
Sage Reference Manual: Algebras

Release 7.6

The Sage Development Team

Mar 25, 2017

CONTENTS

1	Catalog of Algebras	1
2	Free associative algebras and quotients	3
3	Finite dimensional algebras	59
4	Named associative algebras	73
5	Various associative algebras	309
6	Non-associative algebras	343
7	Indices and Tables	363
	Bibliography	365

CATALOG OF ALGEBRAS

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

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

- `algebras.Brauer`
- `algebras.Clifford`
- `algebras.ClusterAlgebra`
- `algebras.Descent`
- `algebras.DifferentialWeyl`
- `algebras.Exterior`
- `algebras.FiniteDimensional`
- `algebras.Free`
- `algebras.FreeZinbiel`
- `algebras.PreLieAlgebra`
- `algebras.GradedCommutative`
- `algebras.Group`
- `algebras.Hall`
- `algebras.Incidence`
- `algebras.IwahoriHecke`
- `algebras.Moebius`
- `algebras.Jordan`
- `algebras.NilCoxeter`
- `algebras.OrlikSolomon`
- `algebras.Partition`
- `algebras.PlanarPartition`
- `algebras.Quaternion`
- `algebras.RationalCherednik`

- `algebras.Schur`
- `algebras.Shuffle`
- `algebras.Steenrod`
- `algebras.TemperleyLieb`
- `algebras.YokonumaHecke`
- `algebras.Tensor`

FREE ASSOCIATIVE ALGEBRAS AND QUOTIENTS

2.1 Free algebras

AUTHORS:

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

EXAMPLES:

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

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

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

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

```

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

```

```

sage: F.<x,y,z> = FreeAlgebra(GF(5),3)
sage: TestSuite(F).run()
sage: F is loads(dumps(F))
True
sage: F.<x,y,z> = FreeAlgebra(GF(5),3, implementation='letterplace')
sage: TestSuite(F).run()
sage: F is loads(dumps(F))
True

```

```

sage: F = FreeAlgebra(GF(5),3, ['xx', 'zba', 'Y'])
sage: TestSuite(F).run()
sage: F is loads(dumps(F))
True
sage: F = FreeAlgebra(GF(5),3, ['xx', 'zba', 'Y'], implementation='letterplace')
sage: TestSuite(F).run()
sage: F is loads(dumps(F))
True

```

```

sage: F = FreeAlgebra(GF(5),3, 'abc')
sage: TestSuite(F).run()
sage: F is loads(dumps(F))
True
sage: F = FreeAlgebra(GF(5),3, 'abc', implementation='letterplace')
sage: TestSuite(F).run()
sage: F is loads(dumps(F))
True

```

```

sage: F = FreeAlgebra(FreeAlgebra(ZZ,2,'ab'), 2, 'x')
sage: TestSuite(F).run()
sage: F is loads(dumps(F))
True

```

Note that the letterplace implementation can only be used if the corresponding (multivariate) polynomial ring has an implementation in Singular:

```

sage: FreeAlgebra(FreeAlgebra(ZZ,2,'ab'), 2, 'x', implementation='letterplace')
Traceback (most recent call last):
...
NotImplementedError: The letterplace implementation is not available for the free_
↪algebra you requested

```

```

class sage.algebras.free_algebra. FreeAlgebraFactory
    Bases: sage.structure.factory.UniqueFactory

```


A constructor of free algebras.

See `free_algebra` for examples and corner cases.

EXAMPLES:

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

Free algebras are globally unique:

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

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

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

```
sage: H = FreeAlgebra(GF(5), ['x','y','z'], implementation='letterplace',
↳ degrees=[1,2,3])
sage: F != H != G
True
sage: H is FreeAlgebra(GF(5), ['x','y','z'], implementation='letterplace',
↳ degrees=[1,2,3])
True
```

```
sage: copy(H) is H is loads(dumps(H))
True
sage: TestSuite(H).run()
```

Free algebras commute with their base ring.

```
sage: K.<a,b> = FreeAlgebra(QQ,2)
sage: K.is_commutative()
False
sage: L.<c> = FreeAlgebra(K,1)
sage: L.is_commutative()
False
sage: s = a*b^2 * c^3; s
a*b^2*c^3
sage: parent(s)
Free Algebra on 1 generators (c,) over Free Algebra on 2 generators (a, b) over
↳Rational Field
sage: c^3 * a * b^2
a*b^2*c^3
```

create_key (*base_ring*, *arg1=None*, *arg2=None*, *sparse=False*, *order='degrevlex'*, *names=None*, *name=None*, *implementation=None*, *degrees=None*)
Create the key under which a free algebra is stored.

create_object (*version*, *key*)
Construct the free algebra that belongs to a unique key.

NOTE:

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

```
class sage.algebras.free_algebra. FreeAlgebra_generic ( R,n,names)
Bases: sage.combinat.free_module.CombinatorialFreeModule ,
sage.rings.ring.Algebra
```

The free algebra on n generators over a base ring.

INPUT:

- R – a ring
- n – an integer
- $names$ – the generator names

EXAMPLES:

```
sage: F.<x,y,z> = FreeAlgebra(QQ, 3); F
Free Algebra on 3 generators (x, y, z) over Rational Field
sage: mul(F.gens())
x*y*z
sage: mul([ F.gen(i%3) for i in range(12) ])
x*y*z*x*y*z*x*y*z*x*y*z*x*y*z
sage: mul([ F.gen(i%3) for i in range(12) ]) + mul([ F.gen(i%2) for i in
↳range(12) ])
x*y*x*y*x*y*x*y*x*y*x*y*x*y + x*y*z*x*y*z*x*y*z*x*y*z
sage: (2 + x*z + x^2)^2 + (x - y)^2
4 + 5*x^2 - x*y + 4*x*z - y*x + y^2 + x^4 + x^3*z + x*z*x^2 + x*z*x*z
```

Free algebras commute with their base ring.

```

sage: K.<a,b> = FreeAlgebra(QQ)
sage: K.is_commutative()
False
sage: L.<c,d> = FreeAlgebra(K)
sage: L.is_commutative()
False
sage: s = a*b^2 * c^3; s
a*b^2*c^3
sage: parent(s)
Free Algebra on 2 generators (c, d) over Free Algebra on 2 generators (a, b) over
↪Rational Field
sage: c^3 * a * b^2
a*b^2*c^3

```

Element

alias of `FreeAlgebraElement`

algebra_generators ()

Return the algebra generators of `self`.

EXAMPLES:

```

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

```

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

The G -Algebra derived from this algebra by relations. By default is assumed, that two variables commute.

Todo

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

EXAMPLES:

```

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

```

gen (*i*)The *i* -th generator of the algebra.

EXAMPLES:

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

gens ()Return the generators of *self* .

EXAMPLES:

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

is_commutative ()

Return True if this free algebra is commutative.

EXAMPLES:

```
sage: R.<x> = FreeAlgebra(QQ, 1)
sage: R.is_commutative()
True
sage: R.<x, y> = FreeAlgebra(QQ, 2)
sage: R.is_commutative()
False
```

is_field (*proof=True*)

Return True if this Free Algebra is a field, which is only if the base ring is a field and there are no generators

EXAMPLES:

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

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

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

INPUT:

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

EXAMPLES:

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

```

sage: F.lie_polynomial(x^2*y*x)
x^2*y*x - 2*x*y*x^2 + y*x^3
sage: F.lie_polynomial(y*z*x*z*x*z)
y*z*x*z*x*z - y*z*x*z^2*x - y*z^2*x^2*z + y*z^2*x*z*x
- z*y*x*z*x*z + z*y*x*z^2*x + z*y*z*x^2*z - z*y*z*x*z*x

```

monoid ()

The free monoid of generators of the algebra.

EXAMPLES:

```

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

```

ngens ()

The number of generators of the algebra.

EXAMPLES:

```

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

```

one_basis ()

Return the index of the basis element 1.

EXAMPLES:

```

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

```

pbw_basis ()

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

EXAMPLES:

```

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

```

pbw_element (elt)

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

EXAMPLES:

```

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

```

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

EXAMPLES:

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

product_on_basis (x, y)Return the product of the basis elements indexed by `x` and `y`.

EXAMPLES:

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

quo (mons, mats=None, names=None)

Return a quotient algebra.

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

EXAMPLES:

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

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

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

Return a quotient algebra.

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

EXAMPLES:

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

```
sage: n = 3
sage: A = FreeAlgebra(QQ,n,'i')
sage: F = A.monoid()
```

```

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

```

```

class sage.algebras.free_algebra. PBWBasisOfFreeAlgebra ( alg)
    Bases: sage.combinat.free_module.CombinatorialFreeModule

```

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

EXAMPLES:

```

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

```

We can convert between the two bases:

```

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

```

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

```

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

```

We verify Examples 1 and 2 in [MR1989]:

```

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

```

class Element (*M*, *x*)

Bases: `sage.combinat.free_module.CombinatorialFreeModuleElement`

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

expand ()

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

EXAMPLES:

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

`PBWBasisOfFreeAlgebra.algebra_generators` ()

Return the generators of `self` as an algebra.

EXAMPLES:

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

`PBWBasisOfFreeAlgebra.expansion` (*t*)

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

EXAMPLES:

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

`PBWBasisOfFreeAlgebra.free_algebra` ()

Return the associated free algebra of `self`.

EXAMPLES:

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

`PBWBasisOfFreeAlgebra.gen` (*i*)

Return the *i*-th generator of `self`.

EXAMPLES:


```

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

```

`PBWBasisOfFreeAlgebra.gens()`

Return the generators of self as an algebra.

EXAMPLES:

```

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

```

`PBWBasisOfFreeAlgebra.one_basis()`

Return the index of the basis element for 1.

