInfoSeries
sage: KnotInfoSeries(10, True, True)
Series of knots K10
sage: _.inject()
Defining K10
sage: for i in range(160, 166):
....:     K = K10(i)
....:     k = K.link(K.items.name, snappy=True)
....:     print(k, '--->', k.sage_link().get_knotinfo())
<Link 10_160: 1 comp; 10 cross> ---> KnotInfo['K10_160']
<Link 10_161: 1 comp; 10 cross> ---> KnotInfo['K10_161m']
<Link 10_162: 1 comp; 10 cross> ---> KnotInfo['K10_161']
<Link 10_163: 1 comp; 10 cross> ---> KnotInfo['K10_162']
<Link 10_164: 1 comp; 10 cross> ---> KnotInfo['K10_163']
<Link 10_165: 1 comp; 10 cross> ---> KnotInfo['K10_164']
sage: snappy.Link('10_166')
<Link 10_166: 1 comp; 10 cross>
sage: _.sage_link().get_knotinfo()
KnotInfo['K10_165m']
```

```
>>> from sage.all import *
>>> import snappy                                # optional - snappy
...

>>> # optional - database_knotinfo snappy
>>> from sage.knots.knotinfo import KnotInfoSeries
>>> KnotInfoSeries(Integer(10), True, True)
Series of knots K10
>>> _.inject()
Defining K10
>>> for i in range(Integer(160), Integer(166)):
...     K = K10(i)
...     k = K.link(K.items.name, snappy=True)
...     print(k, '--->', k.sage_link().get_knotinfo())
<Link 10_160: 1 comp; 10 cross> ---> KnotInfo['K10_160']
<Link 10_161: 1 comp; 10 cross> ---> KnotInfo['K10_161m']
<Link 10_162: 1 comp; 10 cross> ---> KnotInfo['K10_161']
<Link 10_163: 1 comp; 10 cross> ---> KnotInfo['K10_162']
<Link 10_164: 1 comp; 10 cross> ---> KnotInfo['K10_163']
<Link 10_165: 1 comp; 10 cross> ---> KnotInfo['K10_164']
>>> snappy.Link('10_166')
<Link 10_166: 1 comp; 10 cross>
>>> _.sage_link().get_knotinfo()
```

(continues on next page)

(continued from previous page)

```
KnotInfo['K10_165m']
```

Another pair of confusion (see the corresponding [Warning](#)):

```
sage: # optional - database_knotinfo snappy
sage: Ks10_86 = snappy.Link('10_86')
sage: Ks10_83 = snappy.Link('10_83')
sage: Ks10_86.sage_link().get_knotinfo(unique=False)
[KnotInfo['K10_83c'], KnotInfo['K10_83m']]
sage: Ks10_83.sage_link().get_knotinfo(unique=False)
[KnotInfo['K10_86'], KnotInfo['K10_86r']]
```

```
>>> from sage.all import *
>>> # optional - database_knotinfo snappy
>>> Ks10_86 = snappy.Link('10_86')
>>> Ks10_83 = snappy.Link('10_83')
>>> Ks10_86.sage_link().get_knotinfo(unique=False)
[KnotInfo['K10_83c'], KnotInfo['K10_83m']]
>>> Ks10_83.sage_link().get_knotinfo(unique=False)
[KnotInfo['K10_86'], KnotInfo['K10_86r']]
```

Non prime knots can be detected, as well:

```
sage: b = BraidGroup(4)((1, 2, 2, 2, -1, 2, 2, 2, -3, -3, -3))
sage: Kb = Knot(b)
sage: Kb.get_knotinfo()
KnotInfo['K3_1']^2*KnotInfo['K3_1m']

sage: K = Link([[4, 2, 5, 1], [8, 6, 9, 5], [6, 3, 7, 4], [2, 7, 3, 8],
....: [10, 15, 11, 16], [12, 21, 13, 22], [14, 11, 15, 12], [16, 9, 17, 10],
....: [18, 25, 19, 26], [20, 23, 21, 24], [22, 13, 23, 14], [24, 19, 25, 20],
....: [26, 17, 1, 18]])
sage: K.get_knotinfo() # optional - database_knotinfo, long time
KnotInfo['K4_1']*KnotInfo['K9_2m']
```

```
>>> from sage.all import *
>>> b = BraidGroup(Integer(4))((Integer(1), Integer(2), Integer(2),
↳ Integer(2), -Integer(1), Integer(2), Integer(2), Integer(2), -Integer(3), -
↳ Integer(3), -Integer(3)))
>>> Kb = Knot(b)
>>> Kb.get_knotinfo()
KnotInfo['K3_1']^2*KnotInfo['K3_1m']

>>> K = Link([[Integer(4), Integer(2), Integer(5), Integer(1)], [Integer(8),
↳ Integer(6), Integer(9), Integer(5)], [Integer(6), Integer(3), Integer(7),
↳ Integer(4)], [Integer(2), Integer(7), Integer(3), Integer(8)],
... [Integer(10), Integer(15), Integer(11), Integer(16)], [Integer(12),
↳ Integer(21), Integer(13), Integer(22)], [Integer(14), Integer(11),
↳ Integer(15), Integer(12)], [Integer(16), Integer(9), Integer(17),
↳ Integer(10)],
... [Integer(18), Integer(25), Integer(19), Integer(26)], [Integer(20),
↳ Integer(23), Integer(21), Integer(24)], [Integer(22), Integer(13),
```

(continues on next page)

(continued from previous page)

```

↪Integer(23), Integer(14)], [Integer(24), Integer(19), Integer(25), ↪
↪Integer(20)],
... [Integer(26), Integer(17), Integer(1), Integer(18)]]
>>> K.get_knotinfo()      # optional - database_knotinfo, long time
KnotInfo['K4_1']*KnotInfo['K9_2m']

```

**homfly\_polynomial**(var1=None, var2=None, normalization='lm')

Return the HOMFLY polynomial of `self`.

The HOMFLY polynomial  $P(K)$  of a link  $K$  is a Laurent polynomial in two variables defined using skein relations and for the unknot  $U$ , we have  $P(U) = 1$ .

INPUT:

- `var1` – (default: 'L') the first variable. If `normalization` is set to `az` resp. `vz` the default is `a` resp. `v`
- `var2` – (default: 'M') the second variable. If `normalization` is set to `az` resp. `vz` the default is `z`
- `normalization` – (default: `lm`) the system of coordinates and can be one of the following:
  - 'lm' – corresponding to the Skein relation  $L \cdot P(K_+) + L^{-1} \cdot P(K_-) + M \cdot P(K_0) = 0$
  - 'az' – corresponding to the Skein relation  $a \cdot P(K_+) - a^{-1} \cdot P(K_-) = z \cdot P(K_0)$
  - 'vz' – corresponding to the Skein relation  $v^{-1} \cdot P(K_+) - v \cdot P(K_-) = z \cdot P(K_0)$

where  $P(K_+)$ ,  $P(K_-)$  and  $P(K_0)$  represent the HOMFLY polynomials of three links that vary only in one crossing; that is the positive, negative, or smoothed links respectively

OUTPUT: a Laurent polynomial over the integers

#### Note

Use the 'az' normalization to agree with the data in [KnotAtlas]

Use the 'vz' normalization to agree with the data [KnotInfo](#).

EXAMPLES:

We give some examples:

```

sage: g = BraidGroup(2).gen(0)
sage: K = Knot(g^5)
sage: K.homfly_polynomial()
↪needs sage.libs.homfly
L^-4*M^4 - 4*L^-4*M^2 + 3*L^-4 - L^-6*M^2 + 2*L^-6

```

```

>>> from sage.all import *
>>> g = BraidGroup(Integer(2)).gen(Integer(0))
>>> K = Knot(g**Integer(5))
>>> K.homfly_polynomial()
↪needs sage.libs.homfly
L^-4*M^4 - 4*L^-4*M^2 + 3*L^-4 - L^-6*M^2 + 2*L^-6

```

The Hopf link:

```
sage: L = Link([[1,4,2,3],[4,1,3,2]])
sage: L.homfly_polynomial('x', 'y') #_
↪needs sage.libs.homfly
-x^-1*y + x^-1*y^-1 + x^-3*y^-1
```

```
>>> from sage.all import *
>>> L = Link([[Integer(1),Integer(4),Integer(2),Integer(3)],[Integer(4),
↪Integer(1),Integer(3),Integer(2)]]])
>>> L.homfly_polynomial('x', 'y') #_
↪needs sage.libs.homfly
-x^-1*y + x^-1*y^-1 + x^-3*y^-1
```

Another version of the Hopf link where the orientation has been changed. Therefore we substitute  $x \mapsto L^{-1}$  and  $y \mapsto M$ :

```
sage: L = Link([[1,3,2,4],[4,2,3,1]])
sage: L.homfly_polynomial() #_
↪needs sage.libs.homfly
L^3*M^-1 - L*M + L*M^-1
sage: L = Link([[1,3,2,4],[4,2,3,1]])
sage: L.homfly_polynomial(normalization='az') #_
↪needs sage.libs.homfly
a^3*z^-1 - a*z - a*z^-1
```

```
>>> from sage.all import *
>>> L = Link([[Integer(1),Integer(3),Integer(2),Integer(4)],[Integer(4),
↪Integer(2),Integer(3),Integer(1)]]])
>>> L.homfly_polynomial() #_
↪needs sage.libs.homfly
L^3*M^-1 - L*M + L*M^-1
>>> L = Link([[Integer(1),Integer(3),Integer(2),Integer(4)],[Integer(4),
↪Integer(2),Integer(3),Integer(1)]]])
>>> L.homfly_polynomial(normalization='az') #_
↪needs sage.libs.homfly
a^3*z^-1 - a*z - a*z^-1
```

The figure-eight knot:

```
sage: L = Link([[2,5,4,1],[5,3,7,6],[6,9,1,4],[9,7,3,2]])
sage: L.homfly_polynomial() #_
↪needs sage.libs.homfly
-L^2 + M^2 - 1 - L^-2
sage: L.homfly_polynomial('a', 'z', 'az') #_
↪needs sage.libs.homfly
a^2 - z^2 - 1 + a^-2
```

```
>>> from sage.all import *
>>> L = Link([[Integer(2),Integer(5),Integer(4),Integer(1)],[Integer(5),
↪Integer(3),Integer(7),Integer(6)],[Integer(6),Integer(9),Integer(1),
↪Integer(4)],[Integer(9),Integer(7),Integer(3),Integer(2)]]])
>>> L.homfly_polynomial() #_
↪needs sage.libs.homfly
```

(continues on next page)

(continued from previous page)

```

-L^2 + M^2 - 1 - L^-2
>>> L.homfly_polynomial('a', 'z', 'az') #_
↳needs sage.libs.homfly
a^2 - z^2 - 1 + a^-2

```

The “monster” unknot:

```

sage: L = Link([[3,1,2,4], [8,9,1,7], [5,6,7,3], [4,18,6,5],
.....:         [17,19,8,18], [9,10,11,14], [10,12,13,11],
.....:         [12,19,15,13], [20,16,14,15], [16,20,17,2]])
sage: L.homfly_polynomial() #_
↳needs sage.libs.homfly
1

```

```

>>> from sage.all import *
>>> L = Link([[Integer(3),Integer(1),Integer(2),Integer(4)], [Integer(8),
↳Integer(9),Integer(1),Integer(7)], [Integer(5),Integer(6),Integer(7),
↳Integer(3)], [Integer(4),Integer(18),Integer(6),Integer(5)],
...           [Integer(17),Integer(19),Integer(8),Integer(18)], [Integer(9),
↳Integer(10),Integer(11),Integer(14)], [Integer(10),Integer(12),Integer(13),
↳Integer(11)],
...           [Integer(12),Integer(19),Integer(15),Integer(13)], [Integer(20),
↳Integer(16),Integer(14),Integer(15)], [Integer(16),Integer(20),Integer(17),
↳Integer(2)]]])
>>> L.homfly_polynomial() #_
↳needs sage.libs.homfly
1

```

Comparison with KnotInfo:

```

sage: # needs sage.libs.homfly
sage: KI = K.get_knotinfo(mirror_version=False); KI
<KnotInfo.K5_1: '5_1'>
sage: K.homfly_polynomial(normalization='vz') == KI.homfly_polynomial()
True

```

```

>>> from sage.all import *
>>> # needs sage.libs.homfly
>>> KI = K.get_knotinfo(mirror_version=False); KI
<KnotInfo.K5_1: '5_1'>
>>> K.homfly_polynomial(normalization='vz') == KI.homfly_polynomial()
True

```

The knot  $9_6$ :

```

sage: # needs sage.libs.homfly
sage: B = BraidGroup(3)
sage: K = Knot(B([-1,-1,-1,-1,-1,-1,-2,1,-2,-2]))
sage: K.homfly_polynomial()
L^10*M^4 - L^8*M^6 - 3*L^10*M^2 + 4*L^8*M^4 + L^6*M^6 + L^10
- 3*L^8*M^2 - 5*L^6*M^4 - L^8 + 7*L^6*M^2 - 3*L^6
sage: K.homfly_polynomial('a', 'z', normalization='az')

```

(continues on next page)

(continued from previous page)

$$-a^{10}z^4 + a^8z^6 - 3a^{10}z^2 + 4a^8z^4 + a^6z^6 - a^{10} + 3a^8z^2 + 5a^6z^4 - a^8 + 7a^6z^2 + 3a^6$$

```
>>> from sage.all import *
>>> # needs sage.libs.homfly
>>> B = BraidGroup(Integer(3))
>>> K = Knot(B([-Integer(1), -Integer(1), -Integer(1), -Integer(1), -Integer(1), -Integer(1), -Integer(2), Integer(1), -Integer(2), -Integer(2)]))
>>> K.homfly_polynomial()
L^10*M^4 - L^8*M^6 - 3*L^10*M^2 + 4*L^8*M^4 + L^6*M^6 + L^10
- 3*L^8*M^2 - 5*L^6*M^4 - L^8 + 7*L^6*M^2 - 3*L^6
>>> K.homfly_polynomial('a', 'z', normalization='az')
-a^10*z^4 + a^8*z^6 - 3*a^10*z^2 + 4*a^8*z^4 + a^6*z^6 - a^10
+ 3*a^8*z^2 + 5*a^6*z^4 - a^8 + 7*a^6*z^2 + 3*a^6
```

## REFERENCES:

- [Wikipedia article HOMFLY\\_polynomial](#)
- <http://mathworld.wolfram.com/HOMFLYPolynomial.html>

**is\_alternating()**

Return whether the given knot diagram is alternating.

Alternating diagram implies every overcross is followed by an undercross or the vice-versa.

We look at the Gauss code if the sign is alternating, `True` is returned else the knot is not alternating `False` is returned.

**Warning**

This does not check if a knot admits an alternating diagram or not. Thus, this term is used differently than in some of the literature, such as in Hoste-Thistlethwaite table.

**Note**

Links with more than one component are considered to not be alternating (knots) even when such a diagram exists.

## EXAMPLES:

```
sage: B = BraidGroup(4)
sage: L = Link(B([-1, -1, -1, -1]))
sage: L.is_alternating()
False
sage: L = Link(B([1, -2, -1, 2]))
sage: L.is_alternating()
False
sage: L = Link(B([-1, 3, 1, 3, 2]))
sage: L.is_alternating()
False
sage: L = Link(B([1]*16 + [2, 1, 2, 1, 2, 2, 2, 2, 2, 2, 2, 1, 2, 1, 2, -1, 2, -2]))
```

(continues on next page)

(continued from previous page)

```
sage: L.is_alternating()
False
sage: L = Link(B([-1, 2, -1, 2]))
sage: L.is_alternating()
True
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> L = Link(B([-Integer(1), -Integer(1), -Integer(1), -Integer(1)]))
>>> L.is_alternating()
False
>>> L = Link(B([Integer(1), -Integer(2), -Integer(1), Integer(2)]))
>>> L.is_alternating()
False
>>> L = Link(B([-Integer(1), Integer(3), Integer(1), Integer(3), Integer(2)]))
>>> L.is_alternating()
False
>>> L = Link(B([Integer(1)]*Integer(16) + [Integer(2), Integer(1), Integer(2),
↪ Integer(1), Integer(2), Integer(2), Integer(2), Integer(2), Integer(2),
↪ Integer(2), Integer(2), Integer(1), Integer(2), Integer(1), Integer(2), -
↪ Integer(1), Integer(2), -Integer(2)]))
>>> L.is_alternating()
False
>>> L = Link(B([-Integer(1), Integer(2), -Integer(1), Integer(2)]))
>>> L.is_alternating()
True
```

We give the  $5_2$  knot with an alternating diagram and a non-alternating diagram:

```
sage: K5_2 = Link([[1, 4, 2, 5], [3, 8, 4, 9], [5, 10, 6, 1],
.....: [7, 2, 8, 3], [9, 6, 10, 7]])
sage: K5_2.is_alternating()
True

sage: K5_2b = Link(K5_2.braid())
sage: K5_2b.is_alternating()
False
```

```
>>> from sage.all import *
>>> K5_2 = Link([Integer(1), Integer(4), Integer(2), Integer(5)],
↪ [Integer(3), Integer(8), Integer(4), Integer(9)], [Integer(5), Integer(10),
↪ Integer(6), Integer(1)],
... [Integer(7), Integer(2), Integer(8), Integer(3)],
↪ [Integer(9), Integer(6), Integer(10), Integer(7)])
>>> K5_2.is_alternating()
True

>>> K5_2b = Link(K5_2.braid())
>>> K5_2b.is_alternating()
False
```

**is\_colorable** ( $n=None$ )

Return whether the link is  $n$ -colorable.

A link is  $n$ -colorable if its arcs can be painted with  $n$  colours, labeled from 0 to  $n - 1$ , in such a way that at any crossing, the average of the indices of the undercrossings equals twice the index of the overcrossing.

INPUT:

- $n$  – the number of colors to consider (if omitted the value of the determinant of `self` will be taken)

EXAMPLES:

We show that the trefoil knot is 3-colorable:

```
sage: K = Link([[1, -2, 3, -1, 2, -3]], [1, 1, 1])
sage: K.is_colorable(3)
↳needs sage.libs.pari sage.modules
True
```

```
>>> from sage.all import *
>>> K = Link([[Integer(1), -Integer(2), Integer(3), -Integer(1), Integer(2),
↳-Integer(3)], [Integer(1), Integer(1), Integer(1)]])
>>> K.is_colorable(Integer(3))
↳ # needs sage.libs.pari sage.modules
True
```

But the figure eight knot is not:

```
sage: K8 = Link([[1, -2, 4, -3, 2, -1, 3, -4]], [1, 1, -1, -1])
sage: K8.is_colorable(3)
↳needs sage.libs.pari sage.modules
False
```

```
>>> from sage.all import *
>>> K8 = Link([[Integer(1), -Integer(2), Integer(4), -Integer(3), Integer(2),
↳-Integer(1), Integer(3), -Integer(4)], [Integer(1), Integer(1), -
↳Integer(1), -Integer(1)]])
>>> K8.is_colorable(Integer(3))
↳ # needs sage.libs.pari sage.modules
False
```

But it is colorable with respect to the value of its determinant:

```
sage: K8.determinant()
5
sage: K8.is_colorable()
True
```

```
>>> from sage.all import *
>>> K8.determinant()
5
>>> K8.is_colorable()
True
```

An examples with non prime determinant:

```
sage: K = Knots().from_table(6, 1)
sage: K.determinant()
9
sage: K.is_colorable()
True
```

```
>>> from sage.all import *
>>> K = Knots().from_table(Integer(6), Integer(1))
>>> K.determinant()
9
>>> K.is_colorable()
True
```

## REFERENCES:

- [Wikipedia article Fox\\_n-coloring](#)
- Chapter 3 of [Liv1993]

## See also

*colorings()* and *coloring\_maps()*

**is\_isotopic**(*other*)

Check whether *self* is isotopic to *other*.

## INPUT:

- *other* – another instance of *Link*

## EXAMPLES:

```
sage: l1 = Link([[2, 9, 3, 10], [4, 13, 5, 14], [6, 11, 7, 12],
....:          [8, 1, 9, 2], [10, 7, 11, 8], [12, 5, 13, 6],
....:          [14, 3, 1, 4]])
sage: l2 = Link([[1, 8, 2, 9], [9, 2, 10, 3], [3, 14, 4, 1],
....:          [13, 4, 14, 5], [5, 12, 6, 13], [11, 6, 12, 7],
....:          [7, 10, 8, 11]])
sage: l1.is_isotopic(l2)
True

sage: l3 = l2.mirror_image()
sage: l1.is_isotopic(l3)
False

sage: # optional - database_knotinfo
sage: L = KnotInfo.L7a7_0_0
sage: L.series(oriented=True).inject()
Defining L7a7
sage: L == L7a7(0)
True
sage: l = L.link()
sage: l.is_isotopic(L7a7(1).link())
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```

...
NotImplementedError: comparison not possible!
sage: l1.is_isotopic(L7a7(2).link())
True
sage: l1.is_isotopic(L7a7(3).link())
False

```

```

>>> from sage.all import *
>>> l1 = Link([[Integer(2), Integer(9), Integer(3), Integer(10)], [Integer(4),
↳ Integer(13), Integer(5), Integer(14)], [Integer(6), Integer(11),
↳ Integer(7), Integer(12)],
...           [Integer(8), Integer(1), Integer(9), Integer(2)], [Integer(10),
↳ Integer(7), Integer(11), Integer(8)], [Integer(12), Integer(5),
↳ Integer(13), Integer(6)],
...           [Integer(14), Integer(3), Integer(1), Integer(4)]])
>>> l2 = Link([[Integer(1), Integer(8), Integer(2), Integer(9)], [Integer(9),
↳ Integer(2), Integer(10), Integer(3)], [Integer(3), Integer(14), Integer(4),
↳ Integer(1)],
...           [Integer(13), Integer(4), Integer(14), Integer(5)],
↳ [Integer(5), Integer(12), Integer(6), Integer(13)], [Integer(11),
↳ Integer(6), Integer(12), Integer(7)],
...           [Integer(7), Integer(10), Integer(8), Integer(11)]])
>>> l1.is_isotopic(l2)
True

>>> l3 = l2.mirror_image()
>>> l1.is_isotopic(l3)
False

>>> # optional - database_knotinfo
>>> L = KnotInfo.L7a7_0_0
>>> L.series(oriented=True).inject()
Defining L7a7
>>> L == L7a7(Integer(0))
True
>>> l = L.link()
>>> l.is_isotopic(L7a7(Integer(1)).link())
Traceback (most recent call last):
...
NotImplementedError: comparison not possible!
>>> l.is_isotopic(L7a7(Integer(2)).link())
True
>>> l.is_isotopic(L7a7(Integer(3)).link())
False

```

Using verbosity:

```

sage: set_verbosity(1)
sage: l1.is_isotopic(l2)
verbose 1 (... link.py, is_isotopic) identified by KnotInfo (KnotInfo.K7_2,
↳ SymmetryMutant.mirror_image)
True

```

(continues on next page)

(continued from previous page)

```
sage: l1.is_isotopic(l3)
verbose 1 (... link.py, is_isotopic) different Homfly-PT polynomials
False
sage: set_verbose(0)
```

```
>>> from sage.all import *
>>> set_verbose(Integer(1))
>>> l1.is_isotopic(l2)
verbose 1 (... link.py, is_isotopic) identified by KnotInfo (KnotInfo.K7_2, ↪
↪SymmetryMutant.mirror_image)
True
>>> l1.is_isotopic(l3)
verbose 1 (... link.py, is_isotopic) different Homfly-PT polynomials
False
>>> set_verbose(Integer(0))
```

**is\_knot()**

Return True if self is a knot.

Every knot is a link but the converse is not true.

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: L = Link(B([1, 3, 1, -3]))
sage: L.is_knot()
False
sage: B = BraidGroup(8)
sage: L = Link(B([1, 2, 3, 4, 5, 6]))
sage: L.is_knot()
True
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> L = Link(B([Integer(1), Integer(3), Integer(1), -Integer(3)]))
>>> L.is_knot()
False
>>> B = BraidGroup(Integer(8))
>>> L = Link(B([Integer(1), Integer(2), Integer(3), Integer(4), Integer(5), ↪
↪Integer(6)]))
>>> L.is_knot()
True
```

**jones\_polynomial** (*variab=None, skein\_normalization=False, algorithm='jonesrep'*)

Return the Jones polynomial of self.

The normalization is so that the unknot has Jones polynomial 1. If *skein\_normalization* is True, the variable of the result is replaced by a itself to the power of 4, so that the result agrees with the conventions of [Lic1997] (which in particular differs slightly from the conventions used otherwise in this class), had one used the conventional Kauffman bracket variable notation directly.

If *variab* is None return a polynomial in the variable *A* or *t*, depending on the value *skein\_normalization*. In particular, if *skein\_normalization* is False, return the result in terms of the variable *t*, also used in [Lic1997].

## ALGORITHM:

The calculation goes through one of two possible algorithms, depending on the value of `algorithm`. Possible values are `'jonesrep'` which uses the Jones representation of a braid representation of `self` to compute the polynomial of the trace closure of the braid, and `statesum` which recursively computes the Kauffman bracket of `self`. Depending on how the link is given, there might be significant time gains in using one over the other. When the trace closure of the braid is `self`, the algorithms give the same result.