EXAMPLES:

```

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

```

`PBWBasisOfFreeAlgebra.product(u, v)`

Return the product of two elements u and v.

EXAMPLES:

```

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

```

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

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

EXAMPLES:

```

sage: from sage.algebras.free_algebra import is_FreeAlgebra
sage: is_FreeAlgebra(5)
False
sage: is_FreeAlgebra(ZZ)
False
sage: is_FreeAlgebra(FreeAlgebra(ZZ, 100, 'x'))
True
sage: is_FreeAlgebra(FreeAlgebra(ZZ, 10, 'x', implementation='letterplace'))
True
sage: is_FreeAlgebra(FreeAlgebra(ZZ, 10, 'x', implementation='letterplace',
↪degrees=list(range(1, 11))))

```

True

2.2 Free algebra elements

AUTHORS:

- David Kohel (2005-09)

class sage.algebras.free_algebra_element.**FreeAlgebraElement** (A,x)
Bases: sage.structure.element.AlgebraElement, sage.combinat.free_module.CombinatorialFreeModule

A free algebra element.

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

EXAMPLES:

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

variables ()
Return the variables used in self.

EXAMPLES:

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

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

AUTHOR:

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

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

EXAMPLES:

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

Here is an example with degree weights:

```

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

```

TODO:

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

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

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

class `sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace`
 Bases: `sage.rings.ring.Algebra`

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

NOTE:

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

EXAMPLES:

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

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

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

commutative_ring()

Return the commutative version of this free algebra.

NOTE:

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

EXAMPLES:

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

current_ring ()

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

EXAMPLES:

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

degbound ()

Return the degree bound that is currently used.

NOTE:

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

EXAMPLES:

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

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

gen (i)

Return the i -th generator.

INPUT:

i – an integer.

OUTPUT:

Generator number i .

EXAMPLES:

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

generator_degrees ()

ideal_monoid ()

Return the monoid of ideals of this free algebra.

EXAMPLES:

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

is_commutative ()

Tell whether this algebra is commutative, i.e., whether the generator number is one.

EXAMPLES:

```
sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace')
sage: F.is_commutative()
False
sage: FreeAlgebra(QQ, implementation='letterplace', names=['x']).is_
↳ commutative()
True
```

is_field ()

Tell whether this free algebra is a field.

NOTE:

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

ngens ()

Return the number of generators.

EXAMPLES:

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

set_degbound (d)

Increase the degree bound that is currently in place.

NOTE:

The degree bound can not be decreased.

EXAMPLES:

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

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

```
sage: F.degbound()
4
sage: F.set_degbound(2)
sage: F.degbound()
4
```

term_order_of_block ()

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

EXAMPLES:

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

```
sage.algebras.letterplace.free_algebra_letterplace.poly_reduce ( ring=None,
                                                                interrupt-
                                                                ible=True, at-
                                                                tributes=None,
                                                                *args)
```

This function is an automatically generated C wrapper around the Singular function ‘NF’.

This wrapper takes care of converting Sage datatypes to Singular datatypes and vice versa. In addition to whatever parameters the underlying Singular function accepts when called, this function also accepts the following keyword parameters:

INPUT:

- args – a list of arguments
- ring – a multivariate polynomial ring
- interruptible – if True pressing Ctrl-C during the execution of this function will interrupt the computation (default: True)
- attributes – a dictionary of optional Singular attributes assigned to Singular objects (default: None)

If ring is not specified, it is guessed from the given arguments. If this is not possible, then a dummy ring, univariate polynomial ring over QQ, is used.

EXAMPLES:

```
sage: groebner = sage.libs.singular.function_factory.ff.groebner
sage: P.<x, y> = PolynomialRing(QQ)
sage: I = P.ideal(x^2-y, y+x)
sage: groebner(I)
[x + y, y^2 - y]
sage: triangL = sage.libs.singular.function_factory.ff.triang__lib.triangL
sage: P.<x1, x2> = PolynomialRing(QQ, order='lex')
sage: f1 = 1/2*((x1^2 + 2*x1 - 4)*x2^2 + 2*(x1^2 + x1)*x2 + x1^2)
sage: f2 = 1/2*((x1^2 + 2*x1 + 1)*x2^2 + 2*(x1^2 + x1)*x2 - 4*x1^2)
sage: I = Ideal(Ideal(f1,f2).groebner_basis()[::-1])
sage: triangL(I, attributes={I: {'isSB': 1}})
[[x2^4 + 4*x2^3 - 6*x2^2 - 20*x2 + 5, 8*x1 - x2^3 + x2^2 + 13*x2 - 5],
 [x2, x1^2],
 [x2, x1^2],
 [x2, x1^2]]
```

The Singular documentation for 'NF' is given below.

5.1.124 reduce

`*Syntax:*

```
`reduce (' poly_expression`, ' ideal_expression `)'
`reduce (' poly_expression`, ' ideal_expression`, ' int_expression
`)'
`reduce (' poly_expression`, ' poly_expression`, ' ideal_expression
`)'
`reduce (' vector_expression`, ' ideal_expression `)'
`reduce (' vector_expression`, ' ideal_expression`, ' int_expression
`)'
`reduce (' vector_expression`, ' module_expression `)'
`reduce (' vector_expression`, ' module_expression`, '
int_expression `)'
`reduce (' vector_expression`, ' poly_expression`, '
module_expression `)'
`reduce (' ideal_expression`, ' ideal_expression `)'
`reduce (' ideal_expression`, ' ideal_expression`, ' int_expression
`)'
`reduce (' ideal_expression`, ' matrix_expression`, '
ideal_expression `)'
`reduce (' module_expression`, ' ideal_expression `)'
`reduce (' module_expression`, ' ideal_expression`, ' int_expression
`)'
`reduce (' module_expression`, ' module_expression `)'
`reduce (' module_expression`, ' module_expression`, '
int_expression `)'
`reduce (' module_expression`, ' matrix_expression`, '
module_expression `)'
`reduce (' poly/vector/ideal/module`, ' ideal/module`, ' int`, '
intvec `)'
`reduce (' ideal`, ' matrix`, ' ideal`, ' int `)'
`reduce (' poly`, ' poly`, ' ideal`, ' int `)'
`reduce (' poly`, ' poly`, ' ideal`, ' int`, ' intvec `)'
```

`*Type:*

the type of the first argument

`*Purpose:*

reduces a polynomial, vector, ideal or module to its normal form with respect to an ideal or module represented by a standard basis. Returns 0 if and only if the polynomial (resp. vector, ideal, module) is an element (resp. subideal, submodule) of the ideal (resp. module). The result may have no meaning if the second argument is not a standard basis.

The third (optional) argument of type int modifies the behavior:

- * 0 default
- * 1 consider only the leading term and do no tail reduction.
- * 2 tail reduction: in the local/mixed ordering case: reduce also with bad ecart
- * 4 reduce without division, return possibly a non-zero constant multiple of the remainder

If a second argument ``u'` of type `poly` or `matrix` is given, the first argument ``p'` is replaced by ``p/u'`. This works only for zero dimensional ideals (resp. modules) in the third argument and gives, even in a local ring, a reduced normal form which is the projection to the quotient by the ideal (resp. module). One may give a degree bound in the fourth argument with respect to a weight vector in the fifth argument in order to have a finite computation. If some of the weights are zero, the procedure may not terminate!

``*Note_*`

The commands ``reduce'` and ``NF'` are synonymous.

``*Example:*`

```
ring r1 = 0, (z,y,x), ds;
poly s1=2x5y+7x2y4+3x2yz3;
poly s2=1x2y2z2+3z8;
poly s3=4xy5+2x2y2z3+11x10;
ideal i=s1,s2,s3;
ideal j=std(i);
reduce(3z3yx2+7y4x2+yx5+z12y2x2, j);
==> -yx5+2401/81y14x2+2744/81y11x5+392/27y8x8+224/81y5x11+16/81y2x14
reduce(3z3yx2+7y4x2+yx5+z12y2x2, j, 1);
==> -yx5+z12y2x2
// 4 arguments:
ring rs=0,x,ds;
// normalform of 1/(1+x) w.r.t. (x3) up to degree 5
reduce(poly(1),1+x,ideal(x3),5);
==> // ** _ is no standard basis
==> 1-x+x2
```

`* Menu:`

See

```
* ideal::
* module::
* std::
* vector::
```

`sage.algebras.letterplace.free_algebra_letterplace.singular_system` (*ring=None, interruptible=True, attributes=None, *args*)

This function is an automatically generated C wrapper around the Singular function `'system'`.

This wrapper takes care of converting Sage datatypes to Singular datatypes and vice versa. In addition to whatever parameters the underlying Singular function accepts when called, this function also accepts the following keyword parameters:

INPUT:

- `args` – a list of arguments
- `ring` – a multivariate polynomial ring
- `interruptible` – if `True` pressing `Ctrl-C` during the execution of this function will interrupt the

computation (default: True)

• `attributes` – a dictionary of optional Singular attributes assigned to Singular objects (default: None)

If `ring` is not specified, it is guessed from the given arguments. If this is not possible, then a dummy ring, univariate polynomial ring over $\mathbb{Q}\mathbb{Q}$, is used.

EXAMPLES:

```
sage: groebner = sage.libs.singular.function_factory.ff.groebner
sage: P.<x, y> = PolynomialRing(QQ)
sage: I = P.ideal(x^2-y, y+x)
sage: groebner(I)
[x + y, y^2 - y]
sage: triangL = sage.libs.singular.function_factory.ff.triang__lib.triangL
sage: P.<x1, x2> = PolynomialRing(QQ, order='lex')
sage: f1 = 1/2*((x1^2 + 2*x1 - 4)*x2^2 + 2*(x1^2 + x1)*x2 + x1^2)
sage: f2 = 1/2*((x1^2 + 2*x1 + 1)*x2^2 + 2*(x1^2 + x1)*x2 - 4*x1^2)
sage: I = Ideal(Ideal(f1,f2).groebner_basis()[::-1])
sage: triangL(I, attributes={I: {'isSB': 1}})
[[x2^4 + 4*x2^3 - 6*x2^2 - 20*x2 + 5, 8*x1 - x2^3 + x2^2 + 13*x2 - 5],
 [x2, x1^2],
 [x2, x1^2],
 [x2, x1^2]]
```

The Singular documentation for ‘system’ is given below.

```
5.1.148 system
-----

`*Syntax:*'
  `system (' string_expression `)'
  `system (' string_expression`, ' expression `)'

`*Type:*'
  depends on the desired function, may be none

`*Purpose:*'
  interface to internal data and the operating system. The
  string_expression determines the command to execute. Some commands
  require an additional argument (second form) where the type of the
  argument depends on the command. See below for a list of all
  possible commands.

`*Note_`*'
  Not all functions work on every platform.

`*Functions:*'

  `system("alarm", ' int `)'
      abort the Singular process after computing for that many
      seconds (system+user cpu time).

  `system("absFact", ' poly `)'
      absolute factorization of the polynomial (from a polynomial
      ring over a transcendental extension) Returns a list of the
      ideal of the factors, intvec of multiplicities, ideal of
      minimal polynomials and the number of factors.

  `system("blackbox")`
```

```

    list all blackbox data types.

`system("browsers");'
    returns a string about available help browsers.  *Note The
    online help system::.

`system("bracket",' poly, poly `)'
    returns the Lie bracket [p,q].

`system("btest",' poly, i2 `)'
    internal for shift algebra (with i2 variables): last block of
    the poly

`system("complexNearZero",' number_expression `)'
    checks for a small value for floating point numbers

`system("contributors")'
    returns names of people who contributed to the SINGULAR
    kernel as string.

`system("cpu")'
    returns the number of cpus as int (for creating multiple
    threads/processes).  (see `system("--cpus")').

`system("denom_list")'
    returns the list of denominators (number) which occurred in
    the latest std computation(s).  Is reset to the empty list
    at ring changes or by this system call.

`system("eigenvals",' matrix `)'
    returns the list of the eigenvalues of the matrix (as ideal,
    intvec).  (see `system("hessenberg")').

`system("env",' ring `)'
    returns the enveloping algebra (i.e.  $R \otimes R^{\text{opp}}$ ) See
    `system("opp")'.

`system("executable",' string `)'
    returns the path of the command given as argument or the
    empty string (for: not found) See `system("Singular")'.  See
    `system("getenv","PATH")'.

`system("freegb",' ideal, i2, i3 `)'
    returns the standard basis in the shift algebra i (with i3
    variables) up to degree i2.  See `system("opp")'.

`system("getenv",' string_expression `)'
    returns the value of the shell environment variable given as
    the second argument. The return type is string.

`system("getPrecDigits")'
    returns the precision for floating point numbers

`system("gmsnf",' ideal, ideal, matrix,int, int `)'
    Gauss-Manin system: for gmspoly.lib, gmssing.lib

`system("HC")'
    returns the degree of the "highest corner" from the last std

```

```

    computation (or 0).

`system("hessenberg",' matrix `)'
    returns the Hessenberg matrix (via QR algorithm).

`system("install",' s1, s2, p3, i4 `)'
    install a new method p3 for s2 for the newstruct type s1. s2
    must be a reserved operator with i4 operands (i4 may be
    1,2,3; use 4 for more than 3 or a varying number of arguments)
    See *Note Commands for user defined types::.

`system("LLL",' B `)'
    B must be a matrix or an intmat. Interface to NTLs LLL
    (Exact Arithmetic Variant over ZZ). Returns the same type as
    the input.
    B is an m x n matrix, viewed as m rows of n-vectors. m may
    be less than, equal to, or greater than n, and the rows need
    not be linearly independent. B is transformed into an
    LLL-reduced basis. The first m-rank(B) rows of B are zero.
    More specifically, elementary row transformations are
    performed on B so that the non-zero rows of new-B form an
    LLL-reduced basis for the lattice spanned by the rows of
    old-B.

`system("nblocks")' or `system("nblocks",' ring_name `)'
    returns the number of blocks of the given ring, or of the
    current basering, if no second argument is given. The return
    type is int.

`system("nc_hilb",' ideal, int, [...]' `)'
    internal support for ncHilb.lib, return nothing

`system("neworder",' ideal `)'
    string of the ring variables in an heurically good order for
    `char_series'

`system("newstruct")'
    list all newstruct data types.

`system("opp",' ring `)'
    returns the opposite ring.

`system("oppose",' ring R, poly p `)'
    returns the opposite polynomial of p from R.

`system("pcvLAddL",' list, list `)'
    `system("pcvPMulL",' poly, list `)'
    `system("pcvMinDeg",' poly `)'
    `system("pcvP2CV",' list, int, int `)'
    `system("pcvCV2P",' list, int, int `)'
    `system("pcvDim",' int, int `)'
    `system("pcvBasis",' int, int `)' internal for mondromy.lib

`system("pid")'
    returns the process number as int (for creating unique names).

`system("random")' or `system("random",' int `)'
    returns or sets the seed of the random generator.

```

```

`system("reserve",' int `)'
    reserve a port and listen with the given backlog. (see
    `system("reservedLink")').

`system("reservedLink")'
    accept a connect at the reserved port and return a
    (write-only) link to it. (see `system("reserve")').

`system("semaphore",' string, int `)'
    operations for semaphores: string may be `"init"', `"exists"',
    `"acquire"', `"try_acquire"', `"release"', `"get_value"', and
    int is the number of the semaphore. Returns -2 for wrong
    command, -1 for error or the result of the command.

`system("semic",' list, list `)'
    or `system("semic",' list, list, int `)' computes from list
    of spectrum numbers and list of spectrum numbers the
    semicontinuity index (qh, if 3rd argument is 1).

`system("setenv",'string_expression, string_expression`)'
    sets the shell environment variable given as the second
    argument to the value given as the third argument. Returns
    the third argument. Might not be available on all platforms.

`system("sh",' string_expression `)'
    shell escape, returns the return code of the shell as int.
    The string is sent literally to the shell.

`system("shrinktest",' poly, i2 `)'
    internal for shift algebra (with i2 variables): shrink the
    poly

`system("Singular")'
    returns the absolute (path) name of the running SINGULAR as
    string.

`system("SingularLib")'
    returns the colon separated library search path name as
    string.

`system("spadd",' list, list `)'
    or `system("spadd",' list, list, int `)' computes from list
    of spectrum numbers and list of spectrum numbers the sum of
    the lists.

`system("spectrum",' poly `)'
    or `system("spectrum",' poly, int `)'

`system("spmul",' list, int `)'
    or `system("spmul",' list, list, int `)' computes from list
    of spectrum numbers the multiple of it.

`system("std_syz",' module, int `)'
    compute a partial groebner base of a module, stopp after the
    given column

`system("stest",' poly, i2, i3, i4 `)'

```

```

    internal for shift algebra (with i4 variables): shift the
    poly by i2, up to degree i3

`system("tensorModuleMult",' int, module `)'
    internal for sheafcoh.lib (see id_TensorModuleMult)

`system("twostd",' ideal `)'
    returns the two-sided standard basis of the two-sided ideal.

`system("uname")'
    returns a string identifying the architecture for which
    SINGULAR was compiled.

`system("version")'
    returns the version number of SINGULAR as int.  (Version
    a-b-c-d returns a*10000+b*1000+c*100+d)

`system("with")'
    without an argument: returns a string describing the current
    version of SINGULAR, its build options, the used path names
    and other configurations
    with a string argument: test for that feature and return an
    int.

`system("--cpus")'
    returns the number of available cpu cores as int (for using
    multiple cores).  (see `system("cpu")').

`system("--")'
    prints the values of all options.

`system("--long_option_name")'
    returns the value of the (command-line) option
    long_option_name. The type of the returned value is either
    string or int.  *Note Command line options::, for more info.

`system("--long_option_name",' expression')'
    sets the value of the (command-line) option long_option_name
    to the value given by the expression. Type of the expression
    must be string, or int. *Note Command line options::, for
    more info. Among others, this can be used for setting the
    seed of the random number generator, the used help browser,
    the minimal display time, or the timer resolution.

`*Example:*'
    // a listing of the current directory:
    system("sh","ls");
    // execute a shell, return to SINGULAR with exit:
    system("sh","sh");
    string unique_name="/tmp/xx"+string(system("pid"));
    unique_name;
    ==> /tmp/xx4711
    system("uname")
    ==> ix86-Linux
    system("getenv","PATH");
    ==> /bin:/usr/bin:/usr/local/bin
    system("Singular");
    ==> /usr/local/bin/Singular

```

```

// report value of all options
system("--");
==> // --batch          0
==> // --execute        0
==> // --sdb            0
==> // --echo           1
==> // --profile        0
==> // --quiet          1
==> // --sort           0
==> // --random         12345678
==> // --no-tty         1
==> // --user-option    0
==> // --allow-net      0
==> // --browser        0
==> // --cntrlc         0
==> // --emacs          0
==> // --no-stdlib      0
==> // --no-rc          1
==> // --no-warn        0
==> // --no-out         0
==> // --no-shell       0
==> // --min-time       "0.5"
==> // --cpus           4
==> // --MPport         0
==> // --MPhost         0
==> // --link           0
==> // --ticks-per-sec  1
// set minimal display time to 0.02 seconds
system("--min-time", "0.02");
// set timer resolution to 0.01 seconds
system("--ticks-per-sec", 100);
// re-seed random number generator
system("--random", 12345678);
// allow your web browser to access HTML pages from the net
system("--allow-net", 1);
// and set help browser to firefox
system("--browser", "firefox");
==> // ** Could not get 'DataDir'.
==> // ** Either set environment variable 'SINGULAR_DATA_DIR' to
→ 'DataDir',
==> // ** or make sure that 'DataDir' is at "/home/hannes/singular/doc/.
→ ./Sin\
    gular/./share/"
==> // ** Could not get 'IdxFile'.
==> // ** Either set environment variable 'SINGULAR_IDX_FILE' to
→ 'IdxFile',
==> // ** Could not get 'DataDir'.
==> // ** Either set environment variable 'SINGULAR_DATA_DIR' to
→ 'DataDir',
==> // ** or make sure that 'DataDir' is at "/home/hannes/singular/doc/.
→ ./Sin\
    gular/./share/"
==> // ** or make sure that 'IdxFile' is at "%D/singular/singular.idx"
==> // ** resource `x` not found
==> // ** Setting help browser to 'dummy'.