## INPUT:

- `variab` – variable (default: `None`); the variable in the resulting polynomial; if unspecified, use either a default variable in  $\mathbb{Z}[A, A^{-1}]$  or the variable  $t$  in the symbolic ring
- `skein_normalization` – boolean (default: `False`); determines the variable of the resulting polynomial
- `algorithm` – string (default: `'jonesrep'`); algorithm to use and can be one of the following:
  - `'jonesrep'` – use the Jones representation of the braid representation
  - `'statesum'` – recursively computes the Kauffman bracket

## OUTPUT:

If `skein_normalization` is `False`, this returns an element in the symbolic ring as the Jones polynomial of the link might have fractional powers when the link is not a knot. Otherwise the result is a Laurent polynomial in `variab`.

## EXAMPLES:

The unknot:

```
sage: B = BraidGroup(9)
sage: b = B([1, 2, 3, 4, 5, 6, 7, 8])
sage: Link(b).jones_polynomial()
1
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(9))
>>> b = B([Integer(1), Integer(2), Integer(3), Integer(4), Integer(5),
↪ Integer(6), Integer(7), Integer(8)])
>>> Link(b).jones_polynomial()
1
```

The “monster” unknot:

```
sage: L = Link([[3, 1, 2, 4], [8, 9, 1, 7], [5, 6, 7, 3], [4, 18, 6, 5],
....:          [17, 19, 8, 18], [9, 10, 11, 14], [10, 12, 13, 11],
....:          [12, 19, 15, 13], [20, 16, 14, 15], [16, 20, 17, 2]])
sage: L.jones_polynomial()
↪needs sage.symbolic
1
```

```
>>> from sage.all import *
>>> L = Link([[Integer(3), Integer(1), Integer(2), Integer(4)], [Integer(8),
↪ Integer(9), Integer(1), Integer(7)], [Integer(5), Integer(6), Integer(7),
↪ Integer(3)], [Integer(4), Integer(18), Integer(6), Integer(5)],
...          [Integer(17), Integer(19), Integer(8), Integer(18)], [Integer(9),
```

(continues on next page)

(continued from previous page)

```

↪Integer(10), Integer(11), Integer(14)], [Integer(10), Integer(12), Integer(13),
↪Integer(11)],
...      [Integer(12), Integer(19), Integer(15), Integer(13)], [Integer(20),
↪Integer(16), Integer(14), Integer(15)], [Integer(16), Integer(20), Integer(17),
↪Integer(2)]]
>>> L.jones_polynomial()                                     #_
↪needs sage.symbolic
1

```

The Ochiai unknot:

```

sage: L = Link([[ [1, -2, -3, -8, -12, 13, -14, 15, -7, -1, 2, -4, 10, 11, -13, 12,
.....:             -11, -16, 4, 3, -5, 6, -9, 7, -15, 14, 16, -10, 8, 9, -6, 5]],
.....:          [-1, -1, 1, 1, 1, 1, -1, 1, 1, -1, 1, -1, -1, -1, -1]])
sage: L.jones_polynomial()                                   # long time #_
↪needs sage.symbolic
1

```

```

>>> from sage.all import *
>>> L = Link([ [Integer(1), -Integer(2), -Integer(3), -Integer(8), -Integer(12),
↪Integer(13), -Integer(14), Integer(15), -Integer(7), -Integer(1), Integer(2), -
↪Integer(4), Integer(10), Integer(11), -Integer(13), Integer(12),
...           -Integer(11), -Integer(16), Integer(4), Integer(3), -Integer(5),
↪Integer(6), -Integer(9), Integer(7), -Integer(15), Integer(14), Integer(16), -
↪Integer(10), Integer(8), Integer(9), -Integer(6), Integer(5)],
...          [-Integer(1), -Integer(1), Integer(1), Integer(1), Integer(1),
↪Integer(1), -Integer(1), Integer(1), Integer(1), -Integer(1), Integer(1), -
↪Integer(1), -Integer(1), -Integer(1), -Integer(1), -Integer(1)]]
>>> L.jones_polynomial()                                   # long time #_
↪needs sage.symbolic
1

```

Two unlinked unknots:

```

sage: B = BraidGroup(4)
sage: b = B([1, 3])
sage: Link(b).jones_polynomial()                             #_
↪needs sage.symbolic
-sqrt(t) - 1/sqrt(t)

```

```

>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> b = B([Integer(1), Integer(3)])
>>> Link(b).jones_polynomial()                               #_
↪needs sage.symbolic
-sqrt(t) - 1/sqrt(t)

```

The Hopf link:

```

sage: B = BraidGroup(2)
sage: b = B([-1, -1])
sage: Link(b).jones_polynomial()                             #_

```

(continues on next page)

(continued from previous page)

```
↪needs sage.symbolic
-1/sqrt(t) - 1/t^(5/2)
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(2))
>>> b = B([-Integer(1), -Integer(1)])
>>> Link(b).jones_polynomial()
↪needs sage.symbolic
-1/sqrt(t) - 1/t^(5/2) #_
```

Different representations of the trefoil and one of its mirror:

```
sage: B = BraidGroup(2)
sage: b = B([-1, -1, -1])
sage: Link(b).jones_polynomial(skein_normalization=True)
-A^-16 + A^-12 + A^-4
sage: Link(b).jones_polynomial()
1/t + 1/t^3 - 1/t^4
sage: B = BraidGroup(3)
sage: b = B([-1, -2, -1, -2])
sage: Link(b).jones_polynomial(skein_normalization=True)
-A^-16 + A^-12 + A^-4
sage: R.<x> = LaurentPolynomialRing(GF(2))
sage: Link(b).jones_polynomial(skein_normalization=True, variab=x)
x^-16 + x^-12 + x^-4
sage: B = BraidGroup(3)
sage: b = B([1, 2, 1, 2])
sage: Link(b).jones_polynomial(skein_normalization=True)
A^4 + A^12 - A^16
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(2))
>>> b = B([-Integer(1), -Integer(1), -Integer(1)])
>>> Link(b).jones_polynomial(skein_normalization=True)
-A^-16 + A^-12 + A^-4
>>> Link(b).jones_polynomial()
1/t + 1/t^3 - 1/t^4
>>> B = BraidGroup(Integer(3))
>>> b = B([-Integer(1), -Integer(2), -Integer(1), -Integer(2)])
>>> Link(b).jones_polynomial(skein_normalization=True)
-A^-16 + A^-12 + A^-4
>>> R = LaurentPolynomialRing(GF(Integer(2)), names=('x',)); (x,) = R._first_
↪ngens(1)
>>> Link(b).jones_polynomial(skein_normalization=True, variab=x)
x^-16 + x^-12 + x^-4
>>> B = BraidGroup(Integer(3))
>>> b = B([Integer(1), Integer(2), Integer(1), Integer(2)])
>>> Link(b).jones_polynomial(skein_normalization=True)
A^4 + A^12 - A^16
```

$K_{11n42}$  (the mirror of the “Kinoshita-Terasaka” knot) and  $K_{11n34}$  (the mirror of the “Conway” knot) in [KnotAtlas]:

```

sage: B = BraidGroup(4)
sage: K11n42 = Link(B([1, -2, 3, -2, 3, -2, -2, -1, 2, -3, -3, 2, 2]))
sage: K11n34 = Link(B([1, 1, 2, -3, 2, -3, 1, -2, -2, -3, -3]))
sage: bool(K11n42.jones_polynomial() == K11n34.jones_polynomial()) #_
↪needs sage.symbolic
True

```

```

>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> K11n42 = Link(B([Integer(1), -Integer(2), Integer(3), -Integer(2),
↪Integer(3), -Integer(2), -Integer(2), -Integer(1), Integer(2), -Integer(3),
↪Integer(3), Integer(2), Integer(2)]))
>>> K11n34 = Link(B([Integer(1), Integer(1), Integer(2), -Integer(3),
↪Integer(2), -Integer(3), Integer(1), -Integer(2), -Integer(2), -Integer(3),
↪Integer(3)]))
>>> bool(K11n42.jones_polynomial() == K11n34.jones_polynomial()) #_
↪needs sage.symbolic
True

```

The two algorithms for computation give the same result when the trace closure of the braid representation is the link itself:

```

sage: # needs sage.symbolic
sage: L = Link([[[-1, 2, -3, 4, 5, 1, -2, 6, 7, 3, -4, -7, -6, -5]],
...:          [-1, -1, -1, -1, 1, -1, 1]])
sage: jonesrep = L.jones_polynomial(algorithm='jonesrep')
sage: statesum = L.jones_polynomial(algorithm='statesum')
sage: bool(jonesrep == statesum)
True

```

```

>>> from sage.all import *
>>> # needs sage.symbolic
>>> L = Link([[-Integer(1), Integer(2), -Integer(3), Integer(4), Integer(5),
↪Integer(1), -Integer(2), Integer(6), Integer(7), Integer(3), -Integer(4), -
↪Integer(7), -Integer(6), -Integer(5)]],
...          [-Integer(1), -Integer(1), -Integer(1), -Integer(1), Integer(1),
↪-Integer(1), Integer(1)])
>>> jonesrep = L.jones_polynomial(algorithm='jonesrep')
>>> statesum = L.jones_polynomial(algorithm='statesum')
>>> bool(jonesrep == statesum)
True

```

When we have thrown away unknots so that the trace closure of the braid is not necessarily the link itself, this is only true up to a power of the Jones polynomial of the unknot:

```

sage: B = BraidGroup(3)
sage: b = B([1])
sage: L = Link(b)
sage: b.components_in_closure()
2
sage: L.number_of_components()
1
sage: b.jones_polynomial()

```

(continues on next page)

(continued from previous page)

```

↪needs sage.symbolic
-sqrt(t) - 1/sqrt(t)
sage: L.jones_polynomial()                                     #_
↪needs sage.symbolic
1
sage: L.jones_polynomial(algorithm='statesum')                #_
↪needs sage.symbolic
1

```

```

>>> from sage.all import *
>>> B = BraidGroup(Integer(3))
>>> b = B([Integer(1)])
>>> L = Link(b)
>>> b.components_in_closure()
2
>>> L.number_of_components()
1
>>> b.jones_polynomial()                                     #_
↪needs sage.symbolic
-sqrt(t) - 1/sqrt(t)
>>> L.jones_polynomial()                                     #_
↪needs sage.symbolic
1
>>> L.jones_polynomial(algorithm='statesum')                #_
↪needs sage.symbolic
1

```

**khovanov\_homology** (*ring=Integer Ring, height=None, degree=None*)

Return the Khovanov homology of the link.

INPUT:

- *ring* – (default:  $\mathbb{Z}\mathbb{Z}$ ) the coefficient ring
- *height* – the height of the homology to compute, if not specified, all the heights are computed
- *degree* – the degree of the homology to compute, if not specified, all the degrees are computed

OUTPUT:

The Khovanov homology of the Link. It is given as a dictionary whose keys are the different heights. For each height, the homology is given as another dictionary whose keys are the degrees.

EXAMPLES:

```

sage: K = Link([[[1, -2, 3, -1, 2, -3]], [-1, -1, -1]])
sage: K.khovanov_homology()                                     #_
↪needs sage.modules
{-9: {-3: Z},
 -7: {-3: 0, -2: C2},
 -5: {-3: 0, -2: Z, -1: 0, 0: 0},
 -3: {-3: 0, -2: 0, -1: 0, 0: Z},
 -1: {0: Z}}

```

```

>>> from sage.all import *
>>> K = Link([[Integer(1), -Integer(2), Integer(3), -Integer(1), Integer(2),
↳ -Integer(3)], [-Integer(1), -Integer(1), -Integer(1)]])
>>> K.khovanov_homology() #
↳ needs sage.modules
{-9: {-3: Z},
 -7: {-3: 0, -2: C2},
 -5: {-3: 0, -2: Z, -1: 0, 0: 0},
 -3: {-3: 0, -2: 0, -1: 0, 0: Z},
 -1: {0: Z}}

```

The figure eight knot:

```

sage: L = Link([[1, 6, 2, 7], [5, 2, 6, 3], [3, 1, 4, 8], [7, 5, 8, 4]])
sage: L.khovanov_homology(height=-1) #
↳ needs sage.modules
{-1: {-2: 0, -1: Z, 0: Z, 1: 0, 2: 0}}

```

```

>>> from sage.all import *
>>> L = Link([[Integer(1), Integer(6), Integer(2), Integer(7)], [Integer(5),
↳ Integer(2), Integer(6), Integer(3)], [Integer(3), Integer(1), Integer(4),
↳ Integer(8)], [Integer(7), Integer(5), Integer(8), Integer(4)]])
>>> L.khovanov_homology(height=-Integer(1)) #
↳ needs sage.modules
{-1: {-2: 0, -1: Z, 0: Z, 1: 0, 2: 0}}

```

The Hopf link:

```

sage: # needs sage.modules
sage: B = BraidGroup(2)
sage: b = B([1, 1])
sage: K = Link(b)
sage: K.khovanov_homology(degree=2)
{2: {2: 0}, 4: {2: Z}, 6: {2: Z}}

```

```

>>> from sage.all import *
>>> # needs sage.modules
>>> B = BraidGroup(Integer(2))
>>> b = B([Integer(1), Integer(1)])
>>> K = Link(b)
>>> K.khovanov_homology(degree=Integer(2))
{2: {2: 0}, 4: {2: Z}, 6: {2: Z}}

```

**khovanov\_polynomial**(var1='q', var2='t', base\_ring=Integer Ring)

Return the Khovanov polynomial of `self`.

This is the Poincaré polynomial of the Khovanov homology.

INPUT:

- `var1` – (default: 'q') the first variable. Its exponents give the (torsion free) rank of the height of Khovanov homology
- `var2` – (default: 't') the second variable. Its exponents give the (torsion free) rank of the degree of Khovanov homology

- `base_ring` – (default: `ZZ`) the ring of the polynomial's coefficients

OUTPUT:

A two variate Laurent Polynomial over the `base_ring`, more precisely an instance of `LaurentPolynomial`.

EXAMPLES:

```
sage: K = Link([[1, -2, 3, -1, 2, -3]], [-1, -1, -1])
sage: K.khovanov_polynomial()                                     #_
↪needs sage.modules
q^-1 + q^-3 + q^-5*t^-2 + q^-9*t^-3
sage: K.khovanov_polynomial(base_ring=GF(2))                     #_
↪needs sage.modules
q^-1 + q^-3 + q^-5*t^-2 + q^-7*t^-2 + q^-9*t^-3
```

```
>>> from sage.all import *
>>> K = Link([[[Integer(1), -Integer(2), Integer(3), -Integer(1), Integer(2),
↪-Integer(3)]], [-Integer(1), -Integer(1), -Integer(1)]]])
>>> K.khovanov_polynomial()                                     #_
↪needs sage.modules
q^-1 + q^-3 + q^-5*t^-2 + q^-9*t^-3
>>> K.khovanov_polynomial(base_ring=GF(Integer(2)))             _
↪ # needs sage.modules
q^-1 + q^-3 + q^-5*t^-2 + q^-7*t^-2 + q^-9*t^-3
```

The figure eight knot:

```
sage: L = Link([[1, 6, 2, 7], [5, 2, 6, 3], [3, 1, 4, 8], [7, 5, 8, 4]])
sage: L.khovanov_polynomial(var1='p')                             #_
↪needs sage.modules
p^5*t^2 + p*t + p + p^-1 + p^-1*t^-1 + p^-5*t^-2
sage: L.khovanov_polynomial(var1='p', var2='s', base_ring=GF(4)) #_
↪needs sage.modules sage.rings.finite_rings
p^5*s^2 + p^3*s^2 + p*s + p + p^-1 + p^-1*s^-1 + p^-3*s^-1 + p^-5*s^-2
```

```
>>> from sage.all import *
>>> L = Link([[[Integer(1), Integer(6), Integer(2), Integer(7)], [Integer(5),
↪Integer(2), Integer(6), Integer(3)], [Integer(3), Integer(1), Integer(4),
↪Integer(8)], [Integer(7), Integer(5), Integer(8), Integer(4)]]])
>>> L.khovanov_polynomial(var1='p')                             #_
↪needs sage.modules
p^5*t^2 + p*t + p + p^-1 + p^-1*t^-1 + p^-5*t^-2
>>> L.khovanov_polynomial(var1='p', var2='s', base_ring=GF(Integer(4))) _
↪ # needs sage.modules sage.rings.finite_rings
p^5*s^2 + p^3*s^2 + p*s + p + p^-1 + p^-1*s^-1 + p^-3*s^-1 + p^-5*s^-2
```

The Hopf link:

```
sage: B = BraidGroup(2)
sage: b = B([1, 1])
sage: K = Link(b)
sage: K.khovanov_polynomial()                                     #_
```

(continues on next page)

(continued from previous page)

```
↪needs sage.modules
q^6*t^2 + q^4*t^2 + q^2 + 1
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(2))
>>> b = B([Integer(1), Integer(1)])
>>> K = Link(b)
>>> K.khovanov_polynomial() #↵
↪needs sage.modules
q^6*t^2 + q^4*t^2 + q^2 + 1
```

**See also**

`khovanov_homology()`

**links\_gould\_polynomial**(varnames='t0, t1')

Return the Links-Gould polynomial of `self`. See [MW2012], section 3 and references given there. See also the docstring of `links_gould_polynomial()`.

INPUT:

- `varnames` – string (default: 't0, t1')

OUTPUT: a Laurent polynomial in the given variable names

EXAMPLES:

```
sage: Hopf = Link([[1, 3, 2, 4], [4, 2, 3, 1]])
sage: Hopf.links_gould_polynomial()
-1 + t1^-1 + t0^-1 - t0^-1*t1^-1
```

```
>>> from sage.all import *
>>> Hopf = Link([[Integer(1), Integer(3), Integer(2), Integer(4)],↵
↪[Integer(4), Integer(2), Integer(3), Integer(1)]])
>>> Hopf.links_gould_polynomial()
-1 + t1^-1 + t0^-1 - t0^-1*t1^-1
```

**mirror\_image()**

Return the mirror image of `self`.

EXAMPLES:

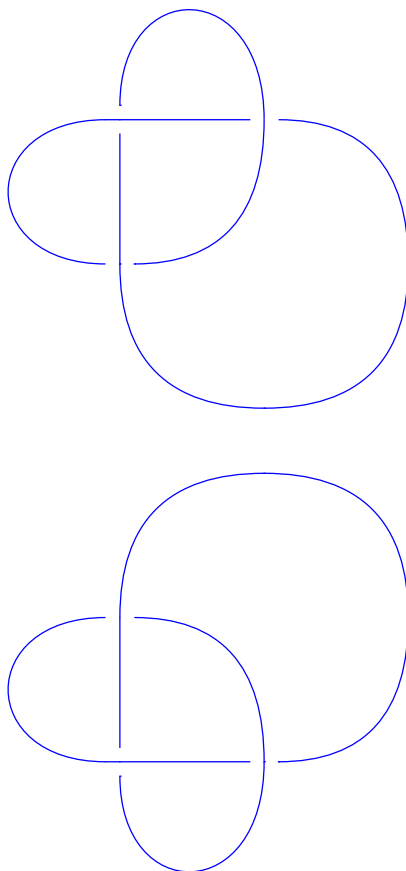
```
sage: g = BraidGroup(2).gen(0)
sage: K = Link(g^3)
sage: K2 = K.mirror_image(); K2
Link with 1 component represented by 3 crossings
sage: K2.braid()
s^-3
```

```
>>> from sage.all import *
>>> g = BraidGroup(Integer(2)).gen(Integer(0))
>>> K = Link(g**Integer(3))
```

(continues on next page)

(continued from previous page)

```
>>> K2 = K.mirror_image(); K2
Link with 1 component represented by 3 crossings
>>> K2.braid()
s^-3
```



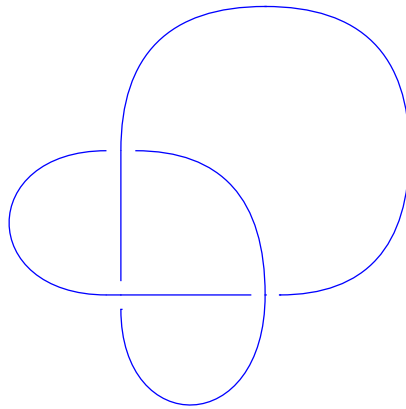
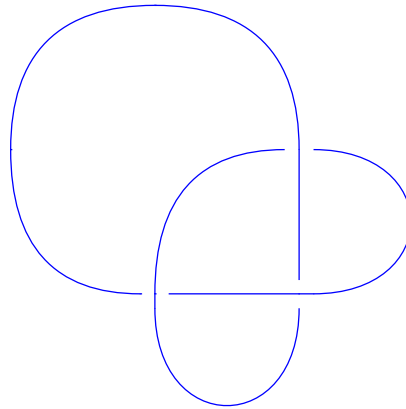
```
sage: K = Knot([[[1, -2, 3, -1, 2, -3]], [1, 1, 1]])
sage: K2 = K.mirror_image(); K2
Knot represented by 3 crossings
sage: K.pd_code()
[[4, 2, 5, 1], [2, 6, 3, 5], [6, 4, 1, 3]]
sage: K2.pd_code()
[[4, 1, 5, 2], [2, 5, 3, 6], [6, 3, 1, 4]]
```

```
>>> from sage.all import *
>>> K = Knot([[[Integer(1), -Integer(2), Integer(3), -Integer(1), Integer(2),
↪ -Integer(3)]], [Integer(1), Integer(1), Integer(1)]])
>>> K2 = K.mirror_image(); K2
Knot represented by 3 crossings
>>> K.pd_code()
[[4, 2, 5, 1], [2, 6, 3, 5], [6, 4, 1, 3]]
```

(continues on next page)

(continued from previous page)

```
>>> K2.pd_code()
[[4, 1, 5, 2], [2, 5, 3, 6], [6, 3, 1, 4]]
```

**number\_of\_components()**

Return the number of connected components of `self`.

OUTPUT: number of connected components

EXAMPLES:

```
sage: B = BraidGroup(4)
sage: L = Link(B([-1, 3, 1, 3]))
sage: L.number_of_components()
4
sage: B = BraidGroup(8)
sage: L = Link(B([-2, 4, 1, 6, 1, 4]))
sage: L.number_of_components()
5
sage: L = Link(B([1, 2, 1, 2]))
sage: L.number_of_components()
1
sage: L = Link(B.one())
```

(continues on next page)

(continued from previous page)

```
sage: L.number_of_components()
1
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> L = Link(B([-Integer(1), Integer(3), Integer(1), Integer(3)]))
>>> L.number_of_components()
4
>>> B = BraidGroup(Integer(8))
>>> L = Link(B([-Integer(2), Integer(4), Integer(1), Integer(6), Integer(1),
↳ Integer(4)]))
>>> L.number_of_components()
5
>>> L = Link(B([Integer(1), Integer(2), Integer(1), Integer(2)]))
>>> L.number_of_components()
1
>>> L = Link(B.one())
>>> L.number_of_components()
1
```

**omega\_signature(omega)**

Compute the  $\omega$ -signature of self.

INPUT:

- $\omega$  – a complex number of modulus 1; this is assumed to be coercible to  $\mathbb{Q}\bar{\mathbb{Q}}$

This is defined as the signature of the Hermitian matrix

$$(1 - \omega)V + (1 - \omega^{-1})V^t,$$

where  $V$  is the *Seifert matrix*, as explained on page 122 of [Liv1993].

According to [Con2018], this is also known as the Levine-Tristram signature, the equivariant signature or the Tristram-Levine signature.

**See also**

`signature()`, `seifert_matrix()`

**EXAMPLES:**

```
sage: # needs sage.modules sage.rings.number_field
sage: B = BraidGroup(4)
sage: K = Knot(B([1, 1, 1, 2, -1, 2, -3, 2, -3]))
sage: omega = QQbar.zeta(3)
sage: K.omega_signature(omega)
-2
```

```
>>> from sage.all import *
>>> # needs sage.modules sage.rings.number_field
>>> B = BraidGroup(Integer(4))
>>> K = Knot(B([Integer(1), Integer(1), Integer(1), Integer(2), -Integer(1),
```

(continues on next page)

(continued from previous page)

```

↪Integer(2),-Integer(3),Integer(2),-Integer(3)])
>>> omega = QQbar.zeta(Integer(3))
>>> K.omega_signature(omega)
-2

```

**orientation()**

Return the orientation of the crossings of the link diagram of `self`.

EXAMPLES:

```

sage: L = Link([[1, 2, 5, 4], [3, 7, 6, 5], [4, 6, 9, 8], [7, 11, 10, 9],
....:          [8, 10, 13, 1], [11, 3, 2, 13]])
sage: L.orientation()
[-1, 1, -1, 1, -1, 1]
sage: L = Link([[1, 6, 2, 7], [7, 2, 8, 3], [3, 10, 4, 11], [11, 4, 12, 5],
....:          [14, 6, 1, 5], [13, 8, 14, 9], [12, 10, 13, 9]])
sage: L.orientation()
[-1, -1, -1, -1, 1, -1, 1]
sage: L = Link([[1, 3, 3, 2], [2, 5, 5, 4], [4, 7, 7, 1]])
sage: L.orientation()
[-1, -1, -1]

```

```

>>> from sage.all import *
>>> L = Link([[Integer(1), Integer(2), Integer(5), Integer(4)], [Integer(3), ↪
↪Integer(7), Integer(6), Integer(5)], [Integer(4), Integer(6), Integer(9), ↪
↪Integer(8)], [Integer(7), Integer(11), Integer(10), Integer(9)],
...          [Integer(8), Integer(10), Integer(13), Integer(1)], ↪
↪[Integer(11), Integer(3), Integer(2), Integer(13)]])
>>> L.orientation()
[-1, 1, -1, 1, -1, 1]
>>> L = Link([[Integer(1), Integer(6), Integer(2), Integer(7)], [Integer(7), ↪
↪Integer(2), Integer(8), Integer(3)], [Integer(3), Integer(10), Integer(4), ↪
↪Integer(11)], [Integer(11), Integer(4), Integer(12), Integer(5)],
...          [Integer(14), Integer(6), Integer(1), Integer(5)], [Integer(13),
↪ Integer(8), Integer(14), Integer(9)], [Integer(12), Integer(10), ↪
↪Integer(13), Integer(9)]])
>>> L.orientation()
[-1, -1, -1, -1, 1, -1, 1]
>>> L = Link([[Integer(1), Integer(3), Integer(3), Integer(2)], [Integer(2), ↪
↪Integer(5), Integer(5), Integer(4)], [Integer(4), Integer(7), Integer(7), ↪
↪Integer(1)]])
>>> L.orientation()
[-1, -1, -1]

```

**oriented\_gauss\_code()**

Return the oriented Gauss code of `self`.

The oriented Gauss code has two parts:

- a. the Gauss code
- b. the orientation of each crossing

The following orientation was taken into consideration for construction of knots:

From the outgoing of the overcrossing if we move in the clockwise direction to reach the outgoing of the undercrossing then we label that crossing as  $-1$ .

From the outgoing of the overcrossing if we move in the anticlockwise direction to reach the outgoing of the undercrossing then we label that crossing as  $+1$ .

One more consideration we take in while constructing the orientation is the order of the orientation is same as the ordering of the crossings in the Gauss code.

#### Note

Convention: under is denoted by  $-1$ , and over by  $+1$  in the crossing info.

#### EXAMPLES:

```
sage: L = Link([[1, 10, 2, 11], [6, 3, 7, 2], [3, 9, 4, 12],
....:         [9, 6, 10, 5], [8, 4, 5, 1], [11, 7, 12, 8]])
sage: L.oriented_gauss_code()
[[-1, 2, -3, 5], [4, -2, 6, -5], [-4, 1, -6, 3]], [-1, 1, 1, 1, -1, -1]]
sage: L = Link([[1, 3, 2, 4], [6, 2, 3, 1], [7, 5, 8, 4], [5, 7, 6, 8]])
sage: L.oriented_gauss_code()
[[-1, 2], [-3, 4], [1, 3, -4, -2]], [-1, -1, 1, 1]]

sage: B = BraidGroup(8)
sage: b = B([1, 1, 1, 1, 1])
sage: L = Link(b)
sage: L.oriented_gauss_code()
[[[1, -2, 3, -4, 5, -1, 2, -3, 4, -5]], [1, 1, 1, 1, 1]]
```

```
>>> from sage.all import *
>>> L = Link([[Integer(1), Integer(10), Integer(2), Integer(11)], [Integer(6),
↪ Integer(3), Integer(7), Integer(2)], [Integer(3), Integer(9), Integer(4), ↪
↪ Integer(12)],
...         [Integer(9), Integer(6), Integer(10), Integer(5)], [Integer(8), ↪
↪ Integer(4), Integer(5), Integer(1)], [Integer(11), Integer(7), Integer(12), ↪
↪ Integer(8)]])
>>> L.oriented_gauss_code()
[[-1, 2, -3, 5], [4, -2, 6, -5], [-4, 1, -6, 3]], [-1, 1, 1, 1, -1, -1]]
>>> L = Link([[Integer(1), Integer(3), Integer(2), Integer(4)], [Integer(6), ↪
↪ Integer(2), Integer(3), Integer(1)], [Integer(7), Integer(5), Integer(8), ↪
↪ Integer(4)], [Integer(5), Integer(7), Integer(6), Integer(8)]])
>>> L.oriented_gauss_code()
[[-1, 2], [-3, 4], [1, 3, -4, -2]], [-1, -1, 1, 1]]

>>> B = BraidGroup(Integer(8))
>>> b = B([Integer(1), Integer(1), Integer(1), Integer(1), Integer(1)])
>>> L = Link(b)
>>> L.oriented_gauss_code()
[[[1, -2, 3, -4, 5, -1, 2, -3, 4, -5]], [1, 1, 1, 1, 1]]
```

#### pd\_code()

Return the planar diagram code of `self`.

The planar diagram is returned in the following format.

We construct the crossing by starting with the entering component of the undercrossing, move in the anti-clockwise direction (see the note below) and then generate the list. If the crossing is given by  $[a, b, c, d]$ , then we interpret this information as:

1.  $a$  is the entering component of the undercrossing;
2.  $b, d$  are the components of the overcrossing;
3.  $c$  is the leaving component of the undercrossing.

#### Note

Until version 10.0 the convention to read the PD code has been to list the components in clockwise direction. As of version 10.1 the convention has changed, since it was opposite to the usage in most other places.

Thus, if you use PD codes from former Sage releases with this version you should check for the correct mirror type.

#### EXAMPLES:

```
sage: L = Link([[1, -2, 3, -4, 2, -1, 4, -3]], [1, 1, -1, -1])
sage: L.pd_code()
[[6, 2, 7, 1], [2, 6, 3, 5], [8, 3, 1, 4], [4, 7, 5, 8]]

sage: B = BraidGroup(2)
sage: b = B([1, 1, 1, 1, 1])
sage: L = Link(b)
sage: L.pd_code()
[[2, 4, 3, 1], [4, 6, 5, 3], [6, 8, 7, 5], [8, 10, 9, 7], [10, 2, 1, 9]]
sage: L = Link([[2, -1], [1, -2]], [1, 1])
sage: L.pd_code()
[[2, 4, 1, 3], [4, 2, 3, 1]]
sage: L = Link([[1, 2, 3, 3], [2, 4, 5, 5], [4, 1, 7, 7]])
sage: L.pd_code()
[[1, 2, 3, 3], [2, 4, 5, 5], [4, 1, 7, 7]]
```

```
>>> from sage.all import *
>>> L = Link([[[Integer(1), -Integer(2), Integer(3), -Integer(4), Integer(2),
↪ -Integer(1), Integer(4), -Integer(3)], [Integer(1), Integer(1), -
↪ Integer(1), -Integer(1)]]])
>>> L.pd_code()
[[6, 2, 7, 1], [2, 6, 3, 5], [8, 3, 1, 4], [4, 7, 5, 8]]

>>> B = BraidGroup(Integer(2))
>>> b = B([Integer(1), Integer(1), Integer(1), Integer(1), Integer(1)])
>>> L = Link(b)
>>> L.pd_code()
[[2, 4, 3, 1], [4, 6, 5, 3], [6, 8, 7, 5], [8, 10, 9, 7], [10, 2, 1, 9]]
>>> L = Link([[[Integer(2), -Integer(1)], [Integer(1), -Integer(2)],
↪ [Integer(1), Integer(1)]]])
>>> L.pd_code()
[[2, 4, 1, 3], [4, 2, 3, 1]]
>>> L = Link([[[Integer(1), Integer(2), Integer(3), Integer(3)], [Integer(2),
↪ Integer(1), Integer(1), Integer(1), Integer(1), Integer(1)]]])
```

(continues on next page)

(continued from previous page)

```

↪Integer(4), Integer(5), Integer(5)], [Integer(4), Integer(1), Integer(7), ↪
↪Integer(7)])])
>>> L.pd_code()
[[1, 2, 3, 3], [2, 4, 5, 5], [4, 1, 7, 7]]

```

`plot(gap=0.1, component_gap=0.5, solver=None, color='blue', **kwargs)`  
 Plot self.

INPUT:

- `gap` – (default: 0.1) the size of the blank gap left for the crossings
- `component_gap` – (default: 0.5) the gap between isolated components
- `solver` – the linear solver to use, see `MixedIntegerLinearProgram`
- `color` – string (default: 'blue'); a color or a coloring (as returned by `colorings()`)

The usual keywords for plots can be used here too.

EXAMPLES:

We construct the simplest version of the unknot:

```

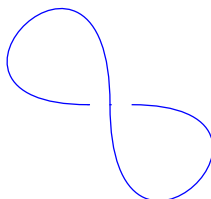
sage: L = Link([[2, 1, 1, 2]])
sage: L.plot()
↪needs sage.plot
Graphics object consisting of ... graphics primitives

```

```

>>> from sage.all import *
>>> L = Link([[Integer(2), Integer(1), Integer(1), Integer(2)]])
>>> L.plot()
↪needs sage.plot
Graphics object consisting of ... graphics primitives

```



We construct a more interesting example of the unknot:

```

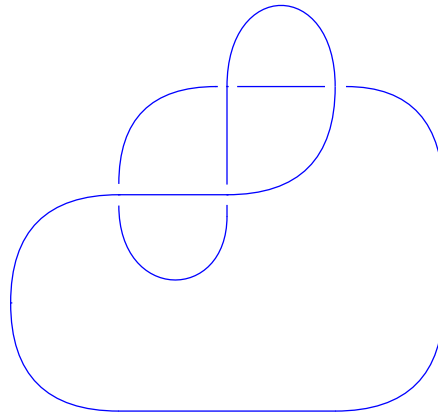
sage: L = Link([[2, 1, 4, 5], [3, 5, 6, 7], [4, 1, 9, 6], [9, 2, 3, 7]])
sage: L.plot()
↪needs sage.plot
Graphics object consisting of ... graphics primitives

```

```

>>> from sage.all import *
>>> L = Link([[Integer(2), Integer(1), Integer(4), Integer(5)], [Integer(3),
↳Integer(5), Integer(6), Integer(7)], [Integer(4), Integer(1), Integer(9),
↳Integer(6)], [Integer(9), Integer(2), Integer(3), Integer(7)]])
>>> L.plot() #_
↳needs sage.plot
Graphics object consisting of ... graphics primitives

```



The “monster” unknot:

```

sage: L = Link([[3,1,2,4], [8,9,1,7], [5,6,7,3], [4,18,6,5],
.....:          [17,19,8,18], [9,10,11,14], [10,12,13,11],
.....:          [12,19,15,13], [20,16,14,15], [16,20,17,2]])
sage: L.plot() #_
↳needs sage.plot
Graphics object consisting of ... graphics primitives

```

```

>>> from sage.all import *
>>> L = Link([[Integer(3),Integer(1),Integer(2),Integer(4)], [Integer(8),
↳Integer(9),Integer(1),Integer(7)], [Integer(5),Integer(6),Integer(7),
↳Integer(3)], [Integer(4),Integer(18),Integer(6),Integer(5)],
...          [Integer(17),Integer(19),Integer(8),Integer(18)], [Integer(9),
↳Integer(10),Integer(11),Integer(14)], [Integer(10),Integer(12),Integer(13),
↳Integer(11)],
...          [Integer(12),Integer(19),Integer(15),Integer(13)], [Integer(20),
↳Integer(16),Integer(14),Integer(15)], [Integer(16),Integer(20),Integer(17),
↳Integer(2)]])
>>> L.plot() #_
↳needs sage.plot
Graphics object consisting of ... graphics primitives

```

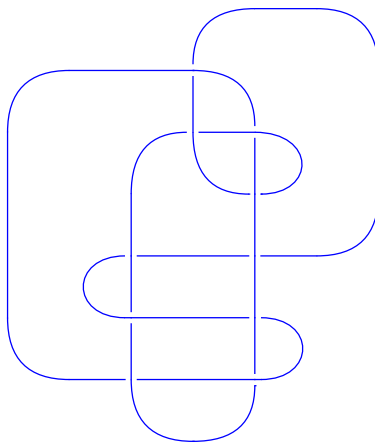
The Ochiai unknot:

```

sage: L = Link([[1,-2,-3,-8,-12,13,-14,15,-7,-1,2,-4,10,11,-13,12,
.....:          -11,-16,4,3,-5,6,-9,7,-15,14,16,-10,8,9,-6,5]],
.....:          [-1,-1,1,1,1,1,-1,1,1,-1,1,-1,-1,-1,-1,-1]])
sage: L.plot() #_

```

(continues on next page)



(continued from previous page)

↪ `needs sage.plot`

Graphics object consisting of ... graphics primitives

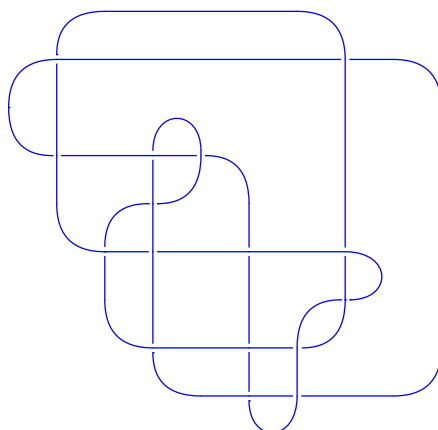
```
>>> from sage.all import *
>>> L = Link([[[Integer(1), -Integer(2), -Integer(3), -Integer(8), -Integer(12),
↪ Integer(13), -Integer(14), Integer(15), -Integer(7), -Integer(1), Integer(2), -
↪ Integer(4), Integer(10), Integer(11), -Integer(13), Integer(12),
... -Integer(11), -Integer(16), Integer(4), Integer(3), -Integer(5),
↪ Integer(6), -Integer(9), Integer(7), -Integer(15), Integer(14), Integer(16), -
↪ Integer(10), Integer(8), Integer(9), -Integer(6), Integer(5) ]],
... [-Integer(1), -Integer(1), Integer(1), Integer(1), Integer(1),
↪ Integer(1), -Integer(1), Integer(1), Integer(1), -Integer(1), Integer(1), -
↪ Integer(1), -Integer(1), -Integer(1), -Integer(1), -Integer(1) ]])
```

>>> `L.plot()`

# ↪

↪ `needs sage.plot`

Graphics object consisting of ... graphics primitives



One of the representations of the trefoil knot:

```
sage: L = Link([[1, 5, 2, 4], [5, 3, 6, 2], [3, 1, 4, 6]])
sage: L.plot()
```

# ↪

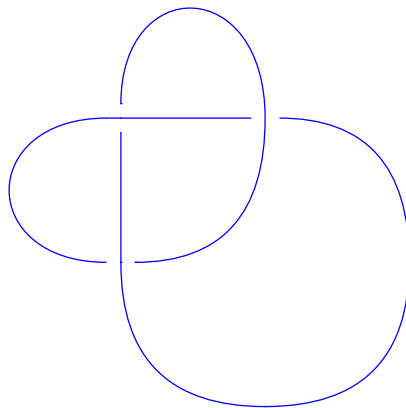
(continues on next page)

(continued from previous page)

`↪needs sage.plot`

Graphics object consisting of 14 graphics primitives

```
>>> from sage.all import *
>>> L = Link([[Integer(1), Integer(5), Integer(2), Integer(4)], [Integer(5), ↪
↪Integer(3), Integer(6), Integer(2)], [Integer(3), Integer(1), Integer(4), ↪
↪Integer(6)]]])
>>> L.plot() #↪
↪needs sage.plot
Graphics object consisting of 14 graphics primitives
```



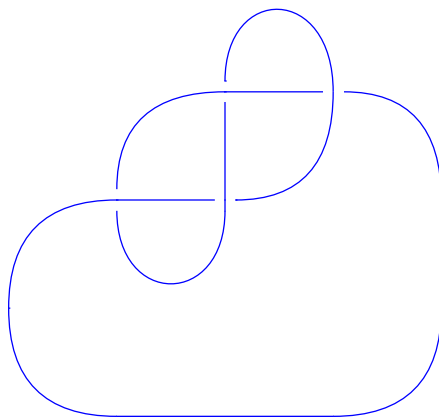
The figure-eight knot:

```
sage: L = Link([[2, 1, 4, 5], [5, 6, 7, 3], [6, 4, 1, 9], [9, 2, 3, 7]])
sage: L.plot() #↪
↪needs sage.plot
Graphics object consisting of ... graphics primitives
```

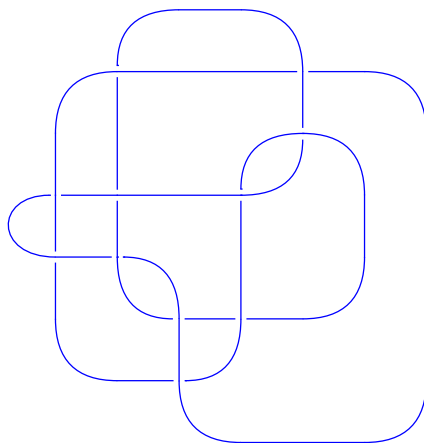
```
>>> from sage.all import *
>>> L = Link([[Integer(2), Integer(1), Integer(4), Integer(5)], [Integer(5), ↪
↪Integer(6), Integer(7), Integer(3)], [Integer(6), Integer(4), Integer(1), ↪
↪Integer(9)], [Integer(9), Integer(2), Integer(3), Integer(7)]]])
>>> L.plot() #↪
↪needs sage.plot
Graphics object consisting of ... graphics primitives
```

The knot  $K_{11n121}$  in [KnotAtlas]:

```
sage: L = Link([[4, 2, 5, 1], [10, 3, 11, 4], [5, 16, 6, 17], [7, 12, 8, 13],
.....:          [18, 9, 19, 10], [2, 11, 3, 12], [13, 20, 14, 21], [15, 6, 16, 7],
.....:          [22, 18, 1, 17], [8, 19, 9, 20], [21, 14, 22, 15]])
sage: L.plot() #↪
↪needs sage.plot
Graphics object consisting of ... graphics primitives
```



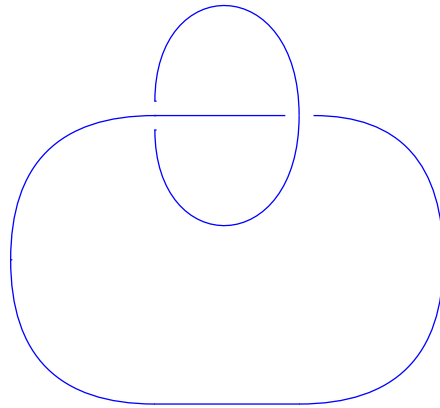
```
>>> from sage.all import *
>>> L = Link([[Integer(4), Integer(2), Integer(5), Integer(1)], [Integer(10),
↪ Integer(3), Integer(11), Integer(4)], [Integer(5), Integer(16), Integer(6),
↪ Integer(17)], [Integer(7), Integer(12), Integer(8), Integer(13)],
...      [Integer(18), Integer(9), Integer(19), Integer(10)], [Integer(2),
↪ Integer(11), Integer(3), Integer(12)], [Integer(13), Integer(20), Integer(14),
↪ Integer(21)], [Integer(15), Integer(6), Integer(16), Integer(7)],
...      [Integer(22), Integer(18), Integer(1), Integer(17)], [Integer(8),
↪ Integer(19), Integer(9), Integer(20)], [Integer(21), Integer(14), Integer(22),
↪ Integer(15)]])
>>> L.plot() #_
↪needs sage.plot
Graphics object consisting of ... graphics primitives
```



One of the representations of the Hopf link:

```
sage: L = Link([[1, 4, 2, 3], [4, 1, 3, 2]])
sage: L.plot() #_
↪needs sage.plot
Graphics object consisting of ... graphics primitives
```

```
>>> from sage.all import *
>>> L = Link([[Integer(1), Integer(4), Integer(2), Integer(3)], [Integer(4),
↳ Integer(1), Integer(3), Integer(2)]]
>>> L.plot() #
↳needs sage.plot
Graphics object consisting of ... graphics primitives
```



Plotting links with multiple isolated components:

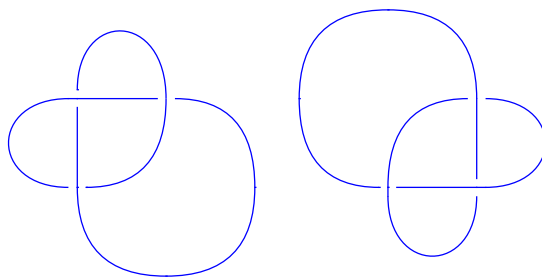
```
sage: L = Link([[[-1, 2, -3, 1, -2, 3], [4, -5, 6, -4, 5, -6]],
....:          [1, 1, 1, 1, 1, 1]])
sage: L.plot() #
↳needs sage.plot
Graphics object consisting of ... graphics primitives
```

```
>>> from sage.all import *
>>> L = Link([[[-Integer(1), Integer(2), -Integer(3), Integer(1), -Integer(2),
↳ Integer(3)], [Integer(4), -Integer(5), Integer(6), -Integer(4), Integer(5),
↳ -Integer(6)]]],
...          [Integer(1), Integer(1), Integer(1), Integer(1), Integer(1),
↳ Integer(1)]]])
>>> L.plot() #
↳needs sage.plot
Graphics object consisting of ... graphics primitives
```

If a coloring is passed, the different arcs are plotted with the corresponding colors (see `colorings()`):

```
sage: B = BraidGroup(4)
sage: b = B([1, 2, 3, 1, 2, -1, -3, 2, 3])
sage: L = Link(b)
sage: L.plot(color=L.colorings()[0]) #
↳needs sage.plot
Graphics object consisting of ... graphics primitives
```

```
>>> from sage.all import *
>>> B = BraidGroup(Integer(4))
>>> b = B([Integer(1), Integer(2), Integer(3), Integer(1), Integer(2), -Integer(1),
(continues on next page)
```

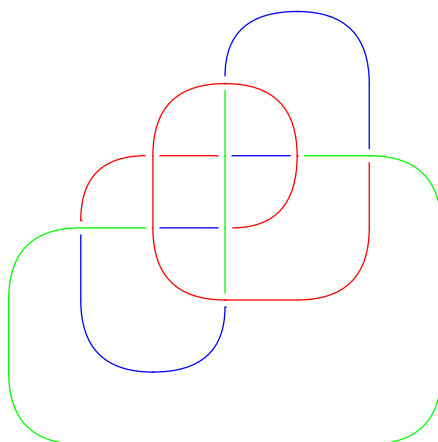


(continued from previous page)

```

↪ Integer(3), Integer(2), Integer(3)])
>>> L = Link(b)
>>> L.plot(color=L.colorings()[Integer(0)])
↪ # needs sage.plot
Graphics object consisting of ... graphics primitives

```

**regions()**

Return the regions from the link diagram of `self`.

Regions are obtained always turning left at each crossing.

Then the regions are represented as a list with the segments that form its boundary, with a sign depending on the orientation of the segment as part of the boundary.

**EXAMPLES:**

```

sage: L = Link([[[-1, +2, -3, 4, +5, +1, -2, +6, +7, 3, -4, -7, -6, -5]],
....:          [-1, -1, -1, -1, 1, -1, 1]])
sage: L.regions()
[[14, -5, 12, -9], [13, 9], [11, 5, 1, 7, 3], [10, -3, 8, -13],
 [6, -1], [4, -11], [2, -7], [-2, -6, -14, -8], [-4, -10, -12]]
sage: L = Link([[1, -2, 3, -4, 2, -1, 4, -3]], [1, 1, -1, -1])

```

(continues on next page)

(continued from previous page)

```

sage: L.regions()
[[8, 4], [7, -4, 1], [6, -1, -3], [5, 3, -8], [2, -5, -7], [-2, -6]]
sage: L = Link([[[-1, +2, 3, -4, 5, -6, 7, 8, -2, -5, +6, +1, -8, -3, 4, -7]],
....:          [-1, -1, -1, -1, 1, 1, -1, 1]])
sage: L.regions()
[[16, 8, 14, 4], [15, -4], [13, -8, 1], [12, -1, -7], [11, 7, -16, 5],
 [10, -5, -15, -3], [9, 3, -14], [6, -11], [2, -9, -13], [-2, -12, -6, -10]]

sage: B = BraidGroup(2)
sage: L = Link(B([[-1, -1, -1]]))
sage: L.regions()
[[6, -5], [5, 1, 3], [4, -3], [2, -1], [-2, -6, -4]]
sage: L = Link([[1, -2, 3, -4], [-1, 5, -3, 2, -5, 4]],
....:          [-1, 1, 1, -1, -1]])
sage: L.regions()
[[10, -4, -7], [9, 7, -3], [8, 3], [6, -9, -2], [5, 2, -8, 4],
 [1, -5], [-1, -10, -6]]
sage: L = Link([[1, 3, 3, 2], [2, 4, 4, 5], [5, 6, 6, 7], [7, 8, 8, 1]])
sage: L.regions()
[[-3], [-4], [-6], [-8], [7, 1, 2, 5], [-1, 8, -7, 6, -5, 4, -2, 3]]

```

```

>>> from sage.all import *
>>> L = Link([[-Integer(1), +Integer(2), -Integer(3), Integer(4), ↵
↵+Integer(5), +Integer(1), -Integer(2), +Integer(6), +Integer(7), Integer(3),
↵ -Integer(4), -Integer(7), -Integer(6), -Integer(5)],
...          [-Integer(1), -Integer(1), -Integer(1), -Integer(1), Integer(1),
↵ -Integer(1), Integer(1)])]
>>> L.regions()
[[14, -5, 12, -9], [13, 9], [11, 5, 1, 7, 3], [10, -3, 8, -13],
 [6, -1], [4, -11], [2, -7], [-2, -6, -14, -8], [-4, -10, -12]]
>>> L = Link([[[Integer(1), -Integer(2), Integer(3), -Integer(4), Integer(2), ↵
↵-Integer(1), Integer(4), -Integer(3)], [Integer(1), Integer(1), -Integer(1),
↵ -Integer(1)]]])
>>> L.regions()
[[8, 4], [7, -4, 1], [6, -1, -3], [5, 3, -8], [2, -5, -7], [-2, -6]]
>>> L = Link([[-Integer(1), +Integer(2), Integer(3), -Integer(4), Integer(5),
↵ -Integer(6), Integer(7), Integer(8), -Integer(2), -Integer(5), +Integer(6),
↵ +Integer(1), -Integer(8), -Integer(3), Integer(4), -Integer(7)],
...          [-Integer(1), -Integer(1), -Integer(1), -Integer(1), Integer(1),
↵ Integer(1), -Integer(1), Integer(1)])]
>>> L.regions()
[[16, 8, 14, 4], [15, -4], [13, -8, 1], [12, -1, -7], [11, 7, -16, 5],
 [10, -5, -15, -3], [9, 3, -14], [6, -11], [2, -9, -13], [-2, -12, -6, -10]]

>>> B = BraidGroup(Integer(2))
>>> L = Link(B([[-Integer(1), -Integer(1), -Integer(1)]])
>>> L.regions()
[[6, -5], [5, 1, 3], [4, -3], [2, -1], [-2, -6, -4]]
>>> L = Link([[[Integer(1), -Integer(2), Integer(3), -Integer(4)], [-
↵Integer(1), Integer(5), -Integer(3), Integer(2), -Integer(5), Integer(4)],
...          [-Integer(1), Integer(1), Integer(1), -Integer(1), -
↵Integer(1)]]])

```

(continues on next page)

(continued from previous page)

```

>>> L.regions()
[[10, -4, -7], [9, 7, -3], [8, 3], [6, -9, -2], [5, 2, -8, 4],
 [1, -5], [-1, -10, -6]]
>>> L = Link([[Integer(1), Integer(3), Integer(3), Integer(2)], [Integer(2), ↵
↵Integer(4), Integer(4), Integer(5)], [Integer(5), Integer(6), Integer(6), ↵
↵Integer(7)], [Integer(7), Integer(8), Integer(8), Integer(1)]])
>>> L.regions()
[[-3], [-4], [-6], [-8], [7, 1, 2, 5], [-1, 8, -7, 6, -5, 4, -2, 3]]

```

**Note**

The link diagram is assumed to have only one completely isolated component. This is because otherwise some regions would have disconnected boundary.