```

2.4 Weighted homogeneous elements of free algebras, in letterplace implementation.

AUTHOR:

- Simon King (2011-03-23): Trac ticket [trac ticket #7797](#)

class sage.algebras.letterplace.free_algebra_element_letterplace. **FreeAlgebraElement_letterplace**
 Bases: sage.structure.element.AlgebraElement

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

EXAMPLES:

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

Here is an example with nontrivial degree weights:

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

degree ()

Return the degree of this element.

NOTE:

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

EXAMPLES:

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


lc()

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

EXAMPLES:

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

letterplace_polynomial()

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

EXAMPLES:

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

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

```
sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace',
↳degrees=[2,1,3])
sage: ((x*y+z)^2).letterplace_polynomial()
x*x__1*y_2*x__3*x__4*y_5 + x*x__1*y_2*z_3*x__4*x__5 + z*x__1*x__2*x__3*x__4*y_5
↳+ z*x__1*x__2*z_3*x__4*x__5
```

lm()

The leading monomial of this free algebra element.

EXAMPLES:

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

lm_divides(p)

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

NOTE:

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

EXAMPLES:

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

```
sage: (y*x*y-y^4).lm_divides((2*x*y+z)^2*z)
True
```

lt ()

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

EXAMPLES:

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

normal_form (I)

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

INPUT:

A twosided homogeneous ideal I of the parent F of this element, x .

OUTPUT:

The normal form of x wrt. I .

NOTE:

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

EXAMPLES:

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

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

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

Here is an example with non-trivial degree weights:

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

reduce (*G*)

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

INPUT:

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

OUTPUT:

The twosided reduction of this element by the argument.

NOTE:

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

EXAMPLES:

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

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

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

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

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

```
sage.algebras.letterplace.free_algebra_element_letterplace.poly_reduce ( ring=None,
in-
ter-
rupt-
ible=True,
at-
tributes=None,
*args)
```

This function is an automatically generated C wrapper around the Singular function ‘NF’.

This wrapper takes care of converting Sage datatypes to Singular datatypes and vice versa. In addition to whatever parameters the underlying Singular function accepts when called, this function also accepts the following keyword parameters:

INPUT:

- *args* – a list of arguments

- `ring` – a multivariate polynomial ring
- `interruptible` – if `True` pressing Ctrl-C during the execution of this function will interrupt the computation (default: `True`)
- `attributes` – a dictionary of optional Singular attributes assigned to Singular objects (default: `None`)

If `ring` is not specified, it is guessed from the given arguments. If this is not possible, then a dummy ring, univariate polynomial ring over $\mathbb{Q}\mathbb{Q}$, is used.

EXAMPLES:

```
sage: groebner = sage.libs.singular.function_factory.ff.groebner
sage: P.<x, y> = PolynomialRing(QQ)
sage: I = P.ideal(x^2-y, y+x)
sage: groebner(I)
[x + y, y^2 - y]
sage: triangL = sage.libs.singular.function_factory.ff.triang__lib.triangL
sage: P.<x1, x2> = PolynomialRing(QQ, order='lex')
sage: f1 = 1/2*((x1^2 + 2*x1 - 4)*x2^2 + 2*(x1^2 + x1)*x2 + x1^2)
sage: f2 = 1/2*((x1^2 + 2*x1 + 1)*x2^2 + 2*(x1^2 + x1)*x2 - 4*x1^2)
sage: I = Ideal(Ideal(f1,f2).groebner_basis()[::-1])
sage: triangL(I, attributes={I: {'isSB': 1}})
[[x2^4 + 4*x2^3 - 6*x2^2 - 20*x2 + 5, 8*x1 - x2^3 + x2^2 + 13*x2 - 5],
 [x2, x1^2],
 [x2, x1^2],
 [x2, x1^2]]
```

The Singular documentation for 'NF' is given below.

```
5.1.124 reduce
-----

`*Syntax:*'
`reduce (' poly_expression`, ' ideal_expression `)'
`reduce (' poly_expression`, ' ideal_expression`, ' int_expression
`)'
`reduce (' poly_expression`, ' poly_expression`, ' ideal_expression
`)'
`reduce (' vector_expression`, ' ideal_expression `)'
`reduce (' vector_expression`, ' ideal_expression`, ' int_expression
`)'
`reduce (' vector_expression`, ' module_expression `)'
`reduce (' vector_expression`, ' module_expression`, '
int_expression `)'
`reduce (' vector_expression`, ' poly_expression`, '
module_expression `)'
`reduce (' ideal_expression`, ' ideal_expression `)'
`reduce (' ideal_expression`, ' ideal_expression`, ' int_expression
`)'
`reduce (' ideal_expression`, ' matrix_expression`, '
ideal_expression `)'
`reduce (' module_expression`, ' ideal_expression `)'
`reduce (' module_expression`, ' ideal_expression`, ' int_expression
`)'
`reduce (' module_expression`, ' module_expression `)'
`reduce (' module_expression`, ' module_expression`, '
int_expression `)'
`reduce (' module_expression`, ' matrix_expression`, '
module_expression `)'
```

```

`reduce (' poly/vector/ideal/module`,` ideal/module`,` int`,`
intvec `)`
`reduce (' ideal`,` matrix`,` ideal`,` int `)`
`reduce (' poly`,` poly`,` ideal`,` int `)`
`reduce (' poly`,` poly`,` ideal`,` int`,` intvec `)`

`*Type:*'
  the type of the first argument

`*Purpose:*'
  reduces a polynomial, vector, ideal or module to its normal form
  with respect to an ideal or module represented by a standard basis.
  Returns 0 if and only if the polynomial (resp. vector, ideal,
  module) is an element (resp. subideal, submodule) of the ideal
  (resp. module). The result may have no meaning if the second
  argument is not a standard basis.
  The third (optional) argument of type int modifies the behavior:
    * 0 default

    * 1 consider only the leading term and do no tail reduction.

    * 2 tail reduction:n the local/mixed ordering case: reduce also
      with bad ecart

    * 4 reduce without division, return possibly a non-zero
      constant multiple of the remainder

  If a second argument `u' of type poly or matrix is given, the
  first argument `p' is replaced by `p/u'. This works only for zero
  dimensional ideals (resp. modules) in the third argument and
  gives, even in a local ring, a reduced normal form which is the
  projection to the quotient by the ideal (resp. module). One may
  give a degree bound in the fourth argument with respect to a
  weight vector in the fifth argument in order have a finite
  computation. If some of the weights are zero, the procedure may
  not terminate!

`*Note_`*'
  The commands `reduce' and `NF' are synonymous.

`*Example:*'
  ring r1 = 0, (z,y,x), ds;
  poly s1=2x5y+7x2y4+3x2yz3;
  poly s2=1x2y2z2+3z8;
  poly s3=4xy5+2x2y2z3+11x10;
  ideal i=s1,s2,s3;
  ideal j=std(i);
  reduce(3z3yx2+7y4x2+yx5+z12y2x2, j);
==> -yx5+2401/81y14x2+2744/81y11x5+392/27y8x8+224/81y5x11+16/81y2x14
  reduce(3z3yx2+7y4x2+yx5+z12y2x2, j, 1);
==> -yx5+z12y2x2
  // 4 arguments:
  ring rs=0,x,ds;
  // normalform of 1/(1+x) w.r.t. (x3) up to degree 5
  reduce(poly(1),1+x,ideal(x3),5);
==> // ** _ is no standard basis
==> 1-x+x2

```

```
* Menu:

See
* ideal::
* module::
* std::
* vector::
```

```
sage.algebras.letterplace.free_algebra_element_letterplace.singular_system ( ring=None,
                                                                    in-
                                                                    ter-
                                                                    rupt-
                                                                    ible=True,
                                                                    at-
                                                                    tributes=None,
                                                                    *args)
```

This function is an automatically generated C wrapper around the Singular function ‘system’.

This wrapper takes care of converting Sage datatypes to Singular datatypes and vice versa. In addition to whatever parameters the underlying Singular function accepts when called, this function also accepts the following keyword parameters:

INPUT:

- `args` – a list of arguments
- `ring` – a multivariate polynomial ring
- `interruptible` – if `True` pressing Ctrl-C during the execution of this function will interrupt the computation (default: `True`)
- `attributes` – a dictionary of optional Singular attributes assigned to Singular objects (default: `None`)

If `ring` is not specified, it is guessed from the given arguments. If this is not possible, then a dummy ring, univariate polynomial ring over $\mathbb{Q}\mathbb{Q}$, is used.