**remove\_loops()**

Return an ambient isotopic link in which all loops are removed.

**EXAMPLES:**

```

sage: b = BraidGroup(4)((3, 2, -1, -1))
sage: L = Link(b)
sage: L.remove_loops()
Link with 2 components represented by 2 crossings
sage: K4 = Link([[1, 7, 2, 6], [3, 1, 4, 8], [5, 5, 6, 4], [7, 3, 8, 2]])
sage: K3 = K4.remove_loops()
sage: K3.pd_code()
[[1, 7, 2, 4], [3, 1, 4, 8], [7, 3, 8, 2]]
sage: U = Link([[1, 2, 2, 1]])
sage: U.remove_loops()
Link with 1 component represented by 0 crossings

```

```

>>> from sage.all import *
>>> b = BraidGroup(Integer(4))((Integer(3), Integer(2), -Integer(1), -
↵Integer(1)))
>>> L = Link(b)
>>> L.remove_loops()
Link with 2 components represented by 2 crossings
>>> K4 = Link([[Integer(1), Integer(7), Integer(2), Integer(6)], [Integer(3), ↵
↵Integer(1), Integer(4), Integer(8)], [Integer(5), Integer(5), Integer(6), ↵
↵Integer(4)], [Integer(7), Integer(3), Integer(8), Integer(2)]])
>>> K3 = K4.remove_loops()
>>> K3.pd_code()
[[1, 7, 2, 4], [3, 1, 4, 8], [7, 3, 8, 2]]
>>> U = Link([[Integer(1), Integer(2), Integer(2), Integer(1)]])
>>> U.remove_loops()
Link with 1 component represented by 0 crossings

```

**reverse()**

Return the reverse of `self`. This is the link obtained from `self` by reverting the orientation on all components.

**EXAMPLES:**

```

sage: K3 = Knot([[5, 2, 4, 1], [3, 6, 2, 5], [1, 4, 6, 3]])
sage: K3r = K3.reverse(); K3r.pd_code()
[[4, 1, 5, 2], [2, 5, 3, 6], [6, 3, 1, 4]]
sage: K3 == K3r
True

```

```

>>> from sage.all import *
>>> K3 = Knot([[Integer(5), Integer(2), Integer(4), Integer(1)], [Integer(3),
↪Integer(6), Integer(2), Integer(5)], [Integer(1), Integer(4), Integer(6),
↪Integer(3)]])
>>> K3r = K3.reverse(); K3r.pd_code()
[[4, 1, 5, 2], [2, 5, 3, 6], [6, 3, 1, 4]]
>>> K3 == K3r
True

```

a non reversable knot:

```

sage: K8_17 = Knot([[6, 1, 7, 2], [14, 7, 15, 8], [8, 4, 9, 3],
....:               [2, 14, 3, 13], [12, 6, 13, 5], [4, 10, 5, 9],
....:               [16, 11, 1, 12], [10, 15, 11, 16]])
sage: K8_17r = K8_17.reverse()
sage: b = K8_17.braid(); b
s0^2*s1^-1*(s1^-1*s0)^2*s1^-1
sage: br = K8_17r.braid(); br
s0^-1*s1*s0^-2*s1^2*s0^-1*s1
sage: b.is_conjugated(br)
False
sage: b == br.reverse()
False
sage: b.is_conjugated(br.reverse())
True
sage: K8_17b = Link(b)
sage: K8_17br = K8_17b.reverse()
sage: bbr = K8_17br.braid(); bbr
(s1^-1*s0)^2*s1^-2*s0^2
sage: br == bbr
False
sage: br.is_conjugated(bbr)
True

```

```

>>> from sage.all import *
>>> K8_17 = Knot([[Integer(6), Integer(1), Integer(7), Integer(2)],
↪[Integer(14), Integer(7), Integer(15), Integer(8)], [Integer(8), Integer(4),
↪Integer(9), Integer(3)],
...               [Integer(2), Integer(14), Integer(3), Integer(13)],
↪[Integer(12), Integer(6), Integer(13), Integer(5)], [Integer(4),
↪Integer(10), Integer(5), Integer(9)],
...               [Integer(16), Integer(11), Integer(1), Integer(12)],
↪[Integer(10), Integer(15), Integer(11), Integer(16)]])
>>> K8_17r = K8_17.reverse()
>>> b = K8_17.braid(); b
s0^2*s1^-1*(s1^-1*s0)^2*s1^-1

```

(continues on next page)

(continued from previous page)

```

>>> br = K8_17r.braid(); br
s0^-1*s1*s0^-2*s1^2*s0^-1*s1
>>> b.is_conjugated(br)
False
>>> b == br.reverse()
False
>>> b.is_conjugated(br.reverse())
True
>>> K8_17b = Link(b)
>>> K8_17br = K8_17b.reverse()
>>> bbr = K8_17br.braid(); bbr
(s1^-1*s0)^2*s1^-2*s0^2
>>> br == bbr
False
>>> br.is_conjugated(bbr)
True

```

**seifert\_circles()**

Return the Seifert circles from the link diagram of `self`.

Seifert circles are the circles obtained by smoothing all crossings respecting the orientation of the segments.

Each Seifert circle is represented as a list of the segments that form it.

EXAMPLES:

```

sage: L = Link([[1, -2, 3, -4, 2, -1, 4, -3]], [1, 1, -1, -1])
sage: L.seifert_circles()
[[1, 7, 5, 3], [2, 6], [4, 8]]
sage: L = Link([[1, -2, 3, -4, 5, -6, 7, 8, -2, -5, 6, 1, -8, -3, 4, -7]],
....:          [-1, -1, -1, -1, 1, 1, -1, 1])
sage: L.seifert_circles()
[[1, 13, 9, 3, 15, 5, 11, 7], [2, 10, 6, 12], [4, 16, 8, 14]]
sage: L = Link([[1, -2, -3, 4, 5, 1, -2, 6, 7, 3, -4, -7, -6, -5]],
....:          [-1, -1, -1, -1, 1, -1, 1])
sage: L.seifert_circles()
[[1, 7, 3, 11, 5], [2, 8, 14, 6], [4, 12, 10], [9, 13]]
sage: L = Link([[1, 7, 2, 6], [7, 3, 8, 2], [3, 11, 4, 10], [11, 5, 12, 4],
....:          [14, 5, 1, 6], [13, 9, 14, 8], [12, 9, 13, 10]])
sage: L.seifert_circles()
[[1, 7, 3, 11, 5], [2, 8, 14, 6], [4, 12, 10], [9, 13]]
sage: L = Link([[1, -2, -3, 5], [4, -2, 6, -5], [-4, 1, -6, 3]],
....:          [-1, 1, 1, 1, -1, -1])
sage: L.seifert_circles()
[[1, 11, 8], [2, 7, 12, 4, 5, 10], [3, 9, 6]]

sage: B = BraidGroup(2)
sage: L = Link(B([1, 1, 1]))
sage: L.seifert_circles()
[[1, 3, 5], [2, 4, 6]]

```

```

>>> from sage.all import *
>>> L = Link([Integer(1), -Integer(2), Integer(3), -Integer(4), Integer(2),

```

(continues on next page)

(continued from previous page)

```

↪ -Integer(1), Integer(4), -Integer(3)], [Integer(1), Integer(1), -
↪ Integer(1), -Integer(1)])
>>> L.seifert_circles()
[[1, 7, 5, 3], [2, 6], [4, 8]]
>>> L = Link([[-Integer(1), Integer(2), Integer(3), -Integer(4), Integer(5), ↪
↪ -Integer(6), Integer(7), Integer(8), -Integer(2), -Integer(5), Integer(6), ↪
↪ Integer(1), -Integer(8), -Integer(3), Integer(4), -Integer(7)]],
...          [-Integer(1), -Integer(1), -Integer(1), -Integer(1), Integer(1),
↪ Integer(1), -Integer(1), Integer(1)])
>>> L.seifert_circles()
[[1, 13, 9, 3, 15, 5, 11, 7], [2, 10, 6, 12], [4, 16, 8, 14]]
>>> L = Link([[-Integer(1), Integer(2), -Integer(3), Integer(4), Integer(5), ↪
↪ Integer(1), -Integer(2), Integer(6), Integer(7), Integer(3), -Integer(4), -
↪ Integer(7), -Integer(6), -Integer(5)]],
...          [-Integer(1), -Integer(1), -Integer(1), -Integer(1), Integer(1),
↪ -Integer(1), Integer(1)])
>>> L.seifert_circles()
[[1, 7, 3, 11, 5], [2, 8, 14, 6], [4, 12, 10], [9, 13]]
>>> L = Link([Integer(1), Integer(7), Integer(2), Integer(6)], [Integer(7), ↪
↪ Integer(3), Integer(8), Integer(2)], [Integer(3), Integer(11), Integer(4), ↪
↪ Integer(10)], [Integer(11), Integer(5), Integer(12), Integer(4)],
...          [Integer(14), Integer(5), Integer(1), Integer(6)], [Integer(13),
↪ Integer(9), Integer(14), Integer(8)], [Integer(12), Integer(9), ↪
↪ Integer(13), Integer(10)])
>>> L.seifert_circles()
[[1, 7, 3, 11, 5], [2, 8, 14, 6], [4, 12, 10], [9, 13]]
>>> L = Link([[-Integer(1), Integer(2), -Integer(3), Integer(5)], ↪
↪ [Integer(4), -Integer(2), Integer(6), -Integer(5)], [-Integer(4), ↪
↪ Integer(1), -Integer(6), Integer(3)]],
...          [-Integer(1), Integer(1), Integer(1), Integer(1), -Integer(1), -
↪ Integer(1)])
>>> L.seifert_circles()
[[1, 11, 8], [2, 7, 12, 4, 5, 10], [3, 9, 6]]

>>> B = BraidGroup(Integer(2))
>>> L = Link(B([Integer(1), Integer(1), Integer(1)]))
>>> L.seifert_circles()
[[1, 3, 5], [2, 4, 6]]

```

**seifert\_matrix()**

Return the Seifert matrix associated with `self`.

**ALGORITHM:**

This is the algorithm presented in Section 3.3 of [Col2013].

**OUTPUT:**

The intersection matrix of a (not necessarily minimal) Seifert surface.

**EXAMPLES:**

```

sage: # needs sage.modules
sage: B = BraidGroup(4)

```

(continues on next page)

(continued from previous page)

```

sage: L = Link(B([-1, 3, 1, 3]))
sage: L.seifert_matrix()
[ 0  0]
[ 0 -1]
sage: B = BraidGroup(8)
sage: L = Link(B([-1, 3, 1, 5, 1, 7, 1, 6]))
sage: L.seifert_matrix()
[ 0  0  0]
[ 1 -1  0]
[ 0  1 -1]
sage: L = Link(B([-2, 4, 1, 6, 1, 4]))
sage: L.seifert_matrix()
[-1  0]
[ 0 -1]

```

```

>>> from sage.all import *
>>> # needs sage.modules
>>> B = BraidGroup(Integer(4))
>>> L = Link(B([-Integer(1), Integer(3), Integer(1), Integer(3)]))
>>> L.seifert_matrix()
[ 0  0]
[ 0 -1]
>>> B = BraidGroup(Integer(8))
>>> L = Link(B([-Integer(1), Integer(3), Integer(1), Integer(5), Integer(1),
↪Integer(7), Integer(1), Integer(6)]))
>>> L.seifert_matrix()
[ 0  0  0]
[ 1 -1  0]
[ 0  1 -1]
>>> L = Link(B([-Integer(2), Integer(4), Integer(1), Integer(6), Integer(1),
↪Integer(4)]))
>>> L.seifert_matrix()
[-1  0]
[ 0 -1]

```

**signature()**

Return the signature of `self`.

This is defined as the signature of the symmetric matrix

$$V + V^t,$$

where  $V$  is the *Seifert matrix*.

**See also**

`omega_signature()`, `seifert_matrix()`

**EXAMPLES:**

```

sage: # needs sage.modules
sage: B = BraidGroup(4)

```

(continues on next page)

(continued from previous page)

```

sage: L = Link(B([-1, 3, 1, 3]))
sage: L.signature()
-1
sage: B = BraidGroup(8)
sage: L = Link(B([-2, 4, 1, 6, 1, 4]))
sage: L.signature()
-2
sage: L = Link(B([1, 2, 1, 2]))
sage: L.signature()
-2

```

```

>>> from sage.all import *
>>> # needs sage.modules
>>> B = BraidGroup(Integer(4))
>>> L = Link(B([-Integer(1), Integer(3), Integer(1), Integer(3)]))
>>> L.signature()
-1
>>> B = BraidGroup(Integer(8))
>>> L = Link(B([-Integer(2), Integer(4), Integer(1), Integer(6), Integer(1), ↵
↵Integer(4)]))
>>> L.signature()
-2
>>> L = Link(B([Integer(1), Integer(2), Integer(1), Integer(2)]))
>>> L.signature()
-2

```

**writhe()**

Return the writhe of self.

**EXAMPLES:**

```

sage: L = Link([[1, -2, 3, -4, 2, -1, 4, -3]], [1, 1, -1, -1])
sage: L.writhe()
0
sage: L = Link([[[-1, 2, -3, 4, 5, 1, -2, 6, 7, 3, -4, -7, -6, -5]],
.....:          [-1, -1, -1, -1, 1, -1, 1]])
sage: L.writhe()
-3
sage: L = Link([[[-1, 2, 3, -4, 5, -6, 7, 8, -2, -5, 6, 1, -8, -3, 4, -7]],
.....:          [-1, -1, -1, -1, 1, 1, -1, 1]])
sage: L.writhe()
-2

```

```

>>> from sage.all import *
>>> L = Link([[Integer(1), -Integer(2), Integer(3), -Integer(4), Integer(2), ↵
↵-Integer(1), Integer(4), -Integer(3)], [Integer(1), Integer(1), -Integer(1),
↵ -Integer(1)])]
>>> L.writhe()
0
>>> L = Link([[[-Integer(1), Integer(2), -Integer(3), Integer(4), Integer(5), ↵
↵Integer(1), -Integer(2), Integer(6), Integer(7), Integer(3), -Integer(4), -
↵Integer(7), -Integer(6), -Integer(5)]],

```

(continues on next page)

(continued from previous page)

```
...          [-Integer(1), -Integer(1), -Integer(1), -Integer(1), ↵  
↵Integer(1), -Integer(1), Integer(1)]])  
>>> L.writhe()  
-3  
>>> L = Link([[[-Integer(1), Integer(2), Integer(3), -Integer(4), Integer(5), ↵  
↵-Integer(6), Integer(7), Integer(8), -Integer(2), -Integer(5), Integer(6), ↵  
↵Integer(1), -Integer(8), -Integer(3), Integer(4), -Integer(7)]]],  
...          [-Integer(1), -Integer(1), -Integer(1), -Integer(1), ↵  
↵Integer(1), Integer(1), -Integer(1), Integer(1)]]))  
>>> L.writhe()  
-2
```

## ACCESS TO THE KNOTINFO DATABASE

This module contains the class *KnotInfoBase* which is derived from *Enum* and provides knots and links listed in the databases at the web-pages *KnotInfo* and *LinkInfo* as its items.

This interface contains a set of about twenty knots and links statically as demonstration cases. The complete database can be installed as an optional Sage package using

- `sage -i database_knotinfo` (does not install if the current version is present)
- `sage -f database_knotinfo` (installs even if the current version is present)

This will install a *Python wrapper* for the original databases in Sage. This wrapper performs an automatic progress of version numbers. For more details and further install instructions please see the corresponding web-page.

To perform all the doctests concerning the usage of the database on the installation add the option `-c`. In this case (for instance `sage -f -c database_knotinfo`) the installation breaks on failing tests.

The installation of the complete database will be necessary in order to have access to all the properties recorded in the databases, as well.

If the entire database is installed as explained above, the import instructions for *KnotInfo* and *KnotInfoSeries*, which can be seen in the opening lines of the examples, are unnecessary.

Be aware that there are a couple of conventions used differently on *KnotInfo* as in Sage.

For different conventions regarding normalization of the polynomial invariants see the according documentation of *KnotInfoBase.homfly\_polynomial()*, *KnotInfoBase.jones\_polynomial()* and *KnotInfoBase.alexander\_polynomial()*.

Also, note that the braid notation is used according to Sage, even though in the source where it is taken from, the braid generators are assumed to have a negative crossing which would be opposite to the convention in Sage (see definition 3 of Gittings, T., “Minimum Braids: A Complete Invariant of Knots and Links”).

Furthermore, note that not all columns available in the database are visible on the web pages. It is planned to remove non-visible columns from the database in the future (see the *Python Wrapper* for updated information).

EXAMPLES:

```
sage: L = KnotInfo.L4a1_0
sage: L.pd_notation()
[[6, 1, 7, 2], [8, 3, 5, 4], [2, 5, 3, 6], [4, 7, 1, 8]]
sage: L.pd_notation(original=True)
'{{6, 1, 7, 2}, {8, 3, 5, 4}, {2, 5, 3, 6}, {4, 7, 1, 8}}'
sage: L.is_knot()
False
sage: L.num_components()
2
```

```
>>> from sage.all import *
>>> L = KnotInfo.L4a1_0
>>> L.pd_notation()
[[6, 1, 7, 2], [8, 3, 5, 4], [2, 5, 3, 6], [4, 7, 1, 8]]
>>> L.pd_notation(original=True)
'{{6, 1, 7, 2}, {8, 3, 5, 4}, {2, 5, 3, 6}, {4, 7, 1, 8}}'
>>> L.is_knot()
False
>>> L.num_components()
2
```

Items for knots need a leading  $K$  for technical reason:

```
sage: K = KnotInfo.K4_1
sage: K.is_knot()
True
```

```
>>> from sage.all import *
>>> K = KnotInfo.K4_1
>>> K.is_knot()
True
```

Injecting the variable name into the namespace:

```
sage: KnotInfo.K5_1.inject()
Defining K5_1
sage: K5_1.dt_notation()
[6, 8, 10, 2, 4]
```

```
>>> from sage.all import *
>>> KnotInfo.K5_1.inject()
Defining K5_1
>>> K5_1.dt_notation()
[6, 8, 10, 2, 4]
```

Defining a link from the original name string:

```
sage: KnotInfo('L6a1{1}').inject()
Defining L6a1_1
sage: L6a1_1.is_alternating()
True
```

```
>>> from sage.all import *
>>> KnotInfo('L6a1{1}').inject()
Defining L6a1_1
>>> L6a1_1.is_alternating()
True
```

Obtaining an instance of `Braid`:

```
sage: L.braid()
s1^-2*s0^-1*s1*s0^-1
```

(continues on next page)

(continued from previous page)

```
sage: type(_)
<class 'sage.groups.braid.BraidGroup_class_with_category.element_class'>
```

```
>>> from sage.all import *
>>> L.braid()
s1^-2*s0^-1*s1*s0^-1
>>> type(_)
<class 'sage.groups.braid.BraidGroup_class_with_category.element_class'>
```

Obtaining an instance of *Link*:

```
sage: l = L.link(); l
Link with 2 components represented by 4 crossings
sage: type(l)
<class 'sage.knots.link.Link'>
```

```
>>> from sage.all import *
>>> l = L.link(); l
Link with 2 components represented by 4 crossings
>>> type(l)
<class 'sage.knots.link.Link'>
```

If you have *SnapPy* installed inside Sage, you can obtain an instance of *Link*, too:

```
sage: # optional - snappy
sage: L6 = KnotInfo.L6a1_0
sage: l6s = L6.link(snappy=True); l6s
...
<Link: 2 comp; 6 cross>
sage: type(l6s)
<class 'spherogram.links.invariants.Link'>
sage: l6 = L6.link()
sage: l6 == l6s.sage_link()
True
sage: L6.link(L6.items.name, snappy=True)
<Link L6a1: 2 comp; 6 cross>
sage: l6sn = _
sage: l6s == l6sn
False
sage: l6m = l6.mirror_image()
sage: l6sn.sage_link().is_isotopic(l6m)
True
```

```
>>> from sage.all import *
>>> # optional - snappy
>>> L6 = KnotInfo.L6a1_0
>>> l6s = L6.link(snappy=True); l6s
...
<Link: 2 comp; 6 cross>
>>> type(l6s)
<class 'spherogram.links.invariants.Link'>
>>> l6 = L6.link()
```

(continues on next page)

(continued from previous page)

```
>>> l6 == l6s.sage_link()
True
>>> L6.link(L6.items.name, snappy=True)
<Link L6a1: 2 comp; 6 cross>
>>> l6sn = _
>>> l6s == l6sn
False
>>> l6m = l6.mirror_image()
>>> l6sn.sage_link().is_isotopic(l6m)
True
```

But observe that the name conversion to SnapPy does not distinguish orientation types:

```
sage: L6b = KnotInfo.L6a1_1
sage: L6b.link(L6b.items.name, snappy=True) # optional - snappy
<Link L6a1: 2 comp; 6 cross>
sage: _.PD_code() == l6sn.PD_code() # optional - snappy
True
```

```
>>> from sage.all import *
>>> L6b = KnotInfo.L6a1_1
>>> L6b.link(L6b.items.name, snappy=True) # optional - snappy
<Link L6a1: 2 comp; 6 cross>
>>> _.PD_code() == l6sn.PD_code() # optional - snappy
True
```

Obtaining the HOMFLY-PT polynomial:

```
sage: L.homfly_polynomial()
-v^-1*z - v^-3*z - v^-3*z^-1 + v^-5*z^-1
sage: _ == l.homfly_polynomial(normalization='vz')
True
```

```
>>> from sage.all import *
>>> L.homfly_polynomial()
-v^-1*z - v^-3*z - v^-3*z^-1 + v^-5*z^-1
>>> _ == l.homfly_polynomial(normalization='vz')
True
```

Obtaining the original string from the database for an arbitrary property:

```
sage: K[K.items.classical_conway_name] # optional - database_knotinfo
'4_1'
```

```
>>> from sage.all import *
>>> K[K.items.classical_conway_name] # optional - database_knotinfo
'4_1'
```

Further methods:

```
sage: K.crossing_number()
4
```

(continues on next page)

(continued from previous page)

```

sage: K.gauss_notation()
[-1, 2, -3, 1, -4, 3, -2, 4]
sage: K.dt_notation()
[4, 6, 8, 2]
sage: K.determinant()
5
sage: K.symmetry_type()
'fully amphicheiral'
sage: _ == K[K.items.symmetry_type]
True
sage: K.is_reversible()
True
sage: K.is_amphicheiral()
True
sage: K.jones_polynomial()
↪needs sage.symbolic
t^2 - t - 1/t + 1/t^2 + 1
sage: K.kauffman_polynomial()
a^2*z^2 + a*z^3 - a^2 - a*z + 2*z^2 + a^-1*z^3 - 1 - a^-1*z + a^-2*z^2 - a^-2
sage: K.alexander_polynomial()
t^2 - 3*t + 1

```

```

>>> from sage.all import *
>>> K.crossing_number()
4
>>> K.gauss_notation()
[-1, 2, -3, 1, -4, 3, -2, 4]
>>> K.dt_notation()
[4, 6, 8, 2]
>>> K.determinant()
5
>>> K.symmetry_type()
'fully amphicheiral'
>>> _ == K[K.items.symmetry_type]
True
>>> K.is_reversible()
True
>>> K.is_amphicheiral()
True
>>> K.jones_polynomial()
↪needs sage.symbolic
t^2 - t - 1/t + 1/t^2 + 1
>>> K.kauffman_polynomial()
a^2*z^2 + a*z^3 - a^2 - a*z + 2*z^2 + a^-1*z^3 - 1 - a^-1*z + a^-2*z^2 - a^-2
>>> K.alexander_polynomial()
t^2 - 3*t + 1

```

Using the `column_type` of a property:

```

sage: def select_column(i):
.....:     return i.column_type() != i.types.OnlyLinks and K[i] == 'Y'
sage: [i.column_name() for i in K.items if select_column(i)] # optional - database_

```

(continues on next page)

(continued from previous page)

```
↪knotinfo
['Alternating', 'Fibered', 'Quasialternating', 'Adequate']
```

```
>>> from sage.all import *
>>> def select_column(i):
...     return i.column_type() != i.types.OnlyLinks and K[i] == 'Y'
>>> [i.column_name() for i in K.items if select_column(i)] # optional - database_
↪knotinfo
['Alternating', 'Fibered', 'Quasialternating', 'Adequate']
```

You can launch web-pages attached to the links:

```
sage: # not tested
sage: K.diagram()
True
sage: L.diagram(single=True)
True
sage: L.knot_atlas_webpage()
True
sage: K.knotilus_webpage()
True
```

```
>>> from sage.all import *
>>> # not tested
>>> K.diagram()
True
>>> L.diagram(single=True)
True
>>> L.knot_atlas_webpage()
True
>>> K.knotilus_webpage()
True
```

and the description web-pages of the properties:

```
sage: K.items.positive.description_webpage() # not tested
True
```

```
>>> from sage.all import *
>>> K.items.positive.description_webpage() # not tested
True
```

To see all the properties available in this interface you can use “tab-completion”. For example type `K.items.` and then hit the `Tab` key. You can select the item you want from the list. If you know some first letters type them first to obtain a reduced selection list.

In a similar way you may select the knots and links. Here you have to type `KnotInfo.` or `KnotInfo.L7` before stroking the `Tab` key. In the latter case the selection list will be reduced to proper links with 7 crossings.

Finally there is a method `Link.get_knotinfo()` of class `Link` to find an instance in the `KnotInfo` database:

```
sage: L = Link([[3,1,2,4], [8,9,1,7], [5,6,7,3], [4,18,6,5],
.....:         [17,19,8,18], [9,10,11,14], [10,12,13,11],
```

(continues on next page)

(continued from previous page)

```
..... [12,19,15,13], [20,16,14,15], [16,20,17,2]])
sage: L.get_knotinfo()
KnotInfo['K0_1']
```

```
>>> from sage.all import *
>>> L = Link([[Integer(3),Integer(1),Integer(2),Integer(4)], [Integer(8),Integer(9),
↪Integer(1),Integer(7)], [Integer(5),Integer(6),Integer(7),Integer(3)], [Integer(4),
↪Integer(18),Integer(6),Integer(5)],
... [Integer(17),Integer(19),Integer(8),Integer(18)], [Integer(9),
↪Integer(10),Integer(11),Integer(14)], [Integer(10),Integer(12),Integer(13),
↪Integer(11)],
... [Integer(12),Integer(19),Integer(15),Integer(13)], [Integer(20),
↪Integer(16),Integer(14),Integer(15)], [Integer(16),Integer(20),Integer(17),
↪Integer(2)]])
>>> L.get_knotinfo()
KnotInfo['K0_1']
```

## REFERENCES:

- [KnotInfo](#)
- [LinkInfo](#)

## AUTHORS:

- Sebastian Oehms August 2020: initial version
- Sebastian Oehms June 2022: add `conway_polynomial()` and `khovanov_polynomial()` ([Issue #33969](#))

Thanks to Chuck Livingston and Allison Moore for their support. For further acknowledgments see the corresponding hompages.

```
class sage.knots.knotinfo.KnotInfo(value, names=<not given>, *values, module=None, qualname=None,
                                type=None, start=1, boundary=None)
```

Bases: `KnotInfoBase`

`K0_1` = `'0_1'`

`K3_1` = `'3_1'`

`K4_1` = `'4_1'`

`K5_1` = `'5_1'`

`K5_2` = `'5_2'`

`K6_1` = `'6_1'`

`K6_2` = `'6_2'`

`K6_3` = `'6_3'`

`K7_1` = `'7_1'`

`K7_2` = `'7_2'`

`L2a1_0` = `'L2a1{0}'`

```

L2a1_1 = 'L2a1{1}'
L4a1_0 = 'L4a1{0}'
L4a1_1 = 'L4a1{1}'
L5a1_0 = 'L5a1{0}'
L5a1_1 = 'L5a1{1}'
L6a1_0 = 'L6a1{0}'
L6a1_1 = 'L6a1{1}'
L6a2_0 = 'L6a2{0}'
L6a2_1 = 'L6a2{1}'

```

```

class sage.knots.knotinfo.KnotInfoBase (value, names=<not given>, *values, module=None,
                                         qualname=None, type=None, start=1, boundary=None)

```

Bases: `Enum`

Enum class to select the knots and links listed in the databases at the web-pages [KnotInfo](#) and [LinkInfo](#).

EXAMPLES:

```

sage: [knot.name for knot in KnotInfo if knot.crossing_number() < 5]
['K0_1', 'K3_1', 'K4_1', 'L2a1_0', 'L2a1_1', 'L4a1_0', 'L4a1_1']

```

```

>>> from sage.all import *
>>> [knot.name for knot in KnotInfo if knot.crossing_number() < Integer(5)]
['K0_1', 'K3_1', 'K4_1', 'L2a1_0', 'L2a1_1', 'L4a1_0', 'L4a1_1']

```

More examples and information can be seen in the module header `knotinfo` (by typing):

```

sage: import sage.knots.knotinfo      # not tested
sage: sage.knots.knotinfo?             # not tested

```

```

>>> from sage.all import *
>>> import sage.knots.knotinfo        # not tested
>>> sage.knots.knotinfo?               # not tested

```

```

alexander_polynomial (var='t', original=False, laurent_poly=False)

```

Return the Alexander polynomial according to the value of column `alexander_polynomial` for this knot as an instance of `Polynomial`.

It is obtained from the Seifert matrix  $V$  of `self` by the following formula (see the `KnotInfo` description web-page; to launch it see the example below):

$$A(L) = \det(V - tV^t)$$

Here  $V^t$  stands for the transpose of  $V$ .

INPUT:

- `var` – (default: `'t'`) the variable
- `original` – boolean (default: `False`); if set to `True` the original table entry is returned as a string

- `laurent_poly` – boolean (default: `False`); see the note below

OUTPUT:

A polynomial over the integers, more precisely an instance of `Polynomial`. If `laurent_poly` is set to `True` a Laurent polynomial over the integers, more precisely an instance of `LaurentPolynomial` is returned. If `original` is set to `True` then a string is returned.

#### Note

As an invariant the Alexander polynomial is only unique up to a unit factor in the Laurent polynomial ring over the integers in the indeterminate  $t$ . While the normalization of the exponents in `KnotInfo` guarantees it to be a proper polynomial, this is not the case for the implementation in Sage. Use the keyword `laurent_poly` to achieve a normalization according to Sage. But, still there may be a difference in sign (see the example below).

EXAMPLES:

```
sage: K = KnotInfo.K4_1
sage: Ka = K.alexander_polynomial(); Ka
t^2 - 3*t + 1
```

```
>>> from sage.all import *
>>> K = KnotInfo.K4_1
>>> Ka = K.alexander_polynomial(); Ka
t^2 - 3*t + 1
```

Comparison with Sage's results:

```
sage: k = K.link()
sage: ka = k.alexander_polynomial(); ka
-t^-1 + 3 - t
sage: K.alexander_polynomial(laurent_poly=True)
t^-1 - 3 + t
sage: _ == -ka
True
```

```
>>> from sage.all import *
>>> k = K.link()
>>> ka = k.alexander_polynomial(); ka
-t^-1 + 3 - t
>>> K.alexander_polynomial(laurent_poly=True)
t^-1 - 3 + t
>>> _ == -ka
True
```

Launch the KnotInfo description web-page:

```
sage: K.items.alexander_polynomial.description_webpage() # not tested
True
```

```
>>> from sage.all import *
>>> K.items.alexander_polynomial.description_webpage() # not tested
```

(continues on next page)

(continued from previous page)

True

**braid()**Return the braid notation of `self` as an instance of `Braid`.

EXAMPLES:

```
sage: K = KnotInfo.K3_1
sage: K.braid()
s^3
sage: K.braid_notation()
(1, 1, 1)
sage: KnotInfo.K13n_1448.braid()      # optional - database_knotinfo
s0^-1*s1*s2*s3*s4*s3^2*s2^-1*s1^-1*s0*s2^-1*s1*(s3*s2)^2*s4^-1*s3*s2*s1^-
↪1*s3*s2^-1*s3
```

```
>>> from sage.all import *
>>> K = KnotInfo.K3_1
>>> K.braid()
s^3
>>> K.braid_notation()
(1, 1, 1)
>>> KnotInfo.K13n_1448.braid()      # optional - database_knotinfo
s0^-1*s1*s2*s3*s4*s3^2*s2^-1*s1^-1*s0*s2^-1*s1*(s3*s2)^2*s4^-1*s3*s2*s1^-
↪1*s3*s2^-1*s3
```

**braid\_index()**Return the value of column `braid_index` for this link as a Python int.

OUTPUT:

Python int giving the minimum of strands needed to represent `self` as closure of a braid.

EXAMPLES:

```
sage: L = KnotInfo.L4a1_0
sage: L.braid_index()
3
sage: KnotInfo.K13a_1.inject()      # optional - database_knotinfo
Defining K13a_1
sage: K13a_1.braid_index()          # optional - database_knotinfo
Traceback (most recent call last):
...
NotImplementedError: this integer is not provided by the database
```

```
>>> from sage.all import *
>>> L = KnotInfo.L4a1_0
>>> L.braid_index()
3
>>> KnotInfo.K13a_1.inject()      # optional - database_knotinfo
Defining K13a_1
>>> K13a_1.braid_index()          # optional - database_knotinfo
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
NotImplementedError: this integer is not provided by the database
```

**braid\_length()**

Return the value of column `braid_length` for this link as a Python int.

OUTPUT:

Python int giving the minimum length of a braid word needed to represent `self` as closure of a braid.

EXAMPLES:

```
sage: K = KnotInfo.K3_1
sage: K.braid_length()
3
```

```
>>> from sage.all import *
>>> K = KnotInfo.K3_1
>>> K.braid_length()
3
```

**braid\_notation(*original=False*)**

Return the value of column `braid_notation` for this link as a Python tuple (Tietze form). For more information type `KnotInfo.K0_1.items.braid_notation.description_webpage()`.

INPUT:

- `original` – boolean (default: `False`); if set to `True` the original table entry is returned as a string

OUTPUT:

Python tuple representing the braid whose closure is `self` in Tietze form.

..NOTE:

There has been a major change to braid representatives for proper links since version 2021.10.1. The former braid reresentatives can be obtained by the column ```braid_notation_old``` (see the final example below).

EXAMPLES:

```
sage: L = KnotInfo.L4a1_0
sage: L.braid_notation()
(-2, -2, -1, 2, -1)
sage: L.braid_notation(original=True)
'{3, {-2, -2, -1, 2, -1}}'
sage: L[L.items.braid_notation_old]
'{4, {1, -2, 3, -2, -1, -2, -3, -2}}'
```

```
>>> from sage.all import *
>>> L = KnotInfo.L4a1_0
>>> L.braid_notation()
(-2, -2, -1, 2, -1)
>>> L.braid_notation(original=True)
```

(continues on next page)

(continued from previous page)

```
'{3, {-2, -2, -1, 2, -1}}'
>>> L[L.items.braid_notation_old]
'{4, {1, -2, 3, -2, -1, -2, -3, -2}}'
```

**conway\_polynomial**(var='t', original=False)

Return the Conway polynomial according to the value of column `conway_polynomial` for this knot or link as an instance of `Polynomial`.

It is obtained from the Seifert matrix  $V$  of `self` by the following formula (see the `KnotInfo` description web-page; to launch it see the example below):

$$\nabla(L) = \det(t^{\frac{1}{2}}V - t^{-\frac{1}{2}}V^t)$$

Here  $V^t$  stands for the transpose of  $V$ .

INPUT:

- `var` – (default: 't') the variable
- `original` – boolean (default: False); if set to True the original table entry is returned as a string

OUTPUT:

A polynomial over the integers, more precisely an instance of `Polynomial`. If `original` is set to True then a string is returned.

EXAMPLES:

```
sage: K = KnotInfo.K4_1
sage: Kc = K.conway_polynomial(); Kc
-t^2 + 1
sage: L = KnotInfo.L5a1_0
sage: Lc = L.conway_polynomial(); Lc
t^3
```

```
>>> from sage.all import *
>>> K = KnotInfo.K4_1
>>> Kc = K.conway_polynomial(); Kc
-t^2 + 1
>>> L = KnotInfo.L5a1_0
>>> Lc = L.conway_polynomial(); Lc
t^3
```

Comparison to Sage's results:

```
sage: Kc == K.link().conway_polynomial()
True
sage: Lc == L.link().conway_polynomial()
True
```

```
>>> from sage.all import *
>>> Kc == K.link().conway_polynomial()
True
>>> Lc == L.link().conway_polynomial()
True
```

Launch the KnotInfo description web-page:

```
sage: K.items.conway_polynomial.description_webpage() # not tested
True
```

```
>>> from sage.all import *
>>> K.items.conway_polynomial.description_webpage() # not tested
True
```

**cosmetic\_crossing\_conjecture\_verified()**

Return whether the Cosmetic Crossing Conjecture has been verified for *self*.

From the KnotInfo [description page](#):

A crossing change in a diagram of a knot  $K$  is called cosmetic if the resulting diagram also represents  $K$ . The cosmetic crossing conjecture posits that for any knot  $K$ , the only cosmetic crossing changes are nugatory, i.e. there exists an embedded 2-sphere in  $S^3$  which intersects  $K$  only at the two points of the relevant crossing. Conversely, it is not hard to see that any nugatory crossing change is cosmetic.

EXAMPLES:

```
sage: knots = [K for K in KnotInfo if K.is_knot() and K.crossing_number() <= 10]
sage: all(K.cosmetic_crossing_conjecture_verified() for K in knots)
True
```

```
>>> from sage.all import *
>>> knots = [K for K in KnotInfo if K.is_knot() and K.crossing_number() <= Integer(10)]
>>> all(K.cosmetic_crossing_conjecture_verified() for K in knots)
True
```

**crossing\_number()**

Return the minimal number of crossings of *self*.

#### Note

In contrast to the number of crossings displayed for instances of *Link* this number is the minimum over all possible diagrams of the link. The number of crossings displayed in the representation string of *Link* refers to the special diagram which could be larger.

EXAMPLES:

```
sage: KnotInfo.L4a1_0.crossing_number()
4
sage: KnotInfo.K3_1.crossing_number()
3
sage: Link(KnotInfo.L4a1_0.braid())
Link with 2 components represented by 5 crossings
```

```
>>> from sage.all import *
>>> KnotInfo.L4a1_0.crossing_number()
```

(continues on next page)

(continued from previous page)

```

4
>>> KnotInfo.K3_1.crossing_number()
3
>>> Link(KnotInfo.L4a1_0.braid())
Link with 2 components represented by 5 crossings

```

**determinant()**

Return the determinant of `self`.

From the KnotInfo description page:

The determinant of a knot is  $\det(V + V^t)$ , where  $V$  is a Seifert matrix for the knot.

To read the complete description type `KnotInfo.K0_1.items.determinant.description_webpage()`.

**Note**

KnotInfo's value for the unknot `0_1` is zero. This is not compatible with Sage's result (the value of the Alexander polynomial at -1). Since this method is needed to identify Sage links we take the according value in that case.

**EXAMPLES:**

```

sage: KnotInfo.L4a1_0.determinant()
4
sage: KnotInfo.K3_1.determinant()
3
sage: KnotInfo.K0_1.determinant()
1

```

```

>>> from sage.all import *
>>> KnotInfo.L4a1_0.determinant()
4
>>> KnotInfo.K3_1.determinant()
3
>>> KnotInfo.K0_1.determinant()
1

```

**diagram(single=False, new=0, autoraise=True)**

Launch the diagram of `self` given on the KnotInfo web-page.

INPUT:

- `single` – boolean (default: `False`); if set to `True` only one diagram is shown
- `new` – integer according to `open()` of `webbrowser` (0 default, 1 new window, 2 new tab)
- `autoraise` – boolean (default: `True`)

**EXAMPLES:**

```

sage: K = KnotInfo.K3_1
sage: K.diagram() # not tested
True

```

(continues on next page)

(continued from previous page)

```
sage: K.diagram(single=True) # not tested
True
```

```
>>> from sage.all import *
>>> K = KnotInfo.K3_1
>>> K.diagram() # not tested
True
>>> K.diagram(single=True) # not tested
True
```

**dt\_notation** (*original=False*)

Return the value of column `dt_notation` for this link as a Python list of Python lists. For more information type `KnotInfo.K0_1.items.dt_notation.description_webpage()`.

INPUT:

- `original` – boolean (default: `False`); if set to `True` the original table entry is returned as a string

OUTPUT:

Python list of python lists each entry of the outer list representing a crossing.

EXAMPLES:

```
sage: L = KnotInfo.L4a1_0
sage: L.dt_notation()
[[6, 8], [2, 4]]
sage: L.dt_notation(original=True)
'[{6, 8}, {2, 4}]'
sage: L = KnotInfo.L4a1_0
sage: K = KnotInfo.K4_1
sage: K.dt_notation()
[4, 6, 8, 2]
```

```
>>> from sage.all import *
>>> L = KnotInfo.L4a1_0
>>> L.dt_notation()
[[6, 8], [2, 4]]
>>> L.dt_notation(original=True)
'[{6, 8}, {2, 4}]'
>>> L = KnotInfo.L4a1_0
>>> K = KnotInfo.K4_1
>>> K.dt_notation()
[4, 6, 8, 2]
```

**gauss\_notation** (*original=False*)

Return the value of column `gauss_notation` for this link as a Python list of Python lists. For more information type `KnotInfo.K0_1.items.gauss_notation.description_webpage()`.

INPUT:

- `original` – boolean (default: `False`); if set to `True` the original table entry is returned as a string

EXAMPLES:

```
sage: L = KnotInfo.L4a1_0
sage: L.gauss_notation()
[[1, -3, 2, -4], [3, -1, 4, -2]]
sage: L.gauss_notation(original=True)
'{{1, -3, 2, -4}, {3, -1, 4, -2}}'
```

```
>>> from sage.all import *
>>> L = KnotInfo.L4a1_0
>>> L.gauss_notation()
[[1, -3, 2, -4], [3, -1, 4, -2]]
>>> L.gauss_notation(original=True)
'{{1, -3, 2, -4}, {3, -1, 4, -2}}'
```

**homfly\_polynomial**(var1='v', var2='z', original=False)

Return the HOMFLY-PT polynomial according to the value of column `homfly_polynomial` for this knot or link (in the latter case the column `homflypt_polynomial` is used) as an instance of the element class according to the output of `Link.homfly_polynomial()` of `Link`.

The HOMFLY-PT polynomial  $P(L)$  of a link  $L$  satisfies the following skein relation (see the corresponding [KnotInfo description page](#)):

$$P(O) = 1, \quad v^{-1}P(L_+) - vP(L_-) = zP(L_0)$$

INPUT:

- `var1` – string (default: 'v'); for the name of the first variable
- `var2` – string (default: 'z'); for the name of the second variable
- `original` – boolean (default: False); if set to True the original table entry is returned as a string

OUTPUT:

A Laurent polynomial over the integers, more precisely an instance of `LaurentPolynomial_mpair`. If `original` is set to True then a string is returned.

#### Note

The skein-relation for the HOMFLY-PT polynomial given on KnotInfo does not match the default used in Sage. For comparison you have to use the keyword argument `normalization='vz'` on the side of Sage.

EXAMPLES:

```
sage: K3_1 = KnotInfo.K3_1
sage: PK3_1 = K3_1.homfly_polynomial(); PK3_1
-v^4 + v^2*z^2 + 2*v^2
sage: K3_1.homfly_polynomial(original=True)
'(2*v^2-v^4)+v^2*z^2'
sage: PK3_1 == K3_1.link().homfly_polynomial(normalization='vz')
True
```

```
>>> from sage.all import *
>>> K3_1 = KnotInfo.K3_1
```

(continues on next page)

(continued from previous page)

```
>>> PK3_1 = K3_1.homfly_polynomial(); PK3_1
-v^4 + v^2*z^2 + 2*v^2
>>> K3_1.homfly_polynomial(original=True)
'(2*v^2-v^4)+v^2*z^2'
>>> PK3_1 == K3_1.link().homfly_polynomial(normalization='vz')
True
```

for proper links:

```
sage: L4a1_1 = KnotInfo.L4a1_1
sage: PL4a1_1 = L4a1_1.homfly_polynomial(var1='x', var2='y'); PL4a1_1
-x^5*y + x^3*y^3 - x^5*y^-1 + 3*x^3*y + x^3*y^-1
sage: _ == L4a1_1.link().homfly_polynomial('x', 'y', 'vz')
True
```

```
>>> from sage.all import *
>>> L4a1_1 = KnotInfo.L4a1_1
>>> PL4a1_1 = L4a1_1.homfly_polynomial(var1='x', var2='y'); PL4a1_1
-x^5*y + x^3*y^3 - x^5*y^-1 + 3*x^3*y + x^3*y^-1
>>> _ == L4a1_1.link().homfly_polynomial('x', 'y', 'vz')
True
```

check the skein-relation from the KnotInfo description page (applied to one of the positive crossings of the right-handed trefoil):

```
sage: R = PK3_1.parent()
sage: PO = R.one()
sage: L2a1_1 = KnotInfo.L2a1_1
sage: PL2a1_1 = L2a1_1.homfly_polynomial()
sage: v, z = R.gens()
sage: ~v*PK3_1 -v*PO == z*PL2a1_1
True
```

```
>>> from sage.all import *
>>> R = PK3_1.parent()
>>> PO = R.one()
>>> L2a1_1 = KnotInfo.L2a1_1
>>> PL2a1_1 = L2a1_1.homfly_polynomial()
>>> v, z = R.gens()
>>> ~v*PK3_1 -v*PO == z*PL2a1_1
True
```

#### REFERENCES:

- [Wikipedia article HOMFLY\\_polynomial](#)

**inject** (*verbose=True*)

Inject `self` with its name into the namespace of the Python code from which this function is called.

INPUT:

- `verbose` – boolean (default: `True`); whether to suppress the message printed on the invocation

EXAMPLES:

```
sage: KnotInfo.K5_2.inject()
Defining K5_2
sage: K5_2.is_alternating()
True
```

```
>>> from sage.all import *
>>> KnotInfo.K5_2.inject()
Defining K5_2
>>> K5_2.is_alternating()
True
```

**is\_adequate()**

Return whether `self` is adequate.

EXAMPLES:

```
sage: KnotInfo.K5_2.is_adequate() # optional - database_knotinfo
True
```

```
>>> from sage.all import *
>>> KnotInfo.K5_2.is_adequate() # optional - database_knotinfo
True
```

**is\_almost\_alternating()**

Return whether `self` is almost alternating.

EXAMPLES:

```
sage: KnotInfo.K5_2.is_almost_alternating() # optional - database_knotinfo
False
```

```
>>> from sage.all import *
>>> KnotInfo.K5_2.is_almost_alternating() # optional - database_knotinfo
False
```

**is\_alternating()**

Return whether `self` is alternating.

EXAMPLES:

```
sage: KnotInfo.K5_2.is_alternating()
True
```

```
>>> from sage.all import *
>>> KnotInfo.K5_2.is_alternating()
True
```

**is\_amphicheiral(positive=False)**

Return whether `self` is amphicheiral.

INPUT:

- `positive` – boolean (default: `False`); whether to check if `self` is positive or negative amphicheiral (see documentation of `symmetry_type()`)

OUTPUT: boolean or `None` if this cannot be determined

`True` if `self` is fully or negative amphicheiral per default. If `positive` is set to `True` than fully and positive amphicheiral links give `True`.

#### Note

For proper links this property is not provided in the database. Anyway, we support it here in this case, as well, except for a few items where it cannot be determined easily and where `None` is returned as answer.

#### EXAMPLES:

```
sage: # optional - database_knotinfo
sage: Kp = KnotInfo.K12a_427
sage: Kp.is_amphicheiral()
False
sage: Kp.is_amphicheiral(positive=True)
True
sage: Kn = KnotInfo.K10_88
sage: Kn.is_amphicheiral()
True
sage: Kn.is_amphicheiral(positive=True)
False
sage: KnotInfo.L4a1_0.is_amphicheiral()
False
sage: KnotInfo.L10n59_1.is_amphicheiral()
True
sage: KnotInfo.L10n36_0.inject()
Defining L10n36_0
sage: L10n36_0.is_amphicheiral() is None
True
```

```
>>> from sage.all import *
>>> # optional - database_knotinfo
>>> Kp = KnotInfo.K12a_427
>>> Kp.is_amphicheiral()
False
>>> Kp.is_amphicheiral(positive=True)
True
>>> Kn = KnotInfo.K10_88
>>> Kn.is_amphicheiral()
True
>>> Kn.is_amphicheiral(positive=True)
False
>>> KnotInfo.L4a1_0.is_amphicheiral()
False
>>> KnotInfo.L10n59_1.is_amphicheiral()
True
>>> KnotInfo.L10n36_0.inject()
Defining L10n36_0
>>> L10n36_0.is_amphicheiral() is None
True
```

`is_fibered()`

Return whether `self` is fibered.