EXAMPLES:

```
sage: groebner = sage.libs.singular.function_factory.ff.groebner
sage: P.<x, y> = PolynomialRing(QQ)
sage: I = P.ideal(x^2-y, y+x)
sage: groebner(I)
[x + y, y^2 - y]
sage: triangL = sage.libs.singular.function_factory.ff.triang__lib.triangL
sage: P.<x1, x2> = PolynomialRing(QQ, order='lex')
sage: f1 = 1/2*((x1^2 + 2*x1 - 4)*x2^2 + 2*(x1^2 + x1)*x2 + x1^2)
sage: f2 = 1/2*((x1^2 + 2*x1 + 1)*x2^2 + 2*(x1^2 + x1)*x2 - 4*x1^2)
sage: I = Ideal(Ideal(f1,f2).groebner_basis()[::-1])
sage: triangL(I, attributes={I: {'isSB': 1}})
[[x2^4 + 4*x2^3 - 6*x2^2 - 20*x2 + 5, 8*x1 - x2^3 + x2^2 + 13*x2 - 5],
 [x2, x1^2],
 [x2, x1^2],
 [x2, x1^2]]
```

The Singular documentation for ‘system’ is given below.

```
5.1.148 system
-----

`*Syntax:*'
```

```

`system (' string_expression `)'
`system (' string_expression`, ' expression `)'

`*Type:*'
    depends on the desired function, may be none

`*Purpose:*'
    interface to internal data and the operating system. The
    string_expression determines the command to execute. Some commands
    require an additional argument (second form) where the type of the
    argument depends on the command. See below for a list of all
    possible commands.

`*Note_`*'
    Not all functions work on every platform.

`*Functions:*'

`system("alarm",' int `)'
    abort the Singular process after computing for that many
    seconds (system+user cpu time).

`system("absFact",' poly `)'
    absolute factorization of the polynomial (from a polynomial
    ring over a transcendental extension) Returns a list of the
    ideal of the factors, intvec of multiplicities, ideal of
    minimal polynomials and the number of factors.

`system("blackbox")'
    list all blackbox data types.

`system("browsers");'
    returns a string about available help browsers.  *Note The
    online help system::.

`system("bracket",' poly, poly `)'
    returns the Lie bracket [p,q].

`system("btest",' poly, i2 `)'
    internal for shift algebra (with i2 variables): last block of
    the poly

`system("complexNearZero",' number_expression `)'
    checks for a small value for floating point numbers

`system("contributors")'
    returns names of people who contributed to the SINGULAR
    kernel as string.

`system("cpu")'
    returns the number of cpus as int (for creating multiple
    threads/processes). (see `system("--cpus")').

`system("denom_list")'
    returns the list of denominators (number) which occurred in
    the latest std computation(s). Is reset to the empty list
    at ring changes or by this system call.

```

```

`system("eigenvals",' matrix `)'
    returns the list of the eigenvalues of the matrix (as ideal,
    intvec). (see `system("hessenberg")').

`system("env",' ring `)'
    returns the enveloping algebra (i.e.  $R \otimes R^{\text{opp}}$ ) See
    `system("opp")'.

`system("executable",' string `)'
    returns the path of the command given as argument or the
    empty string (for: not found) See `system("Singular")'. See
    `system("getenv","PATH")'.

`system("freegb",' ideal, i2, i3 `)'
    returns the standrda basis in the shift algebra  $i$  (with  $i3$ 
    variables) up to degree  $i2$ . See `system("opp")'.

`system("getenv",' string_expression `)'
    returns the value of the shell environment variable given as
    the second argument. The return type is string.

`system("getPrecDigits")'
    returns the precision for floating point numbers

`system("gmsnf",' ideal, ideal, matrix,int, int `)'
    Gauss-Manin system: for gmspoly.lib, gmssing.lib

`system("HC")'
    returns the degree of the "highest corner" from the last std
    computation (or 0).

`system("hessenberg",' matrix `)'
    returns the Hessenberg matrix (via QR algorithm).

`system("install",' s1, s2, p3, i4 `)'
    install a new method  $p3$  for  $s2$  for the newstruct type  $s1$ .  $s2$ 
    must be a reserved operator with  $i4$  operands ( $i4$  may be
    1,2,3; use 4 for more than 3 or a varying number of arguments)
    See *Note Commands for user defined types::.

`system("LLL",' B `)'
    B must be a matrix or an intmat. Interface to NTLs LLL
    (Exact Arithmetic Variant over  $\mathbb{Z}\mathbb{Z}$ ). Returns the same type as
    the input.
    B is an  $m \times n$  matrix, viewed as  $m$  rows of  $n$ -vectors.  $m$  may
    be less than, equal to, or greater than  $n$ , and the rows need
    not be linearly independent. B is transformed into an
    LLL-reduced basis. The first  $m - \text{rank}(B)$  rows of B are zero.
    More specifically, elementary row transformations are
    performed on B so that the non-zero rows of new-B form an
    LLL-reduced basis for the lattice spanned by the rows of
    old-B.

`system("nblocks")' or `system("nblocks",' ring_name `)'
    returns the number of blocks of the given ring, or of the
    current basering, if no second argument is given. The return
    type is int.

```



```

`system("nc_hilb",' ideal, int, [...] `)'
    internal support for ncHilb.lib, return nothing

`system("neworder",' ideal `)'
    string of the ring variables in an heurically good order for
    `char_series'

`system("newstruct")'
    list all newstruct data types.

`system("opp",' ring `)'
    returns the opposite ring.

`system("oppose",' ring R, poly p `)'
    returns the opposite polynomial of p from R.

`system("pcvLAddL",' list, list `)'
    `system("pcvPMulL",' poly, list `)'
    `system("pcvMinDeg",' poly `)'
    `system("pcvP2CV",' list, int, int `)'
    `system("pcvCV2P",' list, int, int `)'
    `system("pcvDim",' int, int `)'
    `system("pcvBasis",' int, int `)' internal for mondromy.lib

`system("pid")'
    returns the process number as int (for creating unique names).

`system("random")' or `system("random",' int `)'
    returns or sets the seed of the random generator.

`system("reserve",' int `)'
    reserve a port and listen with the given backlog. (see
    `system("reservedLink")').

`system("reservedLink")'
    accept a connect at the reserved port and return a
    (write-only) link to it. (see `system("reserve")').

`system("semaphore",' string, int `)'
    operations for semaphores: string may be `"init"', `"exists"',
    `"acquire"', `"try_acquire"', `"release"', `"get_value"', and
    int is the number of the semaphore. Returns -2 for wrong
    command, -1 for error or the result of the command.

`system("semic",' list, list `)'
    or `system("semic",' list, list, int `)' computes from list
    of spectrum numbers and list of spectrum numbers the
    semicontinuity index (qh, if 3rd argument is 1).

`system("setenv",'string_expression, string_expression`)'
    sets the shell environment variable given as the second
    argument to the value given as the third argument. Returns
    the third argument. Might not be available on all platforms.

`system("sh",' string_expression `)'
    shell escape, returns the return code of the shell as int.
    The string is sent literally to the shell.

```

```

`system("shrinktest",' poly, i2 `)'
    internal for shift algebra (with i2 variables): shrink the
    poly

`system("Singular")'
    returns the absolute (path) name of the running SINGULAR as
    string.

`system("SingularLib")'
    returns the colon seperated library search path name as
    string.

`system("spadd",' list, list `)'
    or `system("spadd",' list, list, int `)' computes from list
    of spectrum numbers and list of spectrum numbers the sum of
    the lists.

`system("spectrum",' poly `)'
    or `system("spectrum",' poly, int `)'

`system("spmul",' list, int `)'
    or `system("spmul",' list, list, int `)' computes from list
    of spectrum numbers the multiple of it.

`system("std_syz",' module, int `)'
    compute a partial groebner base of a module, stopp after the
    given column

`system("stest",' poly, i2, i3, i4 `)'
    internal for shift algebra (with i4 variables): shift the
    poly by i2, up to degree i3

`system("tensorModuleMult",' int, module `)'
    internal for sheafcoh.lib (see id_TensorModuleMult)

`system("twostd",' ideal `)'
    returns the two-sided standard basis of the two-sided ideal.

`system("uname")'
    returns a string identifying the architecture for which
    SINGULAR was compiled.

`system("version")'
    returns the version number of SINGULAR as int.  (Version
    a-b-c-d returns a*10000+b*1000+c*100+d)

`system("with")'
    without an argument: returns a string describing the current
    version of SINGULAR, its build options, the used path names
    and other configurations
    with a string argument: test for that feature and return an
    int.

`system("--cpus")'
    returns the number of available cpu cores as int (for using
    multiple cores).  (see `system("cpu")').

`system("'-'")'

```

```

prints the values of all options.

`system("-long_option_name`")'
    returns the value of the (command-line) option
    long_option_name. The type of the returned value is either
    string or int. *Note Command line options::, for more info.

`system("-long_option_name`", ' expression`)'
    sets the value of the (command-line) option long_option_name
    to the value given by the expression. Type of the expression
    must be string, or int. *Note Command line options::, for
    more info. Among others, this can be used for setting the
    seed of the random number generator, the used help browser,
    the minimal display time, or the timer resolution.

`*Example:*'
    // a listing of the current directory:
    system("sh","ls");
    // execute a shell, return to SINGULAR with exit:
    system("sh","sh");
    string unique_name="/tmp/xx"+string(system("pid"));
    unique_name;
    ==> /tmp/xx4711
    system("uname")
    ==> ix86-Linux
    system("getenv","PATH");
    ==> /bin:/usr/bin:/usr/local/bin
    system("Singular");
    ==> /usr/local/bin/Singular
    // report value of all options
    system("--");
    ==> // --batch          0
    ==> // --execute
    ==> // --sdb            0
    ==> // --echo          1
    ==> // --profile       0
    ==> // --quiet         1
    ==> // --sort          0
    ==> // --random        12345678
    ==> // --no-tty        1
    ==> // --user-option
    ==> // --allow-net      0
    ==> // --browser
    ==> // --cntrlc
    ==> // --emacs         0
    ==> // --no-stdlib     0
    ==> // --no-rc         1
    ==> // --no-warn       0
    ==> // --no-out        0
    ==> // --no-shell      0
    ==> // --min-time      "0.5"
    ==> // --cpus          4
    ==> // --MPport
    ==> // --MPhost
    ==> // --link
    ==> // --ticks-per-sec  1
    // set minimal display time to 0.02 seconds
    system("--min-time", "0.02");