EXAMPLES:

```
sage: KnotInfo.K6_3.is_fibered()
True
```

```
>>> from sage.all import *
>>> KnotInfo.K6_3.is_fibered()
True
```

**is\_hyperbolic()**

Return whether `self` is hyperbolic.

EXAMPLES:

```
sage: KnotInfo.K3_1.is_hyperbolic()
False
sage: KnotInfo.K5_2.is_hyperbolic()
True
```

```
>>> from sage.all import *
>>> KnotInfo.K3_1.is_hyperbolic()
False
>>> KnotInfo.K5_2.is_hyperbolic()
True
```

**is\_knot()**

Return whether `self` is a knot or a proper link.

EXAMPLES:

```
sage: KnotInfo.L7a1_0.is_knot()           # optional - database_knotinfo
False
sage: KnotInfo.K6_3.is_knot()
True
```

```
>>> from sage.all import *
>>> KnotInfo.L7a1_0.is_knot()           # optional - database_knotinfo
False
>>> KnotInfo.K6_3.is_knot()
True
```

**is\_oriented()**

Return whether `self` is oriented.

EXAMPLES:

```
sage: KnotInfo.L6a2_1.is_oriented()
True
```

```
>>> from sage.all import *
>>> KnotInfo.L6a2_1.is_oriented()
True
```

**is\_positive()**

Return whether `self` is positive.

EXAMPLES:

```
sage: KnotInfo.K5_2.is_positive()
True
```

```
>>> from sage.all import *
>>> KnotInfo.K5_2.is_positive()
True
```

**is\_positive\_braid()**

Return whether `self` is a positive braid.

EXAMPLES:

```
sage: KnotInfo.K5_2.is_positive_braid() # optional - database_knotinfo
False
```

```
>>> from sage.all import *
>>> KnotInfo.K5_2.is_positive_braid() # optional - database_knotinfo
False
```

**is\_quasi\_alternating()**

Return whether `self` is quasi alternating.

EXAMPLES:

```
sage: KnotInfo.K5_2.is_quasi_alternating() # optional - database_knotinfo
True
```

```
>>> from sage.all import *
>>> KnotInfo.K5_2.is_quasi_alternating() # optional - database_knotinfo
True
```

**is\_quasipositive()**

Return whether `self` is quasi-positive.

EXAMPLES:

```
sage: KnotInfo.K5_2.is_quasipositive() # optional - database_knotinfo
True
```

```
>>> from sage.all import *
>>> KnotInfo.K5_2.is_quasipositive() # optional - database_knotinfo
True
```

**is\_recoverable(unique=True)**

Return if `self` can be recovered from its conversion to Sage links using the `pd_notation` and the `braid_notation` and their mirror images.

The method is indirectly used by the `TestSuite` of the series of `self`.

INPUT:

- `unique` – boolean (default: `True`); if set to `False` it is only checked if `self` is among the recovered items

EXAMPLES:

```
sage: KnotInfo.L4a1_0.inject()
Defining L4a1_0
sage: L4a1_0.is_recoverable()
True
sage: L4a1_0.is_recoverable(unique=False)
True
sage: KnotInfo.L5a1_0.inject()
Defining L5a1_0
sage: L5a1_0.is_recoverable()
False
sage: L5a1_0.is_recoverable(unique=False)
True
```

```
>>> from sage.all import *
>>> KnotInfo.L4a1_0.inject()
Defining L4a1_0
>>> L4a1_0.is_recoverable()
True
>>> L4a1_0.is_recoverable(unique=False)
True
>>> KnotInfo.L5a1_0.inject()
Defining L5a1_0
>>> L5a1_0.is_recoverable()
False
>>> L5a1_0.is_recoverable(unique=False)
True
```

**`is_reversible()`**

Return whether `self` is reversible.

EXAMPLES:

```
sage: KnotInfo.K6_3.is_reversible()
True
```

```
>>> from sage.all import *
>>> KnotInfo.K6_3.is_reversible()
True
```

**`is_strongly_quasipositive()`**

Return whether `self` is strongly quasi-positive.

EXAMPLES:

```
sage: KnotInfo.K5_2.is_strongly_quasipositive() # optional - database_knotinfo
True
```

```
>>> from sage.all import *
>>> KnotInfo.K5_2.is_strongly_quasipositive() # optional - database_knotinfo
True
```

**is\_unique()**

Return whether there is no other isotopic link in the database or not.

OUTPUT: boolean or None if this cannot be determined

EXAMPLES:

```
sage: KnotInfo.L4a1_0.is_unique()
True
sage: KnotInfo.L5a1_0.is_unique()
False
sage: L = KnotInfo.L9a43_0_1          # optional - database_knotinfo
sage: L.series(oriented=True).inject() # optional - database_knotinfo
Defining L9a43
sage: [(L,L.is_unique()) for L in L9a43] # optional - database_knotinfo
[(<KnotInfo.L9a43_0_0: 'L9a43{0,0}'>, True),
 (<KnotInfo.L9a43_1_0: 'L9a43{1,0}'>, False),
 (<KnotInfo.L9a43_0_1: 'L9a43{0,1}'>, None),
 (<KnotInfo.L9a43_1_1: 'L9a43{1,1}'>, False)]
```

```
>>> from sage.all import *
>>> KnotInfo.L4a1_0.is_unique()
True
>>> KnotInfo.L5a1_0.is_unique()
False
>>> L = KnotInfo.L9a43_0_1          # optional - database_knotinfo
>>> L.series(oriented=True).inject() # optional - database_knotinfo
Defining L9a43
>>> [(L,L.is_unique()) for L in L9a43] # optional - database_knotinfo
[(<KnotInfo.L9a43_0_0: 'L9a43{0,0}'>, True),
 (<KnotInfo.L9a43_1_0: 'L9a43{1,0}'>, False),
 (<KnotInfo.L9a43_0_1: 'L9a43{0,1}'>, None),
 (<KnotInfo.L9a43_1_1: 'L9a43{1,1}'>, False)]
```

**property items**

Return an Enum class to select a column item of the KnotInfo database.

EXAMPLES:

```
sage: L = KnotInfo.L4a1_0
sage: it = L.items
sage: [i.name for i in it if i.name.startswith('braid')]
['braid_index', 'braid_length', 'braid_notation', 'braid_notation_old']
sage: L.items.dt_notation.column_name()
'DT Notation'
```

```
>>> from sage.all import *
>>> L = KnotInfo.L4a1_0
>>> it = L.items
>>> [i.name for i in it if i.name.startswith('braid')]
['braid_index', 'braid_length', 'braid_notation', 'braid_notation_old']
>>> L.items.dt_notation.column_name()
'DT Notation'
```

To check if the item is available for proper links or only knots type:

```
sage: it.gauss_notation.column_type()
<KnotInfoColumnTypes.KnotsAndLinks: 'B'>
sage: it.dt_notation.column_type()
<KnotInfoColumnTypes.OnlyKnots: 'K'>
```

```
>>> from sage.all import *
>>> it.gauss_notation.column_type()
<KnotInfoColumnTypes.KnotsAndLinks: 'B'>
>>> it.dt_notation.column_type()
<KnotInfoColumnTypes.OnlyKnots: 'K'>
```

To see the description of the item in your web browser type:

```
sage: it.gauss_notation.description_webpage() # not tested
True
```

```
>>> from sage.all import *
>>> it.gauss_notation.description_webpage() # not tested
True
```

**jones\_polynomial** (*variab=None, skein\_normalization=False, puiseux=False, original=False, use\_sqrt=False*)

Return the Jones polynomial according to the value of column `jones_polynomial` for this knot or link as an element of the symbolic ring `SR` or an instance of `LaurentPolynomial` depending on the keyword `skein_normalization`. Using the keyword `puiseux` instead of an element of the symbolic ring an instance of `PuiseuxSeries` can be returned.

The Jones polynomial  $V(L)$  of a link  $L$  satisfies the following skein relation (see the corresponding [KnotInfo description page](#)):

$$V(O) = 1, \quad t^{-1}V(L_+) - tV(L_-) = (t^{\frac{1}{2}} - t^{-\frac{1}{2}})V(L_0)$$

INPUT:

- `variab` – variable (default: `None`); used according to `Link.jones_polynomial()`
- `skein_normalization` – boolean (default: `False`); used according to `Link.jones_polynomial()`
- `puiseux` – boolean (default: `True`); only used in case `skein_normalization=False`. If set to `True` instead of an element of the symbolic ring an instance of `PuiseuxSeries` is returned
- `original` – boolean (default: `False`); if set to `True` the original table entry is returned as a string
- `use_sqrt` – boolean (default: `False`); see the note below

OUTPUT:

Depends on the keywords (in excluding order):

- `original=True` a string according to the original value from the database
- `skein_normalization=True` a Laurent polynomial over the integers, more precisely an instance of `LaurentPolynomial`
- `puiseux=True` a puiseux series over the integers, more precisely an instance of `PuiseuxSeries`

In all other cases an element of the symbolic ring `SR`.

**Note**

There is a difference to Sage's conventions concerning the Jones polynomial in the case of proper links. KnotInfo does not display these polynomials in the indeterminate  $t$  used in the skein relation. Instead a variable  $x$  is used defined by  $x^2 = t$ . Sage uses  $t$  in both cases, knots and proper links. Thus, to obtain the Jones polynomial for a proper link in  $t$  you have to set the keyword `use_sqrt` to `True`.

**EXAMPLES:**

```
sage: K = KnotInfo.K4_1
sage: Kj = K.jones_polynomial(); Kj                                     #_
↪needs sage.symbolic
t^2 - t - 1/t + 1/t^2 + 1
sage: Kjs = K.jones_polynomial(skein_normalization=True); Kjs
A^-8 - A^-4 + 1 - A^4 + A^8
sage: Kjp = K.jones_polynomial(puiseux=True); Kjp
t^-2 - t^-1 + 1 - t + t^2
```

```
>>> from sage.all import *
>>> K = KnotInfo.K4_1
>>> Kj = K.jones_polynomial(); Kj                                     #_
↪needs sage.symbolic
t^2 - t - 1/t + 1/t^2 + 1
>>> Kjs = K.jones_polynomial(skein_normalization=True); Kjs
A^-8 - A^-4 + 1 - A^4 + A^8
>>> Kjp = K.jones_polynomial(puiseux=True); Kjp
t^-2 - t^-1 + 1 - t + t^2
```

for proper links:

```
sage: L = KnotInfo.L2a1_1
sage: Lj = L.jones_polynomial(); Lj                                     #_
↪needs sage.symbolic
-x^5 - x
sage: Ljt = L.jones_polynomial(use_sqrt=True); Ljt                     #_
↪needs sage.symbolic
-t^(5/2) - sqrt(t)
sage: Ljp = L.jones_polynomial(puiseux=True); Ljp
-t^(1/2) - t^(5/2)
sage: Ljs = L.jones_polynomial(skein_normalization=True); Ljs
-A^2 - A^10
sage: Lj.parent()                                                       #_
↪needs sage.symbolic
Symbolic Ring
sage: Ljt.parent()                                                       #_
↪needs sage.symbolic
Symbolic Ring
sage: Ljp.parent()
Puisseux Series Ring in t over Integer Ring
sage: Ljs.parent()
Univariate Laurent Polynomial Ring in A over Integer Ring
```

```

>>> from sage.all import *
>>> L = KnotInfo.L2a1_1
>>> Lj = L.jones_polynomial(); Lj                                     #_
↳needs sage.symbolic
-x^5 - x
>>> Ljt = L.jones_polynomial(use_sqrt=True); Ljt                     #_
↳needs sage.symbolic
-t^(5/2) - sqrt(t)
>>> Ljp = L.jones_polynomial(puiseux=True); Ljp
-t^(1/2) - t^(5/2)
>>> Ljs = L.jones_polynomial(skein_normalization=True); Ljs
-A^2 - A^10
>>> Lj.parent()                                                       #_
↳needs sage.symbolic
Symbolic Ring
>>> Ljt.parent()                                                       #_
↳needs sage.symbolic
Symbolic Ring
>>> Ljp.parent()
Puisseux Series Ring in t over Integer Ring
>>> Ljs.parent()
Univariate Laurent Polynomial Ring in A over Integer Ring

```

Comparison with Sage's results:

```

sage: k = K.link()
sage: kj = k.jones_polynomial()                                     #_
↳needs sage.symbolic
sage: bool(Kj == kj)                                                #_
↳needs sage.symbolic
True
sage: kjs = k.jones_polynomial(skein_normalization=True)
sage: Kjs == kjs
True
sage: l = L.link()
sage: lj = l.jones_polynomial()                                     #_
↳needs sage.symbolic
sage: bool(Lj == lj)                                                #_
↳needs sage.symbolic
False
sage: bool(Ljt == lj)      # see note above                         #_
↳needs sage.symbolic
True
sage: ljs = l.jones_polynomial(skein_normalization=True)
sage: Ljs == ljs
True

```

```

>>> from sage.all import *
>>> k = K.link()
>>> kj = k.jones_polynomial()                                     #_
↳needs sage.symbolic
>>> bool(Kj == kj)                                                #_
↳needs sage.symbolic

```

(continues on next page)

(continued from previous page)

```

True
>>> kjs = k.jones_polynomial(skein_normalization=True)
>>> Kjs == kjs
True
>>> l = L.link()
>>> lj = l.jones_polynomial() #_
↳needs sage.symbolic
>>> bool(Lj == lj) #_
↳needs sage.symbolic
False
>>> bool(Ljt == lj) # see note above #_
↳needs sage.symbolic
True
>>> ljs = l.jones_polynomial(skein_normalization=True)
>>> Ljs == ljs
True

```

Check the skein-relation from the KnotInfo description page (applied to one of the positive crossings of the right-handed trefoil):

```

sage: K3_1 = KnotInfo.K3_1

sage: # needs sage.symbolic
sage: K3_1j = K3_1.jones_polynomial()
sage: L2a1_1j = Ljt # see note above
sage: R = L2a1_1j.parent()
sage: Oj = R(1)
sage: t = R('t')
sage: lhs = expand(~t*K3_1j - t*Oj)
sage: rhs = expand((sqrt(t) - ~sqrt(t))*L2a1_1j)
sage: bool(lhs == rhs)
True

```

```

>>> from sage.all import *
>>> K3_1 = KnotInfo.K3_1

>>> # needs sage.symbolic
>>> K3_1j = K3_1.jones_polynomial()
>>> L2a1_1j = Ljt # see note above
>>> R = L2a1_1j.parent()
>>> Oj = R(Integer(1))
>>> t = R('t')
>>> lhs = expand(~t*K3_1j - t*Oj)
>>> rhs = expand((sqrt(t) - ~sqrt(t))*L2a1_1j)
>>> bool(lhs == rhs)
True

```

The same with the Puiseux series version:

```

sage: K3_1jp = K3_1.jones_polynomial(puiseux=True)
sage: L2a1_1jp = Ljp
sage: R = L2a1_1jp.parent()

```

(continues on next page)

(continued from previous page)

```
sage: Ojp = R(1)
sage: t = R('t')
sage: ~t*K3_1jp - t*Ojp == (t^(1/2)-~t^(1/2))*L2a1_1jp
True
```

```
>>> from sage.all import *
>>> K3_1jp = K3_1.jones_polynomial(puiseux=True)
>>> L2a1_1jp = Ljp
>>> R = L2a1_1jp.parent()
>>> Ojp = R(Integer(1))
>>> t = R('t')
>>> ~t*K3_1jp - t*Ojp == (t**(Integer(1)/Integer(2))-~t**(Integer(1)/
↳ Integer(2)))*L2a1_1jp
True
```

The same in the case of skein normalization (using  $t = A^4$ ):

```
sage: K3_1js = K3_1.jones_polynomial(skein_normalization=True)
sage: L2a1_1js = L.jones_polynomial(skein_normalization=True)
sage: Rs = K3_1js.parent()
sage: Ojs = Rs.one()
sage: A, = Rs.gens()
sage: ~A^4*K3_1js - A^4*Ojs == (A^2-~A^2)*L2a1_1js
True
```

```
>>> from sage.all import *
>>> K3_1js = K3_1.jones_polynomial(skein_normalization=True)
>>> L2a1_1js = L.jones_polynomial(skein_normalization=True)
>>> Rs = K3_1js.parent()
>>> Ojs = Rs.one()
>>> A, = Rs.gens()
>>> ~A**Integer(4)*K3_1js - A**Integer(4)*Ojs == (A**Integer(2)-~
↳ A**Integer(2))*L2a1_1js
True
```

## REFERENCES:

- [Wikipedia article Jones\\_polynomial](#)

**kauffman\_polynomial** (*var1='a', var2='z', original=False*)

Return the Kauffman polynomial according to the value of column `kauffman_polynomial` for this knot or link as an instance of `LaurentPolynomial_mpair`.

The Kauffman polynomial  $F(L)$  respectively its corresponding invariant under regular isotopy  $\Delta(L) = a^{w(L)}F(L)$  where  $w(L)$  is the writhe of the link  $L$  satisfies the following skein relation (see the corresponding [KnotInfo description page](#)):

$$\Delta(O) = 1, \quad \Delta(L_+) - \Delta(L_-) = z(\Delta(L_0) + \Delta(L_\infty))$$

Furthermore, removing a curl of sign  $\epsilon$  leads to a multiplication of  $\Delta(L)$  with  $a^\epsilon$ .

INPUT:

- `var1` – (default: 'a') the first variable
- `var2` – (default: 'z') the second variable

- `original` – boolean (default: `False`); if set to `True` the original table entry is returned as a string

OUTPUT:

A Laurent polynomial over the integers, more precisely an instance of `LaurentPolynomial_mpair`. If `original` is set to `False` then a string is returned.

EXAMPLES:

```
sage: L = KnotInfo.L2a1_1
sage: K = KnotInfo.K4_1

sage: L.kauffman_polynomial()
a^-1*z - a^-1*z^-1 + a^-2 + a^-3*z - a^-3*z^-1
sage: K.kauffman_polynomial()
a^2*z^2 + a*z^3 - a^2 - a*z + 2*z^2 + a^-1*z^3 - 1 - a^-1*z + a^-2*z^2 - a^-2
```

```
>>> from sage.all import *
>>> L = KnotInfo.L2a1_1
>>> K = KnotInfo.K4_1

>>> L.kauffman_polynomial()
a^-1*z - a^-1*z^-1 + a^-2 + a^-3*z - a^-3*z^-1
>>> K.kauffman_polynomial()
a^2*z^2 + a*z^3 - a^2 - a*z + 2*z^2 + a^-1*z^3 - 1 - a^-1*z + a^-2*z^2 - a^-2
```

Comparison with Jones polynomial:

```
sage: # needs sage.symbolic
sage: k = _
sage: a, z = k.variables()
sage: j = K.jones_polynomial(skein_normalization=True)
sage: t, = j.variables()
sage: k.subs(a=-t^3, z=~t+t) == j.subs(t=t^4)
True
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> k = _
>>> a, z = k.variables()
>>> j = K.jones_polynomial(skein_normalization=True)
>>> t, = j.variables()
>>> k.subs(a=-t**Integer(3), z=~t+t) == j.subs(t=t**Integer(4))
True
```

Check the skein relation:

```
sage: K3_1 = KnotInfo.K3_1
sage: FK3_1 = K3_1.kauffman_polynomial()
sage: FL2a1_1 = L.kauffman_polynomial()
sage: z, a = FK3_1.variables()
sage: ΔK3_1 = FK3_1 * a**K3_1.link().writhe()
sage: ΔL2a1_1 = FL2a1_1 * a**L.link().writhe()
sage: Δ01p = a # unknot with one positive curl
sage: Δ02n = a**-2 # unknot with two negative curls
```

(continues on next page)

(continued from previous page)

```
sage: ΔK3_1 + ΔO1p == z*(ΔL2a1_1 + ΔO2n)
True
```

```
>>> from sage.all import *
>>> K3_1 = KnotInfo.K3_1
>>> FK3_1 = K3_1.kauffman_polynomial()
>>> FL2a1_1 = L.kauffman_polynomial()
>>> z, a = FK3_1.variables()
>>> ΔK3_1 = FK3_1 * a**K3_1.link().writhe()
>>> ΔL2a1_1 = FL2a1_1 * a**L.link().writhe()
>>> ΔO1p = a # unknot with one positive curl
>>> ΔO2n = a**-Integer(2) # unknot with two negative curls
>>> ΔK3_1 + ΔO1p == z*(ΔL2a1_1 + ΔO2n)
True
```

## REFERENCES:

- [Wikipedia article Kauffman\\_polynomial](#)

**khovanov\_polynomial** (*var1='q', var2='t', base\_ring=Integer Ring, original=False, reduced=False, odd=False, KhoHo=False*)

Return the Khovanov polynomial according to the value of column `khovanov_polynomial` for this knot or link as an instance of `LaurentPolynomial_mpair`.

## INPUT:

- `var1` – (default: 'q') the first variable
- `var2` – (default: 't') the second variable
- `base_ring` – (default: `ZZ`) the ring of the polynomial's coefficients
- `original` – boolean (default: `False`); if set to `True` the original table entry is returned as a string
- `reduced` – boolean (default: `False`); if set to `True` the reduced version of the homology is used
- `odd` – boolean (default: `False`); if set to `True` the odd version of the homology is used
- `KhoHo` – boolean (deprecated). The corresponding values have disappeared from the database since January 2024

## OUTPUT:

A Laurent polynomial over the integers, more precisely an instance of `LaurentPolynomial_mpair`. If `original` is set to `True` then a string is returned.

## Note

The data used for multi-component links were calculated with the program [KhoHo](#) which uses the `DT` notation. For knots data calculated with [KnotJob](#) are used. The latter program is more accurate in terms of orientation and reflection as it is based on `PD` code.

## EXAMPLES:

```
sage: K = KnotInfo.K6_3
sage: Kk = K.khovanov_polynomial(); Kk
q^7*t^3 + q^5*t^2 + q^3*t^2 + q^3*t + q*t + 2*q + 2*q^-1 + q^-1*t^-1
```

(continues on next page)

(continued from previous page)

```

+ q^-3*t^-1 + q^-3*t^-2 + q^-5*t^-2 + q^-7*t^-3
sage: Kk2 = K.khovanov_polynomial(var1='p', base_ring=GF(2)); Kk2
p^7*t^3 + p^5*t^3 + p^5*t^2 + p^3*t + p^-1 + p^-1*t^-1 + p^-3*t^-2 + p^-7*t^-3

sage: L = KnotInfo.L5a1_0
sage: Lk = L.khovanov_polynomial(); Lk
q^4*t^2 + t + 2 + 2*q^-2 + q^-2*t^-1 + q^-4*t^-2 + q^-6*t^-2 + q^-8*t^-3
sage: L.khovanov_polynomial(original=True)
'2 + 2/q^2 + 1/(q^8*t^3) + 1/(q^6*t^2) + 1/(q^4*t^2) + 1/(q^2*t) + t + q^4*t^
↪2'

```

```

>>> from sage.all import *
>>> K = KnotInfo.K6_3
>>> Kk = K.khovanov_polynomial(); Kk
q^7*t^3 + q^5*t^2 + q^3*t^2 + q^3*t + q*t + 2*q + 2*q^-1 + q^-1*t^-1
+ q^-3*t^-1 + q^-3*t^-2 + q^-5*t^-2 + q^-7*t^-3
>>> Kk2 = K.khovanov_polynomial(var1='p', base_ring=GF(Integer(2))); Kk2
p^7*t^3 + p^5*t^3 + p^5*t^2 + p^3*t + p^-1 + p^-1*t^-1 + p^-3*t^-2 + p^-7*t^-3

>>> L = KnotInfo.L5a1_0
>>> Lk = L.khovanov_polynomial(); Lk
q^4*t^2 + t + 2 + 2*q^-2 + q^-2*t^-1 + q^-4*t^-2 + q^-6*t^-2 + q^-8*t^-3
>>> L.khovanov_polynomial(original=True)
'2 + 2/q^2 + 1/(q^8*t^3) + 1/(q^6*t^2) + 1/(q^4*t^2) + 1/(q^2*t) + t + q^4*t^
↪2'

```

Obtaining the reduced homology (for knots only):

```

sage: Kkr = K.khovanov_polynomial(reduced=True); Kkr
q^6*t^3 + 2*q^4*t^2 + 2*q^2*t + 3 + 2*q^-2*t^-1 + 2*q^-4*t^-2 + q^-6*t^-3
sage: K.khovanov_polynomial(base_ring=QQ, reduced=True) == Kkr
True
sage: Kkr2 = K.khovanov_polynomial(base_ring=GF(2), reduced=True); Kkr2
q^6*t^3 + 1 + q^-6*t^-3
sage: KnotInfo.K8_19.inject() # optional_
↪database_knotinfo
Defining K8_19
sage: K8kr = K8_19.khovanov_polynomial(reduced=True); K8kr # optional_
↪database_knotinfo
q^16*t^5 + q^12*t^4 + q^12*t^3 + q^10*t^2 + q^6

```

```

>>> from sage.all import *
>>> Kkr = K.khovanov_polynomial(reduced=True); Kkr
q^6*t^3 + 2*q^4*t^2 + 2*q^2*t + 3 + 2*q^-2*t^-1 + 2*q^-4*t^-2 + q^-6*t^-3
>>> K.khovanov_polynomial(base_ring=QQ, reduced=True) == Kkr
True
>>> Kkr2 = K.khovanov_polynomial(base_ring=GF(Integer(2)), reduced=True); Kkr2
q^6*t^3 + 1 + q^-6*t^-3
>>> KnotInfo.K8_19.inject() # optional database_
↪knotinfo
Defining K8_19
>>> K8kr = K8_19.khovanov_polynomial(reduced=True); K8kr # optional database_

```

(continues on next page)

(continued from previous page)

```
↪knotinfo
q16*t5 + q12*t4 + q12*t3 + q10*t2 + q6
```

Obtaining the odd Khovanov homology (for knots only):

```
sage: K.khovanov_polynomial(odd=True) == Kkr
True
sage: K.khovanov_polynomial(base_ring=QQ, odd=True) == Kkr
True
sage: K.khovanov_polynomial(base_ring=GF(2), odd=True) == Kkr2
True
sage: K8ko = K8_19.khovanov_polynomial(odd=True); K8ko      # optional_
↪database_knotinfo
q16*t5 + q10*t2 + q6
sage: K8kr == K8ko                                          # optional_
↪database_knotinfo
False
```

```
>>> from sage.all import *
>>> K.khovanov_polynomial(odd=True) == Kkr
True
>>> K.khovanov_polynomial(base_ring=QQ, odd=True) == Kkr
True
>>> K.khovanov_polynomial(base_ring=GF(Integer(2)), odd=True) == Kkr2
True
>>> K8ko = K8_19.khovanov_polynomial(odd=True); K8ko      # optional database_
↪knotinfo
q16*t5 + q10*t2 + q6
>>> K8kr == K8ko                                          # optional database_
↪knotinfo
False
```

Comparison to Sage's results:

```
sage: Kk == K.link().khovanov_polynomial()
True
sage: Kk2 == K.link().khovanov_polynomial(var1='p', base_ring=GF(2))
True
sage: Lk == L.link().khovanov_polynomial()
True
```

```
>>> from sage.all import *
>>> Kk == K.link().khovanov_polynomial()
True
>>> Kk2 == K.link().khovanov_polynomial(var1='p', base_ring=GF(Integer(2)))
True
>>> Lk == L.link().khovanov_polynomial()
True
```

#### REFERENCES:

- [Wikipedia article Khovanov\\_homology](#)
- [Wikipedia article Reduced\\_homology](#)

- [ORS2013]

**knot\_atlas\_webpage** (*new=0, autoraise=True*)

Launch the Knot Atlas web-page for *self*.