```

```

// set timer resolution to 0.01 seconds
system("--ticks-per-sec", 100);
// re-seed random number generator
system("--random", 12345678);
// allow your web browser to access HTML pages from the net
system("--allow-net", 1);
// and set help browser to firefox
system("--browser", "firefox");
==> // ** Could not get 'DataDir'.
==> // ** Either set environment variable 'SINGULAR_DATA_DIR' to
↪ 'DataDir',
==> // ** or make sure that 'DataDir' is at "/home/hannes/singular/doc/.
↪ ./Sin\
    gular/./share/"
==> // ** Could not get 'IdxFile'.
==> // ** Either set environment variable 'SINGULAR_IDX_FILE' to
↪ 'IdxFile',
==> // ** Could not get 'DataDir'.
==> // ** Either set environment variable 'SINGULAR_DATA_DIR' to
↪ 'DataDir',
==> // ** or make sure that 'DataDir' is at "/home/hannes/singular/doc/.
↪ ./Sin\
    gular/./share/"
==> // ** or make sure that 'IdxFile' is at "%D/singular/singular.idx"
==> // ** resource `x` not found
==> // ** Setting help browser to 'dummy'.

```

2.5 Homogeneous ideals of free algebras.

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

EXAMPLES:

```

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

```

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

```

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

```

AUTHOR:

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

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

Bases: `sage.rings.noncommutative_ideals.Ideal_nc`

Graded homogeneous ideals in free algebras.

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

EXAMPLES:

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

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

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

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

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

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

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

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

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

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

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

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

Again, we can compute normal forms:

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

groebner_basis (*degbound=None*)

Twosided Groebner basis with degree bound.

INPUT:

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

ASSUMPTIONS:

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

NOTES:

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

EXAMPLES:

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

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

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

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

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

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

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

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

reduce (G)

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

INPUT:

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

OUTPUT:

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

EXAMPLES:

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

```
sage.algebras.letterplace.letterplace_ideal.poly_reduce ( ring=None,      inter-
                                                         ruptible=True,      at-
                                                         tributes=None, *args)
```

This function is an automatically generated C wrapper around the Singular function 'NF'.

This wrapper takes care of converting Sage datatypes to Singular datatypes and vice versa. In addition to whatever parameters the underlying Singular function accepts when called, this function also accepts the following keyword parameters:

INPUT:

- `args` – a list of arguments
- `ring` – a multivariate polynomial ring
- `interruptible` – if `True` pressing Ctrl-C during the execution of this function will interrupt the computation (default: `True`)
- `attributes` – a dictionary of optional Singular attributes assigned to Singular objects (default: `None`)

If `ring` is not specified, it is guessed from the given arguments. If this is not possible, then a dummy ring, univariate polynomial ring over $\mathbb{Q}\mathbb{Q}$, is used.

EXAMPLES:

```
sage: groebner = sage.libs.singular.function_factory.ff.groebner
sage: P.<x, y> = PolynomialRing(QQ)
sage: I = P.ideal(x^2-y, y+x)
sage: groebner(I)
[x + y, y^2 - y]
sage: triangL = sage.libs.singular.function_factory.ff.triang__lib.triangL
sage: P.<x1, x2> = PolynomialRing(QQ, order='lex')
```



```

sage: f1 = 1/2*((x1^2 + 2*x1 - 4)*x2^2 + 2*(x1^2 + x1)*x2 + x1^2)
sage: f2 = 1/2*((x1^2 + 2*x1 + 1)*x2^2 + 2*(x1^2 + x1)*x2 - 4*x1^2)
sage: I = Ideal(Ideal(f1,f2).groebner_basis()[::-1])
sage: triangL(I, attributes={I: {'isSB':1}})
[[x2^4 + 4*x2^3 - 6*x2^2 - 20*x2 + 5, 8*x1 - x2^3 + x2^2 + 13*x2 - 5],
 [x2, x1^2],
 [x2, x1^2],
 [x2, x1^2]]

```

The Singular documentation for 'NF' is given below.

5.1.124 reduce

`*Syntax:`

```

`reduce (' poly_expression`, ' ideal_expression `)'
`reduce (' poly_expression`, ' ideal_expression`, ' int_expression
`)'
`reduce (' poly_expression`, ' poly_expression`, ' ideal_expression
`)'
`reduce (' vector_expression`, ' ideal_expression `)'
`reduce (' vector_expression`, ' ideal_expression`, ' int_expression
`)'
`reduce (' vector_expression`, ' module_expression `)'
`reduce (' vector_expression`, ' module_expression`, '
int_expression `)'
`reduce (' vector_expression`, ' poly_expression`, '
module_expression `)'
`reduce (' ideal_expression`, ' ideal_expression `)'
`reduce (' ideal_expression`, ' ideal_expression`, ' int_expression
`)'
`reduce (' ideal_expression`, ' matrix_expression`, '
ideal_expression `)'
`reduce (' module_expression`, ' ideal_expression `)'
`reduce (' module_expression`, ' ideal_expression`, ' int_expression
`)'
`reduce (' module_expression`, ' module_expression `)'
`reduce (' module_expression`, ' module_expression`, '
int_expression `)'
`reduce (' module_expression`, ' matrix_expression`, '
module_expression `)'
`reduce (' poly/vector/ideal/module`, ' ideal/module`, ' int`, '
intvec `)'
`reduce (' ideal`, ' matrix`, ' ideal`, ' int `)'
`reduce (' poly`, ' poly`, ' ideal`, ' int `)'
`reduce (' poly`, ' poly`, ' ideal`, ' int`, ' intvec `)'

```

`*Type:`

the type of the first argument

`*Purpose:`

reduces a polynomial, vector, ideal or module to its normal form with respect to an ideal or module represented by a standard basis. Returns 0 if and only if the polynomial (resp. vector, ideal, module) is an element (resp. subideal, submodule) of the ideal (resp. module). The result may have no meaning if the second argument is not a standard basis. The third (optional) argument of type int modifies the behavior:

- * 0 default
- * 1 consider only the leading term and do no tail reduction.
- * 2 tail reduction:
the local/mixed ordering case: reduce also with bad ecart
- * 4 reduce without division, return possibly a non-zero constant multiple of the remainder

If a second argument ``u'` of type `poly` or `matrix` is given, the first argument ``p'` is replaced by ``p/u'`. This works only for zero dimensional ideals (resp. modules) in the third argument and gives, even in a local ring, a reduced normal form which is the projection to the quotient by the ideal (resp. module). One may give a degree bound in the fourth argument with respect to a weight vector in the fifth argument in order to have a finite computation. If some of the weights are zero, the procedure may not terminate!

``*Note_*`

The commands ``reduce'` and ``NF'` are synonymous.

``*Example:*`

```
ring r1 = 0, (z,y,x), ds;
poly s1=2x5y+7x2y4+3x2yz3;
poly s2=1x2y2z2+3z8;
poly s3=4xy5+2x2y2z3+11x10;
ideal i=s1,s2,s3;
ideal j=std(i);
reduce(3z3yx2+7y4x2+yx5+z12y2x2, j);
==> -yx5+2401/81y14x2+2744/81y11x5+392/27y8x8+224/81y5x11+16/81y2x14
reduce(3z3yx2+7y4x2+yx5+z12y2x2, j, 1);
==> -yx5+z12y2x2
// 4 arguments:
ring rs=0,x,ds;
// normalform of 1/(1+x) w.r.t. (x3) up to degree 5
reduce(poly(1),1+x,ideal(x3),5);
==> // ** _ is no standard basis
==> 1-x+x2
```

* Menu:

See

- * ideal::
- * module::
- * std::
- * vector::

```
sage.algebras.letterplace.letterplace_ideal. singular_system ( ring=None,    in-
                                                    interruptible=True,
                                                    attributes=None,
                                                    *args)
```

This function is an automatically generated C wrapper around the Singular function `'system'`.

This wrapper takes care of converting Sage datatypes to Singular datatypes and vice versa. In addition to whatever parameters the underlying Singular function accepts when called, this function also accepts the following keyword parameters:

INPUT:

- `args` – a list of arguments
- `ring` – a multivariate polynomial ring
- `interruptible` – if `True` pressing `Ctrl-C` during the execution of this function will interrupt the computation (default: `True`)
- `attributes` – a dictionary of optional Singular attributes assigned to Singular objects (default: `None`)

If `ring` is not specified, it is guessed from the given arguments. If this is not possible, then a dummy ring, univariate polynomial ring over $\mathbb{Q}\mathbb{Q}$, is used.

EXAMPLES:

```
sage: groebner = sage.libs.singular.function_factory.ff.groebner
sage: P.<x, y> = PolynomialRing(QQ)
sage: I = P.ideal(x^2-y, y+x)
sage: groebner(I)
[x + y, y^2 - y]
sage: triangL = sage.libs.singular.function_factory.ff.triang__lib.triangL
sage: P.<x1, x2> = PolynomialRing(QQ, order='lex')
sage: f1 = 1/2*((x1^2 + 2*x1 - 4)*x2^2 + 2*(x1^2 + x1)*x2 + x1^2)
sage: f2 = 1/2*((x1^2 + 2*x1 + 1)*x2^2 + 2*(x1^2 + x1)*x2 - 4*x1^2)
sage: I = Ideal(Ideal(f1,f2).groebner_basis()[::-1])
sage: triangL(I, attributes={I: {'isSB': 1}})
[[x2^4 + 4*x2^3 - 6*x2^2 - 20*x2 + 5, 8*x1 - x2^3 + x2^2 + 13*x2 - 5],
 [x2, x1^2],
 [x2, x1^2],
 [x2, x1^2]]
```

The Singular documentation for ‘system’ is given below.

```
5.1.148 system
-----

`*Syntax:*'
  `system (' string_expression `)'
  `system (' string_expression`, ' expression `)'

`*Type:*'
  depends on the desired function, may be none

`*Purpose:*'
  interface to internal data and the operating system. The
  string_expression determines the command to execute. Some commands
  require an additional argument (second form) where the type of the
  argument depends on the command. See below for a list of all
  possible commands.

`*Note_`*'
  Not all functions work on every platform.

`*Functions:*'

  `system("alarm", ' int `)'
      abort the Singular process after computing for that many
      seconds (system+user cpu time).

  `system("absFact", ' poly `)'
```

```

absolute factorization of the polynomial (from a polynomial
ring over a transzedental extension) Returns a list of the
ideal of the factors, intvec of multiplicities, ideal of
minimal polynomials and the bumber of factors.

`system("blackbox")'
    list all blackbox data types.

`system("browsers");'
    returns a string about available help browsers.  *Note The
    online help system::.

`system("bracket",' poly, poly `)'
    returns the Lie bracket [p,q].

`system("btest",' poly, i2 `)'
    internal for shift algebra (with i2 variables): last block of
    the poly

`system("complexNearZero",' number_expression `)'
    checks for a small value for floating point numbers

`system("contributors")'
    returns names of people who contributed to the SINGULAR
    kernel as string.

`system("cpu")'
    returns the number of cpus as int (for creating multiple
    threads/processes). (see `system("--cpus")').

`system("denom_list")'
    returns the list of denominators (number) which occured in
    the latest std computationi(s). Is reset to the empty list
    at ring changes or by this system call.

`system("eigenvals",' matrix `)'
    returns the list of the eigenvalues of the matrix (as ideal,
    intvec). (see `system("hessenberg")').

`system("env",' ring `)'
    returns the enveloping algebra (i.e. R tensor R^opp) See
    `system("opp")'.

`system("executable",' string `)'
    returns the path of the command given as argument or the
    empty string (for: not found) See `system("Singular")'. See
    `system("getenv","PATH")'.

`system("freegb",' ideal, i2, i3 `)'
    returns the standrda basis in the shift algebra i(with i3
    variables) up to degree i2. See `system("opp")'.

`system("getenv",' string_expression `)'
    returns the value of the shell environment variable given as
    the second argument. The return type is string.

`system("getPrecDigits")'
    returns the precision for floating point numbers

```

```

`system("gmsnf",' ideal, ideal, matrix,int, int `)'
    Gauss-Manin system: for gmspoly.lib, gmssing.lib

`system("HC")'
    returns the degree of the "highest corner" from the last std
    computation (or 0).

`system("hessenberg",' matrix `)'
    returns the Hessenberg matrix (via QR algorithm).

`system("install",' s1, s2, p3, i4 `)'
    install a new method p3 for s2 for the newstruct type s1. s2
    must be a reserved operator with i4 operands (i4 may be
    1,2,3; use 4 for more than 3 or a varying number of arguments)
    See *Note Commands for user defined types::.

`system("LLL",' B `)'
    B must be a matrix or an intmat. Interface to NTLs LLL
    (Exact Arithmetic Variant over ZZ). Returns the same type as
    the input.
    B is an m x n matrix, viewed as m rows of n-vectors. m may
    be less than, equal to, or greater than n, and the rows need
    not be linearly independent. B is transformed into an
    LLL-reduced basis. The first m-rank(B) rows of B are zero.
    More specifically, elementary row transformations are
    performed on B so that the non-zero rows of new-B form an
    LLL-reduced basis for the lattice spanned by the rows of
    old-B.

`system("nblocks")' or `system("nblocks",' ring_name `)'
    returns the number of blocks of the given ring, or of the
    current basering, if no second argument is given. The return
    type is int.

`system("nc_hilb",' ideal, int, [...]) `)'
    internal support for nCHilb.lib, return nothing

`system("neworder",' ideal `)'
    string of the ring variables in an heurically good order for
    `char_series'

`system("newstruct")'
    list all newstruct data types.

`system("opp",' ring `)'
    returns the opposite ring.

`system("oppose",' ring R, poly p `)'
    returns the opposite polynomial of p from R.

`system("pcvLAddL",' list, list `)'
    `system("pcvPMulL",' poly, list `)'
    `system("pcvMinDeg",' poly `)'
    `system("pcvP2CV",' list, int, int `)'
    `system("pcvCV2P",' list, int, int `)'
    `system("pcvDim",' int, int `)'
    `system("pcvBasis",' int, int `)' internal for mondromy.lib

```

```

`system("pid")'
    returns the process number as int (for creating unique names).

`system("random")' or `system("random",' int `)'
    returns or sets the seed of the random generator.

`system("reserve",' int `)'
    reserve a port and listen with the given backlog. (see
    `system("reservedLink")').

`system("reservedLink")'
    accept a connect at the reserved port and return a
    (write-only) link to it. (see `system("reserve")').

`system("semaphore",' string, int `)'
    operations for semaphores: string may be `"init"', `"exists"',
    `"acquire"', `"try_acquire"', `"release"', `"get_value"', and
    int is the number of the semaphore. Returns -2 for wrong
    command, -1 for error or the result of the command.

`system("semic",' list, list `)'
    or `system("semic",' list, list, int `)' computes from list
    of spectrum numbers and list of spectrum numbers the
    semicontinuity index (qh, if 3rd argument is 1).

`system("setenv",'string_expression, string_expression`)'
    sets the shell environment variable given as the second
    argument to the value given as the third argument. Returns
    the third argument. Might not be available on all platforms.

`system("sh",' string_expression `)'
    shell escape, returns the return code of the shell as int.
    The string is sent literally to the shell.

`system("shrinktest",' poly, i2 `)'
    internal for shift algebra (with i2 variables): shrink the
    poly

`system("Singular")'
    returns the absolute (path) name of the running SINGULAR as
    string.

`system("SingularLib")'
    returns the colon seperated library search path name as
    string.

`system("spadd",' list, list `)'
    or `system("spadd",' list, list, int `)' computes from list
    of spectrum numbers and list of spectrum numbers the sum of
    the lists.

`system("spectrum",' poly `)'
    or `system("spectrum",' poly, int `)'

`system("spmul",' list, int `)'
    or `system("spmul",' list, list, int `)' computes from list
    of spectrum numbers the multiple of it.

```

```

`system("std_syz",' module, int `)'
    compute a partial groebner base of a module, stopp after the
    given column

`system("stest",' poly, i2, i3, i4 `)'
    internal for shift algebra (with i4 variables): shift the
    poly by i2, up to degree i3

`system("tensorModuleMult",' int, module `)'
    internal for sheafcoh.lib (see id_TensorModuleMult)

`system("twostd",' ideal `)'
    returns the two-sided standard basis of the two-sided ideal.

`system("uname")'
    returns a string identifying the architecture for which
    SINGULAR was compiled.

`system("version")'
    returns the version number of SINGULAR as int.  (Version
    a-b-c-d returns a*10000+b*1000+c*100+d)

`system("with")'
    without an argument: returns a string describing the current
    version of SINGULAR, its build options, the used path names
    and other configurations
    with a string argument: test for that feature and return an
    int.

`system("--cpus")'
    returns the number of available cpu cores as int (for using
    multiple cores).  (see `system("cpu")').

`system("'-'")'
    prints the values of all options.

`system("'--long_option_name`")'
    returns the value of the (command-line) option
    long_option_name. The type of the returned value is either
    string or int.  *Note Command line options::, for more info.

`system("'--long_option_name`",' expression`)'
    sets the value of the (command-line) option long_option_name
    to the value given by the expression. Type of the expression
    must be string, or int. *Note Command line options::, for
    more info. Among others, this can be used for setting the
    seed of the random number generator, the used help browser,
    the minimal display time, or the timer resolution.

`*Example:*'
    // a listing of the current directory:
    system("sh","ls");
    // execute a shell, return to SINGULAR with exit:
    system("sh","sh");
    string unique_name="/tmp/xx"+string(system("pid"));
    unique_name;
    ==> /tmp/xx4711

```

```

system("uname")
==> ix86-Linux
system("getenv", "PATH");
==> /bin:/usr/bin:/usr/local/bin
system("Singular");
==> /usr/local/bin/Singular
// report value of all options
system("--");
==> // --batch          0
==> // --execute        0
==> // --sdb            0
==> // --echo           1
==> // --profile        0
==> // --quiet          1
==> // --sort           0
==> // --random         12345678
==> // --no-tty         1
==> // --user-option    0
==> // --allow-net      0
==> // --browser        0
==> // --cntrlc         0
==> // --emacs          0
==> // --no-stdlib      0
==> // --no-rc          1
==> // --no-warn        0
==> // --no-out         0
==> // --no-shell       0
==> // --min-time       "0.5"
==> // --cpus           4
==> // --MPport         0
==> // --MPhost         0
==> // --link           0
==> // --ticks-per-sec  1
// set minimal display time to 0.02 seconds
system("--min-time", "0.02");
// set timer resolution to 0.01 seconds
system("--ticks-per-sec", 100);
// re-seed random number generator
system("--random", 12345678);
// allow your web browser to access HTML pages from the net
system("--allow-net", 1);
// and set help browser to firefox
system("--browser", "firefox");
==> // ** Could not get 'DataDir'.
==> // ** Either set environment variable 'SINGULAR_DATA_DIR' to
→ 'DataDir',
==> // ** or make sure that 'DataDir' is at "/home/hannes/singular/doc/.
→ ./Sin\
    gular/./share/"
==> // ** Could not get 'IdxFile'.
==> // ** Either set environment variable 'SINGULAR_IDX_FILE' to
→ 'IdxFile',
==> // ** Could not get 'DataDir'.
==> // ** Either set environment variable 'SINGULAR_DATA_DIR' to
→ 'DataDir',
==> // ** or make sure that 'DataDir' is at "/home/hannes/singular/doc/.
→ ./Sin\
    gular/./share/"

```



```

==> // ** or make sure that 'IdxFile' is at "%D/singular/singular.idx"
==> // ** resource `x` not found
==> // ** Setting help browser to 'dummy'.