INPUT:

- *new* – integer according to `open()` of `webbrowser` (0 default, 1 new window, 2 new tab)
- *autoraise* – boolean (default: `True`)

EXAMPLES:

```
sage: K = KnotInfo.K3_1
sage: K.knot_atlas_webpage()           # not tested
True
```

```
>>> from sage.all import *
>>> K = KnotInfo.K3_1
>>> K.knot_atlas_webpage()           # not tested
True
```

**knotilus\_webpage** (*new=0, autoraise=True*)

Launch the Knotilus web-page for *self*.

INPUT:

- *new* – integer according to `open()` of `webbrowser` (0 default, 1 new window, 2 new tab)
- *autoraise* – boolean (default: `True`)

EXAMPLES:

```
sage: K = KnotInfo.K3_1
sage: K.knotilus_webpage(new=1)       # not tested
True
```

```
>>> from sage.all import *
>>> K = KnotInfo.K3_1
>>> K.knotilus_webpage(new=Integer(1)) # not tested
True
```

**link** (*use\_item=<Columns.pd\_notation: ['PD Notation', <KnotInfoColumnTypes.OnlyKnots: 'K'>]>*,  
*snappy=False*)

Return *self* as an instance of `Link` or optional `spherogram.links.invariants.Link` (SnapPy).

INPUT:

- *use\_item* – (default: `self.items.pd_notation`) instance of `KnotInfoColumns` to choose the column that should be used to construct the link. Allowed values are:
  - `self.items.pd_notation`
  - `self.items.braid_notation`
  - `self.items.name` (only for `snappy=True`)
  - `self.items.dt_notation` (only for knots and `snappy=False`)
  - `self.items.gauss_notation` (only for knots and `snappy=False`)

- `snappy` – boolean (default: `False`); if set to `True` the target of the conversion is the `pip` installable package `SnapPy` (explicitely, `spherogram.links.invariants.Link`). If `SnapPy` is not installed an `ImportError` is raised. To install `SnapPy` use `sage -pip install snappy`.

**Note**

We use the PD-notation to construct `self` as default. This ensures that the number of crossings displayed in the representation string of the link coincides with the crossing number as a topological invariant.

Furthermore, note that the mirror version may depend on the used `KnotInfo`-notation. For instance, regarding to the knot `5_1` the Gauss- and the DT-notation refer to the mirror image (see example below).

**EXAMPLES:**

```
sage: K = KnotInfo.K3_1
sage: K.link()
Knot represented by 3 crossings
sage: _.braid()
s^3
sage: _ == K.braid()
True
```

```
>>> from sage.all import *
>>> K = KnotInfo.K3_1
>>> K.link()
Knot represented by 3 crossings
>>> _.braid()
s^3
>>> _ == K.braid()
True
```

using `dt_notation`:

```
sage: K.link(use_item=K.items.dt_notation)
Knot represented by 3 crossings
sage: _.braid()
s^3

sage: L = KnotInfo.L4a1_0
sage: L.link()
Link with 2 components represented by 4 crossings

sage: L.link(use_item=L.items.dt_notation)
Traceback (most recent call last):
...
ValueError: link construction using Columns.dt_notation not possible
```

```
>>> from sage.all import *
>>> K.link(use_item=K.items.dt_notation)
Knot represented by 3 crossings
>>> _.braid()
s^3
```

(continues on next page)

(continued from previous page)

```
>>> L = KnotInfo.L4a1_0
>>> L.link()
Link with 2 components represented by 4 crossings

>>> L.link(use_item=L.items.dt_notation)
Traceback (most recent call last):
...
ValueError: link construction using Columns.dt_notation not possible
```

using snappy:

```
sage: L2 = KnotInfo.L2a1_1
sage: l2 = L2.link()
sage: l2s = L2.link(snappy=True).sage_link() # optional - snappy
sage: l2 == l2s                             # optional - snappy
True
```

```
>>> from sage.all import *
>>> L2 = KnotInfo.L2a1_1
>>> l2 = L2.link()
>>> l2s = L2.link(snappy=True).sage_link() # optional - snappy
>>> l2 == l2s                             # optional - snappy
True
```

but observe:

```
sage: K7 = KnotInfo.K7_2
sage: k7s = K7.link(snappy=True); k7s      # optional - snappy
<Link: 1 comp; 7 cross>
sage: k7sn = K7.link(K7.items.name, snappy=True); k7sn # optional - snappy
<Link 7_2: 1 comp; 7 cross>
sage: k7s.sage_link().is_isotopic(k7sn)    # optional - snappy
False
sage: k7snm = k7sn.sage_link().mirror_image() # optional - snappy
sage: k7s.sage_link().is_isotopic(k7snm)   # optional - snappy
True
```

```
>>> from sage.all import *
>>> K7 = KnotInfo.K7_2
>>> k7s = K7.link(snappy=True); k7s      # optional - snappy
<Link: 1 comp; 7 cross>
>>> k7sn = K7.link(K7.items.name, snappy=True); k7sn # optional - snappy
<Link 7_2: 1 comp; 7 cross>
>>> k7s.sage_link().is_isotopic(k7sn)    # optional - snappy
False
>>> k7snm = k7sn.sage_link().mirror_image() # optional - snappy
>>> k7s.sage_link().is_isotopic(k7snm)   # optional - snappy
True
```

using braid\_notation:

```
sage: L2.link(use_item=L.items.braid_notation) == 12
True
```

```
>>> from sage.all import *
>>> L2.link(use_item=L.items.braid_notation) == 12
True
```

observe:

```
sage: L.link(use_item=L.items.braid_notation)
Link with 2 components represented by 5 crossings

sage: K6_1 = KnotInfo.K6_1
sage: K6_1.link().braid() == K6_1.braid()
False
```

```
>>> from sage.all import *
>>> L.link(use_item=L.items.braid_notation)
Link with 2 components represented by 5 crossings

>>> K6_1 = KnotInfo.K6_1
>>> K6_1.link().braid() == K6_1.braid()
False
```

also observe:

```
sage: K4_1 = KnotInfo.K4_1
sage: K4_1.link().pd_code()
[[4, 2, 5, 1], [8, 6, 1, 5], [6, 3, 7, 4], [2, 7, 3, 8]]
sage: K4_1.pd_notation()
[[4, 2, 5, 1], [8, 6, 1, 5], [6, 3, 7, 4], [2, 7, 3, 8]]

sage: K5_1 = KnotInfo.K5_1
sage: K5_1.link().braid()
s^5
sage: K5_1.link(K5_1.items.dt_notation).braid()
s^-5
sage: K5_1.link(K5_1.items.gauss_notation).braid()
s^-5
```

```
>>> from sage.all import *
>>> K4_1 = KnotInfo.K4_1
>>> K4_1.link().pd_code()
[[4, 2, 5, 1], [8, 6, 1, 5], [6, 3, 7, 4], [2, 7, 3, 8]]
>>> K4_1.pd_notation()
[[4, 2, 5, 1], [8, 6, 1, 5], [6, 3, 7, 4], [2, 7, 3, 8]]

>>> K5_1 = KnotInfo.K5_1
>>> K5_1.link().braid()
s^5
>>> K5_1.link(K5_1.items.dt_notation).braid()
s^-5
```

(continues on next page)

(continued from previous page)

```
>>> K5_1.link(K5_1.items.gauss_notation).braid()
s^-5
```

**name\_unoriented()**

Return the part of the name of `self` which is independent on the orientation.

EXAMPLES:

```
sage: KnotInfo.L10a122_1_0.name_unoriented() # optional - database_knotinfo
'L10a122'
```

```
>>> from sage.all import *
>>> KnotInfo.L10a122_1_0.name_unoriented() # optional - database_knotinfo
'L10a122'
```

**num\_components()**

Return the number of components of `self`.

EXAMPLES:

```
sage: KnotInfo.L6a1_0.num_components()
2
```

```
>>> from sage.all import *
>>> KnotInfo.L6a1_0.num_components()
2
```

**pd\_notation(original=False)**

Return the value of column `pd_notation` for this link as a Python list of Python lists. For more information type `KnotInfo.K0_1.items.pd_notation.description_webpage()`.

INPUT:

- `original` – boolean (default: `False`); if set to `True` the original table entry is returned as a string

OUTPUT:

Python list of python lists each entry of the outer list representing a crossing.

EXAMPLES:

```
sage: L = KnotInfo.L4a1_0
sage: L.pd_notation()
[[6, 1, 7, 2], [8, 3, 5, 4], [2, 5, 3, 6], [4, 7, 1, 8]]
sage: L.pd_notation(original=True)
'{{6, 1, 7, 2}, {8, 3, 5, 4}, {2, 5, 3, 6}, {4, 7, 1, 8}}'
sage: K = KnotInfo.K4_1
sage: K.pd_notation()
[[4, 2, 5, 1], [8, 6, 1, 5], [6, 3, 7, 4], [2, 7, 3, 8]]
```

```
>>> from sage.all import *
>>> L = KnotInfo.L4a1_0
>>> L.pd_notation()
[[6, 1, 7, 2], [8, 3, 5, 4], [2, 5, 3, 6], [4, 7, 1, 8]]
```

(continues on next page)

(continued from previous page)

```
>>> L.pd_notation(original=True)
'{{6, 1, 7, 2}, {8, 3, 5, 4}, {2, 5, 3, 6}, {4, 7, 1, 8}}'
>>> K = KnotInfo.K4_1
>>> K.pd_notation()
[[4, 2, 5, 1], [8, 6, 1, 5], [6, 3, 7, 4], [2, 7, 3, 8]]
```

**series** (*oriented=False*)Return the series of links *self* belongs to.

INPUT:

- *oriented* – boolean (default: *False*); it only affects proper links. By default the items of the series will be again series of links collecting all orientation mutants of an unoriented name. To obtain the series of the individual links this keyword has to be set to *True*.

EXAMPLES:

```
sage: K5 = KnotInfo.K5_2.series()
sage: K5(1)
<KnotInfo.K5_1: '5_1'>
sage: KnotInfo.L4a1_1.series().inject()
Defining L4a
sage: L4a(1)
Series of links L4a1
sage: KnotInfo.L4a1_1.series(oriented=True).inject()
Defining L4a1
sage: L4a(1) == L4a1
True
sage: L4a1(1)
<KnotInfo.L4a1_1: 'L4a1{1}'>
```

```
>>> from sage.all import *
>>> K5 = KnotInfo.K5_2.series()
>>> K5(Integer(1))
<KnotInfo.K5_1: '5_1'>
>>> KnotInfo.L4a1_1.series().inject()
Defining L4a
>>> L4a(Integer(1))
Series of links L4a1
>>> KnotInfo.L4a1_1.series(oriented=True).inject()
Defining L4a1
>>> L4a(Integer(1)) == L4a1
True
>>> L4a1(Integer(1))
<KnotInfo.L4a1_1: 'L4a1{1}'>
```

**signature** ()Return the signature of *self*.

From the KnotInfo description page:

The signature of a knot,  $\sigma(K)$ , is equal to  $\sigma(V + V^t)$ , the signature of  $V + V^t$  where  $V$  is a Seifert matrix for the knot and  $V^t$  is its transpose.

To read the complete description type `KnotInfo.K0_1.items.signatur.description_webpage()`.

EXAMPLES:

```
sage: KnotInfo.K5_2.signature()      # optional - database_knotinfo
-2
```

```
>>> from sage.all import *
>>> KnotInfo.K5_2.signature()      # optional - database_knotinfo
-2
```

**symmetry\_type()**Return the symmetry type of `self`.

From the KnotInfo description page:

If a knot is viewed as the oriented diffeomorphism class of an oriented pair,  $K = (S_3, S_1)$ , with  $S_i$  diffeomorphic to  $S^i$ , there are four oriented knots associated to any particular knot  $K$ . In addition to  $K$  itself, there is the reverse,  $K^r = (S_3, -S_1)$ , the concordance inverse,  $-K = (-S_3, -S_1)$ , and the mirror image,  $K^m = (-S_3, S_1)$ . A knot is called reversible if  $K = K^r$ , negative amphicheiral if  $K = -K$ , and positive amphicheiral if  $K = K^m$ .

A knot possessing any two of these types of symmetry has all three. Thus, in the table, a knot is called reversible if that is the only type of symmetry it has, and likewise for negative amphicheiral. If it has none of these types of symmetry it is called chiral, and if it has all three it is called fully amphicheiral.

For prime knots with fewer than 12 crossings, all amphicheiral knots are negative amphicheiral.

EXAMPLES:

```
sage: KnotInfo.K6_1.series().inject()
Defining K6
sage: [(K.name, K.symmetry_type()) for K in K6]
[('K6_1', 'reversible'),
 ('K6_2', 'reversible'),
 ('K6_3', 'fully amphicheiral')]
```

```
>>> from sage.all import *
>>> KnotInfo.K6_1.series().inject()
Defining K6
>>> [(K.name, K.symmetry_type()) for K in K6]
[('K6_1', 'reversible'),
 ('K6_2', 'reversible'),
 ('K6_3', 'fully amphicheiral')]
```

**three\_genus()**Return the three genus of `self`.

From the KnotInfo description page:

The three-genus of a knot is defined to be the minimal genus of a Seifert surface for a knot.

To read the complete description type `KnotInfo.K0_1.items.three_genus.description_webpage()`.

EXAMPLES:

```
sage: KnotInfo.K5_2.three_genus()  # optional - database_knotinfo
1
```

```
>>> from sage.all import *
>>> KnotInfo.K5_2.three_genus()      # optional - database_knotinfo
1
```

Note that this differs from the corresponding result in Sage since the latter is obtained for a Seifert surface that does not have the minimal genus:

```
sage: KnotInfo.K5_2.link().genus()
3
```

```
>>> from sage.all import *
>>> KnotInfo.K5_2.link().genus()
3
```

```
class sage.knots.knotinfo.KnotInfoSeries(crossing_number, is_knot, is_alternating,
                                           name_unoriented=None)
```

Bases: UniqueRepresentation, SageObject

This class can be used to access knots and links via their index according to the series they belong to.

INPUT:

- `crossing_number` – integer giving the crossing number of this series of links
- `is_knot` – boolean; whether this series is a series of knots or proper links
- `is_alternating` – boolean; whether this series is restricted to alternating links or not This is not relevant for knots with less than 11 crossings
- `name_unoriented` – string restricting the series to all links with that `name_unoriented`

EXAMPLES:

```
sage: from sage.knots.knotinfo import KnotInfoSeries
sage: K6 = KnotInfoSeries(6, True, True); K6
Series of knots K6
sage: K6(3)
<KnotInfo.K6_3: '6_3'>
sage: list(K6)
[<KnotInfo.K6_1: '6_1'>, <KnotInfo.K6_2: '6_2'>, <KnotInfo.K6_3: '6_3'>]
sage: L6a = KnotInfoSeries(6, False, True); L6a
Series of links L6a
sage: L6a(2)
Series of links L6a2
sage: _.inject()
Defining L6a2
sage: list(L6a2)
[<KnotInfo.L6a2_0: 'L6a2{0}'>, <KnotInfo.L6a2_1: 'L6a2{1}'>]
sage: L6a2(0).series() == L6a
True
sage: L6a2(0) == L6a2('0')
True
```

```
>>> from sage.all import *
>>> from sage.knots.knotinfo import KnotInfoSeries
>>> K6 = KnotInfoSeries(Integer(6), True, True); K6
```

(continues on next page)

(continued from previous page)

```

Series of knots K6
>>> K6(Integer(3))
<KnotInfo.K6_3: '6_3'>
>>> list(K6)
[<KnotInfo.K6_1: '6_1'>, <KnotInfo.K6_2: '6_2'>, <KnotInfo.K6_3: '6_3'>]
>>> L6a = KnotInfoSeries(Integer(6), False, True); L6a
Series of links L6a
>>> L6a(Integer(2))
Series of links L6a2
>>> _.inject()
Defining L6a2
>>> list(L6a2)
[<KnotInfo.L6a2_0: 'L6a2{0}'>, <KnotInfo.L6a2_1: 'L6a2{1}'>]
>>> L6a2(Integer(0)).series() == L6a
True
>>> L6a2(Integer(0)) == L6a2('0')
True

```

**inject** (*verbose=True*)

Inject self with its name into the namespace of the Python code from which this function is called.

INPUT:

- *verbose* – boolean (default: True); to suppress the message printed on the invocation

EXAMPLES:

```

sage: KnotInfoSeries(6, True, True).inject()
Defining K6
sage: K6(2)
<KnotInfo.K6_2: '6_2'>

```

```

>>> from sage.all import *
>>> KnotInfoSeries(Integer(6), True, True).inject()
Defining K6
>>> K6(Integer(2))
<KnotInfo.K6_2: '6_2'>

```

**is\_recoverable** (*unique=True, max\_samples=8*)

Return if all items of self can be recovered from its conversion to Sage links using the *pd\_notation* and the *braid\_notation* and their mirror images.

The method is indirectly used by the TestSuite.

INPUT:

- *unique* – boolean (default: True); see [KnotInfoBase.is\\_recoverable\(\)](#)
- *max\_samples* – nonnegative integer or infinity (default: 8); limits the number of items to check (random sample). If set to infinity then no limit is set.

EXAMPLES:

```

sage: KnotInfo.L4a1_0.series().inject()
Defining L4a
sage: L4a.is_recoverable()

```

(continues on next page)

(continued from previous page)

```

True
sage: L4a.is_recoverable(unique=False)
True
sage: KnotInfo.L5a1_0.series().inject()
Defining L5a
sage: L5a.is_recoverable()
False
sage: L5a.is_recoverable(unique=False)
True

```

```

>>> from sage.all import *
>>> KnotInfo.L4a1_0.series().inject()
Defining L4a
>>> L4a.is_recoverable()
True
>>> L4a.is_recoverable(unique=False)
True
>>> KnotInfo.L5a1_0.series().inject()
Defining L5a
>>> L5a.is_recoverable()
False
>>> L5a.is_recoverable(unique=False)
True

```

**list** (*oriented=False, comp=None, det=None, homfly=None*)

Return this series as a Python list.

INPUT:

- *oriented* – boolean (default: `False`); it only affects series of proper links. By default the list items of a series of proper links are again series of links collecting all orientation types of an unoriented name. To obtain the list of the individual links this keyword has to be set to `True`.
- *comp* – (default: `None`) if given an integer for this keyword the list is restricted to links having the according number of components. This keyword implies *oriented=True*.
- *det* – (default: `None`) if given an integer for this keyword the list is restricted to links having the according value for its determinant. This keyword implies *oriented=True*.
- *homfly* – (default: `None`) if given a HOMFLY-PT polynomial having *normalization='vz'* for this keyword the list is restricted to links having the according value for its HOMFLY-PT polynomial. This keyword implies *oriented=True*.

EXAMPLES:

```

sage: from sage.knots.knotinfo import KnotInfoSeries
sage: K6 = KnotInfoSeries(6, True, True); K6
Series of knots K6
sage: K6.list()
[<KnotInfo.K6_1: '6_1'>, <KnotInfo.K6_2: '6_2'>, <KnotInfo.K6_3: '6_3'>]
sage: KnotInfoSeries(2, False, True).inject()
Defining L2a
sage: L2a.list()
[Series of links L2a1]

```

(continues on next page)

(continued from previous page)

```

sage: L2a.list(oriented=True)
[<KnotInfo.L2a1_0: 'L2a1{0}'>, <KnotInfo.L2a1_1: 'L2a1{1}'>]

>>> from sage.all import *
>>> from sage.knots.knotinfo import KnotInfoSeries
>>> K6 = KnotInfoSeries(Integer(6), True, True); K6
Series of knots K6
>>> K6.list()
[<KnotInfo.K6_1: '6_1'>, <KnotInfo.K6_2: '6_2'>, <KnotInfo.K6_3: '6_3'>]
>>> KnotInfoSeries(Integer(2), False, True).inject()
Defining L2a
>>> L2a.list()
[Series of links L2a1]
>>> L2a.list(oriented=True)
[<KnotInfo.L2a1_0: 'L2a1{0}'>, <KnotInfo.L2a1_1: 'L2a1{1}'>]

```

**lower\_list** (*oriented=False, comp=None, det=None, homfly=None*)

Return this series together with all series with smaller crossing number as a Python list.

INPUT:

- *oriented* – boolean (default: False); see the description for *list()*
- *comp* – (default: None) see the description for *list()*
- *det* – (default: None) see the description for *list()*
- *homfly* – (default: None) see the description for *list()*

EXAMPLES:

```

sage: from sage.knots.knotinfo import KnotInfoSeries
sage: KnotInfoSeries(5, True, True).lower_list()
[<KnotInfo.K0_1: '0_1'>,
 <KnotInfo.K3_1: '3_1'>,
 <KnotInfo.K4_1: '4_1'>,
 <KnotInfo.K5_1: '5_1'>,
 <KnotInfo.K5_2: '5_2'>]
sage: KnotInfoSeries(4, False, True).lower_list()
[Series of links L2a1, Series of links L4a1]
sage: KnotInfoSeries(4, False, True).lower_list(oriented=True)
[<KnotInfo.L2a1_0: 'L2a1{0}'>,
 <KnotInfo.L2a1_1: 'L2a1{1}'>,
 <KnotInfo.L4a1_0: 'L4a1{0}'>,
 <KnotInfo.L4a1_1: 'L4a1{1}'>]

```

```

>>> from sage.all import *
>>> from sage.knots.knotinfo import KnotInfoSeries
>>> KnotInfoSeries(Integer(5), True, True).lower_list()
[<KnotInfo.K0_1: '0_1'>,
 <KnotInfo.K3_1: '3_1'>,
 <KnotInfo.K4_1: '4_1'>,
 <KnotInfo.K5_1: '5_1'>,
 <KnotInfo.K5_2: '5_2'>]

```

(continues on next page)

(continued from previous page)

```
>>> KnotInfoSeries(Integer(4), False, True).lower_list()
[Series of links L2a1, Series of links L4a1]
>>> KnotInfoSeries(Integer(4), False, True).lower_list(oriented=True)
[<KnotInfo.L2a1_0: 'L2a1{0}'>,
 <KnotInfo.L2a1_1: 'L2a1{1}'>,
 <KnotInfo.L4a1_0: 'L4a1{0}'>,
 <KnotInfo.L4a1_1: 'L4a1{1}'>]
```

```
class sage.knots.knotinfo.SymmetryMutant (value, names=<not given>, *values, module=None,
                                           qualname=None, type=None, start=1, boundary=None)
```

Bases: Enum

Enum to specify the symmetry mutant link of the prime link listed in the KnotInfo and LinkInfo databases. From the KnotInfo description page:

If a knot is viewed as the oriented diffeomorphism class of an oriented pair,  $K = (S_3, S_1)$ , with  $S_i$  diffeomorphic to  $S^i$ , there are four oriented knots associated to any particular knot  $K$ . In addition to  $K$  itself, there is the reverse,  $K^r = (S_3, -S_1)$ , the concordance inverse,  $-K = (-S_3, -S_1)$ , and the mirror image,  $K^m = (-S_3, S_1)$ .

```
concordance_inverse = 'c'
```

```
is_minimal (link)
```

Return whether `self` is minimal among its matching mutants.

EXAMPLES:

```
sage: from sage.knots.knotinfo import SymmetryMutant
sage: SymmetryMutant.itself.is_minimal(KnotInfo.K6_1)
True
sage: SymmetryMutant.concordance_inverse.is_minimal(KnotInfo.K6_1)
False
```

```
>>> from sage.all import *
>>> from sage.knots.knotinfo import SymmetryMutant
>>> SymmetryMutant.itself.is_minimal(KnotInfo.K6_1)
True
>>> SymmetryMutant.concordance_inverse.is_minimal(KnotInfo.K6_1)
False
```

```
itself = 's'
```

```
matches (link)
```

Return the list of other symmetry mutants that give isotopic links with respect to `link` and `self`. For `self` is `SymmetryMutant.unknown` a boolean is returned which is `True` if the chirality of `link` is unknown.

EXAMPLES:

```
sage: from sage.knots.knotinfo import SymmetryMutant
sage: SymmetryMutant.itself.matches(KnotInfo.K6_1)
[<SymmetryMutant.reverse: 'r'>]
sage: SymmetryMutant.mirror_image.matches(KnotInfo.K6_1)
[<SymmetryMutant.concordance_inverse: 'c'>]
```

```
>>> from sage.all import *
>>> from sage.knots.knotinfo import SymmetryMutant
>>> SymmetryMutant.itself.matches(KnotInfo.K6_1)
[<SymmetryMutant.reverse: 'r'>]
>>> SymmetryMutant.mirror_image.matches(KnotInfo.K6_1)
[<SymmetryMutant.concordance_inverse: 'c'>]
```

**mir()**

Return the mirror image of self.

EXAMPLES:

```
sage: from sage.knots.knotinfo import SymmetryMutant
sage: all( sym.mir().mir() == sym for sym in SymmetryMutant)
True
```

```
>>> from sage.all import *
>>> from sage.knots.knotinfo import SymmetryMutant
>>> all( sym.mir().mir() == sym for sym in SymmetryMutant)
True
```

**mirror\_image = 'm'**

**mixed = 'x'**

**rev()**

Return the reverse of self.

EXAMPLES:

```
sage: from sage.knots.knotinfo import SymmetryMutant
sage: all( sym.rev().rev() == sym for sym in SymmetryMutant)
True
```

```
>>> from sage.all import *
>>> from sage.knots.knotinfo import SymmetryMutant
>>> all( sym.rev().rev() == sym for sym in SymmetryMutant)
True
```

**reverse = 'r'**

**unknown = '?'**

`sage.knots.knotinfo.eval_knotinfo(string, locals={}, to_tuple=True)`

Parse a string from the KnotInfo database and evaluate it by `sage_eval`.

INPUT:

- `string` – string that gives a value of some database entry
- `locals` – dictionary of locals passed to `sage_eval`

EXAMPLES:

```

sage: from sage.knots.knotinfo import eval_knotinfo
sage: L = KnotInfo.L4a1_0
sage: L.braid_notation(original=True)
'{3, {-2, -2, -1, 2, -1}}'
sage: eval_knotinfo(_)
(3, (-2, -2, -1, 2, -1))
sage: KnotInfo.K13a_1.kauffman_polynomial() # optional - database_knotinfo #_
↳indirect doctest
Traceback (most recent call last):
...
NotImplementedError: this value is not provided by the database

```

```

>>> from sage.all import *
>>> from sage.knots.knotinfo import eval_knotinfo
>>> L = KnotInfo.L4a1_0
>>> L.braid_notation(original=True)
'{3, {-2, -2, -1, 2, -1}}'
>>> eval_knotinfo(_)
(3, (-2, -2, -1, 2, -1))
>>> KnotInfo.K13a_1.kauffman_polynomial() # optional - database_knotinfo #_
↳indirect doctest
Traceback (most recent call last):
...
NotImplementedError: this value is not provided by the database

```

`sage.knots.knotinfo.knotinfo_bool(string)`

Prepare a string from the KnotInfo database representing a boolean.

INPUT:

- string – string that gives a value of some database entry

EXAMPLES:

```

sage: from sage.knots.knotinfo import knotinfo_bool
sage: knotinfo_bool('Y')
True
sage: KnotInfo.K13a_1.is_almost_alternating() # optional - database_knotinfo #_
↳indirect doctest
Traceback (most recent call last):
...
NotImplementedError: this boolean is not provided by the database

```

```

>>> from sage.all import *
>>> from sage.knots.knotinfo import knotinfo_bool
>>> knotinfo_bool('Y')
True
>>> KnotInfo.K13a_1.is_almost_alternating() # optional - database_knotinfo #_
↳indirect doctest
Traceback (most recent call last):
...
NotImplementedError: this boolean is not provided by the database

```

`sage.knots.knotinfo.knotinfo_int(string)`

Preparse a string from the KnotInfo database representing an integer.

INPUT:

- string – string that gives a value of some database entry

EXAMPLES:

```
sage: from sage.knots.knotinfo import knotinfo_int
sage: knotinfo_int('7')
7
sage: KnotInfo.K13a_1.braid_index() # optional - database_knotinfo # indirect_
↪doctest
Traceback (most recent call last):
...
NotImplementedError: this integer is not provided by the database
```

```
>>> from sage.all import *
>>> from sage.knots.knotinfo import knotinfo_int
>>> knotinfo_int('7')
7
>>> KnotInfo.K13a_1.braid_index() # optional - database_knotinfo # indirect_
↪doctest
Traceback (most recent call last):
...
NotImplementedError: this integer is not provided by the database
```



## FREE MONOID GENERATED BY PRIME KNOTS AVAILABLE VIA THE

*KnotInfoBase* class.

A generator of this free abelian monoid is a prime knot according to the list at [KnotInfo](#). A fully amphicheiral prime knot is represented by exactly one generator with the corresponding name. For non-chiral prime knots, there are additionally one or three generators with the suffixes *m*, *r* and *c* which specify the mirror and reverse images according to their symmetry type.

AUTHORS:

- Sebastian Oehms June 2024: initial version

```
class sage.knots.free_knotinfo_monoid.FreeKnotInfoMonoid(max_crossing_number, category=None,
                                                         prefix=None, **kws)
```

Bases: *IndexedFreeAbelianMonoid*

Initialize *self* with generators belonging to prime knots with at most *max\_crossing\_number* crossings.

**Element**

alias of *FreeKnotInfoMonoidElement*

**from\_knot** (*knot*, *unique=True*)

Create an element of this abelian monoid from *knot*.

INPUT:

- *knot* – an instance of *Knot*
- *unique* – boolean (default is *True*). This only affects the case where a unique identification is not possible. If set to *False* you can obtain a matching list (see explanation of the output below)

OUTPUT:

An instance of the element class of *self* per default. If the keyword argument *unique* then a list of such instances is returned.

EXAMPLES:

```
sage: from sage.knots.free_knotinfo_monoid import FreeKnotInfoMonoid
sage: FKIM = FreeKnotInfoMonoid()
sage: K = KnotInfo.K5_1.link().mirror_image()
sage: FKIM.from_knot(K)
KnotInfo['K5_1m']

sage: # optional - database_knotinfo
sage: K = Knot(KnotInfo.K9_12.braid())
sage: FKIM.from_knot(K)                                # long time
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
NotImplementedError: this (possibly non prime) knot cannot be
identified uniquely by KnotInfo
use keyword argument `unique` to obtain more details
sage: FKIM.from_knot(K, unique=False)      # long time
[KnotInfo['K4_1']*KnotInfo['K5_2'], KnotInfo['K9_12']]
```

```
>>> from sage.all import *
>>> from sage.knots.free_knotinfo_monoid import FreeKnotInfoMonoid
>>> FKIM = FreeKnotInfoMonoid()
>>> K = KnotInfo.K5_1.link().mirror_image()
>>> FKIM.from_knot(K)
KnotInfo['K5_1m']

>>> # optional - database_knotinfo
>>> K = Knot(KnotInfo.K9_12.braid())
>>> FKIM.from_knot(K)      # long time
Traceback (most recent call last):
...
NotImplementedError: this (possibly non prime) knot cannot be
identified uniquely by KnotInfo
use keyword argument `unique` to obtain more details
>>> FKIM.from_knot(K, unique=False)      # long time
[KnotInfo['K4_1']*KnotInfo['K5_2'], KnotInfo['K9_12']]
```

**inject\_variables** (*select=None, verbose=True*)

Inject self with its name into the namespace of the Python code from which this function is called.

INPUT:

- *select* – instance of *KnotInfoBase*, *KnotInfoSeries* or an integer. In all cases the input is used to restrict the injected generators to the according subset (number of crossings in the case of integer)
- *verbose* – boolean (optional, default *True*) to suppress the message printed on the invocation

EXAMPLES:

```
sage: from sage.knots.free_knotinfo_monoid import FreeKnotInfoMonoid
sage: FKIM = FreeKnotInfoMonoid(5)
sage: FKIM.inject_variables(select=3)
Defining K3_1
Defining K3_1m
sage: FKIM.inject_variables(select=KnotInfo.K5_2)
Defining K5_2
Defining K5_2m
sage: FKIM.inject_variables(select=KnotInfo.K5_2.series())
Defining K5_1
Defining K5_1m
sage: FKIM.inject_variables()
Defining K0_1
Defining K4_1
```

```

>>> from sage.all import *
>>> from sage.knots.free_knotinfo_monoid import FreeKnotInfoMonoid
>>> FKIM = FreeKnotInfoMonoid(Integer(5))
>>> FKIM.inject_variables(select=Integer(3))
Defining K3_1
Defining K3_1m
>>> FKIM.inject_variables(select=KnotInfo.K5_2)
Defining K5_2
Defining K5_2m
>>> FKIM.inject_variables(select=KnotInfo.K5_2.series())
Defining K5_1
Defining K5_1m
>>> FKIM.inject_variables()
Defining K0_1
Defining K4_1

```

**class** sage.knots.free\_knotinfo\_monoid.**FreeKnotInfoMonoidElement** ( $F, x$ )

Bases: `IndexedFreeAbelianMonoidElement`

An element of an indexed free abelian monoid.

**as\_knot** ()

Return the knot represented by `self`.

EXAMPLES:

```

sage: from sage.knots.free_knotinfo_monoid import FreeKnotInfoMonoid
sage: FKIM = FreeKnotInfoMonoid()
sage: FKIM.inject_variables(select=3)
Defining K3_1
Defining K3_1m
sage: K = K3_1^2 * K3_1m
sage: K.as_knot()
Knot represented by 9 crossings

```

```

>>> from sage.all import *
>>> from sage.knots.free_knotinfo_monoid import FreeKnotInfoMonoid
>>> FKIM = FreeKnotInfoMonoid()
>>> FKIM.inject_variables(select=Integer(3))
Defining K3_1
Defining K3_1m
>>> K = K3_1**Integer(2) * K3_1m
>>> K.as_knot()
Knot represented by 9 crossings

```

**to\_knotinfo** ()

Return a word representing `self` as a list of pairs.

Each pair ( $ki$ ,  $sym$ ) consists of a *KnotInfoBase* instance  $ki$  and *SymmetryMutant* instance  $sym$ .

EXAMPLES:

```

sage: from sage.knots.free_knotinfo_monoid import FreeKnotInfoMonoid
sage: FKIM = FreeKnotInfoMonoid()
sage: FKIM.inject_variables(select=3)

```

(continues on next page)

(continued from previous page)

```
Defining K3_1
Defining K3_1m
sage: K = K3_1^2 * K3_1m
sage: K.to_knotinfo()
[(<KnotInfo.K3_1: '3_1'>, <SymmetryMutant.itself: 's'>),
 (<KnotInfo.K3_1: '3_1'>, <SymmetryMutant.itself: 's'>),
 (<KnotInfo.K3_1: '3_1'>, <SymmetryMutant.mirror_image: 'm'>)]
```

```
>>> from sage.all import *
>>> from sage.knots.free_knotinfo_monoid import FreeKnotInfoMonoid
>>> FKIM = FreeKnotInfoMonoid()
>>> FKIM.inject_variables(select=Integer(3))
Defining K3_1
Defining K3_1m
>>> K = K3_1**Integer(2) * K3_1m
>>> K.to_knotinfo()
[(<KnotInfo.K3_1: '3_1'>, <SymmetryMutant.itself: 's'>),
 (<KnotInfo.K3_1: '3_1'>, <SymmetryMutant.itself: 's'>),
 (<KnotInfo.K3_1: '3_1'>, <SymmetryMutant.mirror_image: 'm'>)]
```

## INDICES AND TABLES

- [Index](#)
- [Module Index](#)
- [Search Page](#)



## PYTHON MODULE INDEX

### k

`sage.knots.free_knotinfo_monoid`, [125](#)

`sage.knots.knot`, [1](#)

`sage.knots.knotinfo`, [77](#)

`sage.knots.link`, [11](#)



## A

`alexander_polynomial()` (*sage.knots.knotinfo.KnotInfoBase method*), 84  
`alexander_polynomial()` (*sage.knots.link.Link method*), 18  
`an_element()` (*sage.knots.knot.Knots method*), 6  
`arcs()` (*sage.knots.link.Link method*), 20  
`arf_invariant()` (*sage.knots.knot.Knot method*), 2  
`as_knot()` (*sage.knots.free\_knotinfo\_monoid.FreeKnotInfoMonoidElement method*), 127

## B

`braid()` (*sage.knots.knotinfo.KnotInfoBase method*), 86  
`braid()` (*sage.knots.link.Link method*), 21  
`braid_index()` (*sage.knots.knotinfo.KnotInfoBase method*), 86  
`braid_length()` (*sage.knots.knotinfo.KnotInfoBase method*), 87  
`braid_notation()` (*sage.knots.knotinfo.KnotInfoBase method*), 87

## C

`colored_jones_polynomial()` (*sage.knots.knot.Knot method*), 3  
`coloring_maps()` (*sage.knots.link.Link method*), 22  
`colorings()` (*sage.knots.link.Link method*), 24  
`concordance_inverse` (*sage.knots.knotinfo.Symmetry-Mutant attribute*), 120  
`connected_sum()` (*sage.knots.knot.Knot method*), 4  
`conway_polynomial()` (*sage.knots.knotinfo.KnotInfoBase method*), 88  
`conway_polynomial()` (*sage.knots.link.Link method*), 25  
`cosmetic_crossing_conjecture_verified()` (*sage.knots.knotinfo.KnotInfoBase method*), 89  
`crossing_number()` (*sage.knots.knotinfo.KnotInfoBase method*), 89

## D

`determinant()` (*sage.knots.knotinfo.KnotInfoBase method*), 90  
`determinant()` (*sage.knots.link.Link method*), 26

`diagram()` (*sage.knots.knotinfo.KnotInfoBase method*), 90  
`dowker_notation()` (*sage.knots.link.Link method*), 27  
`dt_code()` (*sage.knots.knot.Knot method*), 5  
`dt_notation()` (*sage.knots.knotinfo.KnotInfoBase method*), 91

## E

`Element` (*sage.knots.free\_knotinfo\_monoid.FreeKnotInfoMonoid attribute*), 125  
`Element` (*sage.knots.knot.Knots attribute*), 6  
`eval_knotinfo()` (*in module sage.knots.knotinfo*), 121

## F

`FreeKnotInfoMonoid` (*class in sage.knots.free\_knotinfo\_monoid*), 125  
`FreeKnotInfoMonoidElement` (*class in sage.knots.free\_knotinfo\_monoid*), 127  
`from_dowker_code()` (*sage.knots.knot.Knots method*), 7  
`from_gauss_code()` (*sage.knots.knot.Knots method*), 8  
`from_knot()` (*sage.knots.free\_knotinfo\_monoid.FreeKnotInfoMonoid method*), 125  
`from_table()` (*sage.knots.knot.Knots method*), 8  
`fundamental_group()` (*sage.knots.link.Link method*), 28

## G

`gauss_code()` (*sage.knots.link.Link method*), 29  
`gauss_notation()` (*sage.knots.knotinfo.KnotInfoBase method*), 91  
`genus()` (*sage.knots.link.Link method*), 30  
`get_knotinfo()` (*sage.knots.link.Link method*), 31

## H

`homfly_polynomial()` (*sage.knots.knotinfo.KnotInfoBase method*), 92  
`homfly_polynomial()` (*sage.knots.link.Link method*), 39

## I

`inject()` (*sage.knots.knotinfo.KnotInfoBase method*), 93  
`inject()` (*sage.knots.knotinfo.KnotInfoSeries method*), 117

- `inject_variables()` (*sage.knots.free\_knotinfo\_monoid.FreeKnotInfoMonoid* method), 126  
`is_adequate()` (*sage.knots.knotinfo.KnotInfoBase* method), 94  
`is_almost_alternating()` (*sage.knots.knotinfo.KnotInfoBase* method), 94  
`is_alternating()` (*sage.knots.knotinfo.KnotInfoBase* method), 94  
`is_alternating()` (*sage.knots.link.Link* method), 42  
`is_amphicheiral()` (*sage.knots.knotinfo.KnotInfoBase* method), 94  
`is_colorable()` (*sage.knots.link.Link* method), 43  
`is_fibered()` (*sage.knots.knotinfo.KnotInfoBase* method), 95  
`is_hyperbolic()` (*sage.knots.knotinfo.KnotInfoBase* method), 96  
`is_isotopic()` (*sage.knots.link.Link* method), 45  
`is_knot()` (*sage.knots.knotinfo.KnotInfoBase* method), 96  
`is_knot()` (*sage.knots.link.Link* method), 47  
`is_minimal()` (*sage.knots.knotinfo.SymmetryMutant* method), 120  
`is_oriented()` (*sage.knots.knotinfo.KnotInfoBase* method), 96  
`is_positive()` (*sage.knots.knotinfo.KnotInfoBase* method), 96  
`is_positive_braid()` (*sage.knots.knotinfo.KnotInfoBase* method), 97  
`is_quasi_alternating()` (*sage.knots.knotinfo.KnotInfoBase* method), 97  
`is_quasipositive()` (*sage.knots.knotinfo.KnotInfoBase* method), 97  
`is_recoverable()` (*sage.knots.knotinfo.KnotInfoBase* method), 97  
`is_recoverable()` (*sage.knots.knotinfo.KnotInfoSeries* method), 117  
`is_reversible()` (*sage.knots.knotinfo.KnotInfoBase* method), 98  
`is_strongly_quasipositive()` (*sage.knots.knotinfo.KnotInfoBase* method), 98  
`is_unique()` (*sage.knots.knotinfo.KnotInfoBase* method), 98  
`items` (*sage.knots.knotinfo.KnotInfoBase* property), 99  
`itself` (*sage.knots.knotinfo.SymmetryMutant* attribute), 120
- ## J
- `jones_polynomial()` (*sage.knots.knotinfo.KnotInfoBase* method), 100  
`jones_polynomial()` (*sage.knots.link.Link* method), 47
- ## K
- `K0_1` (*sage.knots.knotinfo.KnotInfo* attribute), 83  
`K3_1` (*sage.knots.knotinfo.KnotInfo* attribute), 83  
`K4_1` (*sage.knots.knotinfo.KnotInfo* attribute), 83  
`K5_1` (*sage.knots.knotinfo.KnotInfo* attribute), 83  
`K5_2` (*sage.knots.knotinfo.KnotInfo* attribute), 83  
`K6_1` (*sage.knots.knotinfo.KnotInfo* attribute), 83  
`K6_2` (*sage.knots.knotinfo.KnotInfo* attribute), 83  
`K6_3` (*sage.knots.knotinfo.KnotInfo* attribute), 83  
`K7_1` (*sage.knots.knotinfo.KnotInfo* attribute), 83  
`K7_2` (*sage.knots.knotinfo.KnotInfo* attribute), 83  
`kauffman_polynomial()` (*sage.knots.knotinfo.KnotInfoBase* method), 104  
`khovanov_homology()` (*sage.knots.link.Link* method), 52  
`khovanov_polynomial()` (*sage.knots.knotinfo.KnotInfoBase* method), 106  
`khovanov_polynomial()` (*sage.knots.link.Link* method), 53  
`Knot` (class in *sage.knots.knot*), 1  
`knot_atlas_webpage()` (*sage.knots.knotinfo.KnotInfoBase* method), 109  
`knotilus_webpage()` (*sage.knots.knotinfo.KnotInfoBase* method), 109  
`KnotInfo` (class in *sage.knots.knotinfo*), 83  
`knotinfo_bool()` (in module *sage.knots.knotinfo*), 122  
`knotinfo_int()` (in module *sage.knots.knotinfo*), 122  
`KnotInfoBase` (class in *sage.knots.knotinfo*), 84  
`KnotInfoSeries` (class in *sage.knots.knotinfo*), 116  
`Knots` (class in *sage.knots.knot*), 6
- ## L
- `L2a1_0` (*sage.knots.knotinfo.KnotInfo* attribute), 83  
`L2a1_1` (*sage.knots.knotinfo.KnotInfo* attribute), 83  
`L4a1_0` (*sage.knots.knotinfo.KnotInfo* attribute), 84  
`L4a1_1` (*sage.knots.knotinfo.KnotInfo* attribute), 84  
`L5a1_0` (*sage.knots.knotinfo.KnotInfo* attribute), 84  
`L5a1_1` (*sage.knots.knotinfo.KnotInfo* attribute), 84  
`L6a1_0` (*sage.knots.knotinfo.KnotInfo* attribute), 84  
`L6a1_1` (*sage.knots.knotinfo.KnotInfo* attribute), 84  
`L6a2_0` (*sage.knots.knotinfo.KnotInfo* attribute), 84  
`L6a2_1` (*sage.knots.knotinfo.KnotInfo* attribute), 84  
`Link` (class in *sage.knots.link*), 11  
`link()` (*sage.knots.knotinfo.KnotInfoBase* method), 109  
`links_gould_polynomial()` (*sage.knots.link.Link* method), 55  
`list()` (*sage.knots.knotinfo.KnotInfoSeries* method), 118  
`lower_list()` (*sage.knots.knotinfo.KnotInfoSeries* method), 119
- ## M
- `matches()` (*sage.knots.knotinfo.SymmetryMutant* method), 120  
`mir()` (*sage.knots.knotinfo.SymmetryMutant* method), 121  
`mirror_image` (*sage.knots.knotinfo.SymmetryMutant* attribute), 121  
`mirror_image()` (*sage.knots.link.Link* method), 55

`mixed` (*sage.knots.knotinfo.SymmetryMutant attribute*), [SymmetryMutant \(class in sage.knots.knotinfo\)](#), [120](#)  
[121](#)

`module`

`sage.knots.free_knotinfo_monoid`, [125](#)  
`sage.knots.knot`, [1](#)  
`sage.knots.knotinfo`, [77](#)  
`sage.knots.link`, [11](#)

## N

`name_unoriented()` (*sage.knots.knotinfo.KnotInfoBase method*), [113](#)  
`num_components()` (*sage.knots.knotinfo.KnotInfoBase method*), [113](#)  
`number_of_components()` (*sage.knots.link.Link method*), [57](#)

## O

`omega_signature()` (*sage.knots.link.Link method*), [58](#)  
`one()` (*sage.knots.knot.Knots method*), [10](#)  
`orientation()` (*sage.knots.link.Link method*), [59](#)  
`oriented_gauss_code()` (*sage.knots.link.Link method*), [59](#)

## P

`pd_code()` (*sage.knots.link.Link method*), [60](#)  
`pd_notation()` (*sage.knots.knotinfo.KnotInfoBase method*), [113](#)  
`plot()` (*sage.knots.link.Link method*), [62](#)

## R

`regions()` (*sage.knots.link.Link method*), [68](#)  
`remove_loops()` (*sage.knots.link.Link method*), [70](#)  
`rev()` (*sage.knots.knotinfo.SymmetryMutant method*), [121](#)  
`reverse` (*sage.knots.knotinfo.SymmetryMutant attribute*), [121](#)  
`reverse()` (*sage.knots.link.Link method*), [70](#)

## S

`sage.knots.free_knotinfo_monoid`  
`module`, [125](#)  
`sage.knots.knot`  
`module`, [1](#)  
`sage.knots.knotinfo`  
`module`, [77](#)  
`sage.knots.link`  
`module`, [11](#)  
`seifert_circles()` (*sage.knots.link.Link method*), [72](#)  
`seifert_matrix()` (*sage.knots.link.Link method*), [73](#)  
`series()` (*sage.knots.knotinfo.KnotInfoBase method*), [114](#)  
`signature()` (*sage.knots.knotinfo.KnotInfoBase method*), [114](#)  
`signature()` (*sage.knots.link.Link method*), [74](#)  
`symmetry_type()` (*sage.knots.knotinfo.KnotInfoBase method*), [115](#)

## T

`three_genus()` (*sage.knots.knotinfo.KnotInfoBase method*), [115](#)  
`to_knotinfo()` (*sage.knots.free\_knotinfo\_monoid.FreeKnotInfoMonoidElement method*), [127](#)

## U

`unknown` (*sage.knots.knotinfo.SymmetryMutant attribute*), [121](#)

## W

`writhe()` (*sage.knots.link.Link method*), [75](#)