```

2.6 Finite dimensional free algebra quotients

REMARK:

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

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

```

class sage.algebras.free_algebra_quotient.FreeAlgebraQuotient ( A, mons, mats,
                                                                names)
    Bases: sage.structure.unique_representation.UniqueRepresentation,
            sage.rings.ring.Algebra, object

```

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

EXAMPLES:

Quaternion algebra defined in terms of three generators:

```

sage: n = 3
sage: A = FreeAlgebra(QQ,n,'i')
sage: F = A.monoid()
sage: i, j, k = F.gens()
sage: mons = [ F(1), i, j, k ]
sage: M = MatrixSpace(QQ,4)
sage: mats = [M([0,1,0,0, -1,0,0,0, 0,0,0,-1, 0,0,1,0]), M([0,0,1,0, 0,0,0,1, -
→1,0,0,0, 0,-1,0,0]), M([0,0,0,1, 0,0,-1,0, 0,1,0,0, -1,0,0,0]) ]
sage: H3.<i,j,k> = FreeAlgebraQuotient(A,mons,mats)
sage: x = 1 + i + j + k
sage: x
1 + i + j + k
sage: x**128
-170141183460469231731687303715884105728 +
→170141183460469231731687303715884105728*i +
→170141183460469231731687303715884105728*j +
→170141183460469231731687303715884105728*k

```

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

```

sage: n = 2
sage: A = FreeAlgebra(QQ,n,'x')
sage: F = A.monoid()
sage: i, j = F.gens()
sage: mons = [ F(1), i, j, i*j ]
sage: r = len(mons)
sage: M = MatrixSpace(QQ,r)
sage: mats = [M([0,1,0,0, -1,0,0,0, 0,0,0,-1, 0,0,1,0]), M([0,0,1,0, 0,0,0,1, -
→1,0,0,0, 0,-1,0,0]) ]

```

```

sage: H2.<i,j> = A.quotient(mons,mats)
sage: k = i*j
sage: x = 1 + i + j + k
sage: x
1 + i + j + i*j
sage: x**128
-170141183460469231731687303715884105728 +
↪170141183460469231731687303715884105728*i +
↪170141183460469231731687303715884105728*j +
↪170141183460469231731687303715884105728*i*j

```

Element

alias of `FreeAlgebraQuotientElement`

dimension ()

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

EXAMPLES:

```

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

```

free_algebra ()

The free algebra generating the algebra.

EXAMPLES:

```

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

```

gen (i)

The i-th generator of the algebra.

EXAMPLES:

```

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

```

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

```

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

```

Negative indexing into the generators is not supported:

```

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

```

matrix_action ()

EXAMPLES:

```

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

```

module ()

The free module of the algebra.

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

monoid ()

The free monoid of generators of the algebra.

EXAMPLES:

```

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

```

monomial_basis ()

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

EXAMPLES:

```

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

```

ngens ()

The number of generators of the algebra.

EXAMPLES:

```

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

```

rank ()

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

EXAMPLES:

```

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

```

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

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

INPUT:

- R – a commutative ring

OUTPUT:

- Q – quaternion algebra

- gens – generators for Q

EXAMPLES:

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

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

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

2.7 Free algebra quotient elements

AUTHORS:

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

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

Bases: sage.structure.element.AlgebraElement

Create the element x of the FreeAlgebraQuotient A.

EXAMPLES:

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

vector ()

Return underlying vector representation of this element.

EXAMPLES:

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

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

EXAMPLES:

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

Of course this is testing the data type:

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


FINITE DIMENSIONAL ALGEBRAS

3.1 Finite-Dimensional Algebras

class sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra. **FiniteDimensionalAlgebra**

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.rings.ring.Algebra

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

INPUT:

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

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

EXAMPLES:

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

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

Element

alias of `FiniteDimensionalAlgebraElement`

base_extend (F)

Return `self` base changed to the field `F`.

EXAMPLES:

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

basis ()

Return a list of the basis elements of `self`.

EXAMPLES:

```
sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1, 0], [0, 1]]),
↪Matrix([[0, 1], [0, 0]])])
sage: A.basis()
[e0, e1]
```

cardinality ()

Return the cardinality of `self`.

EXAMPLES:

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

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

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

degree ()

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

EXAMPLES:

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

from_base_ring (x)**gen (i)**

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

EXAMPLES:


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

ideal (gens=None, given_by_matrix=False)

Return the right ideal of self generated by gens .

INPUT:

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

EXAMPLES:

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

is_associative ()

Return True if self is associative.

EXAMPLES:

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

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

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

is_commutative ()

Return True if self is commutative.

EXAMPLES:

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

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

is_finite ()

Return True if the cardinality of self is finite.

EXAMPLES:

```

sage: A = FiniteDimensionalAlgebra(GF(7), [Matrix([[1, 0], [0, 1]]), ↵
↵Matrix([[0, 1], [2, 3]])])
sage: A.is_finite()
True

sage: B = FiniteDimensionalAlgebra(RR, [Matrix([[1, 0], [0, 1]]), Matrix([[0, ↵
↵1], [2, 3]])])
sage: B.is_finite()
False

sage: C = FiniteDimensionalAlgebra(RR, [])
sage: C.is_finite()
True

```

is_unitary ()

Return True if self has a two-sided multiplicative identity element.

Warning: This uses linear algebra; thus expect wrong results when the base ring is not a field.

EXAMPLES:

```

sage: A = FiniteDimensionalAlgebra(QQ, [])
sage: A.is_unitary()
True

sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1, 0], [0, 1]]), ↵
↵Matrix([[0, 1], [-1, 0]])])
sage: B.is_unitary()
True

sage: C = FiniteDimensionalAlgebra(QQ, [Matrix([[0, 0], [0, 0]]), ↵
↵Matrix([[0, 0], [0, 0]])])
sage: C.is_unitary()
False

sage: D = FiniteDimensionalAlgebra(QQ, [Matrix([[1, 0], [0, 1]]), ↵
↵Matrix([[1, 0], [0, 1]])])
sage: D.is_unitary()
False

sage: E = FiniteDimensionalAlgebra(QQ, [Matrix([[1, 0], [1, 0]]), ↵
↵Matrix([[0, 1], [0, 1]])])
sage: E.is_unitary()
False

sage: F = FiniteDimensionalAlgebra(QQ, [Matrix([[1, 0, 0], [0, 1, 0], [0, 0, 1]]), ↵
↵Matrix([[0, 1, 0], [0, 0, 0], [0, 0, 0]]), Matrix([[0, 0, 1], [0, 0, 0], [1, 0, 0]])])
sage: F.is_unitary()
True

sage: G = FiniteDimensionalAlgebra(QQ, [Matrix([[1, 0, 0], [0, 1, 0], [0, 0, 1]]), ↵
↵Matrix([[0, 1, 0], [0, 0, 0], [0, 0, 0]]), Matrix([[0, 1, 0], [0, 0, 0], [1, 0, 0]])])

```

```
sage: G.is_unitary() # Unique right identity, but no left identity.
False
```

is_zero ()

Return True if self is the zero ring.

EXAMPLES:

```
sage: A = FiniteDimensionalAlgebra(QQ, [])
sage: A.is_zero()
True

sage: B = FiniteDimensionalAlgebra(GF(7), [Matrix([0])])
sage: B.is_zero()
False
```

left_table ()

Return the list of matrices for left multiplication by the basis elements.

EXAMPLES:

```
sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0], [0,1]]),
↪Matrix([[0,1], [-1,0]])])
sage: T = B.left_table(); T
(
[1 0]  [ 0  1]
[0 1], [-1  0]
)
```

We check immutability:

```
sage: T[0] = "vandalized by h4xx0r"
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
sage: T[1][0] = [13, 37]
Traceback (most recent call last):
...
ValueError: matrix is immutable; please change a copy instead
(i.e., use copy(M) to change a copy of M).
```

maximal_ideal ()

Compute the maximal ideal of the local algebra self.

Note: self must be unitary, commutative, associative and local (have a unique maximal ideal).

OUTPUT:

•*FiniteDimensionalAlgebraIdeal* ; the unique maximal ideal of self . If self is not a local algebra, a *ValueError* is raised.

EXAMPLES:

```
sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1, 0], [0, 1]]),
↪Matrix([[0, 1], [0, 0]])])
sage: A.maximal_ideal()
Ideal (0, e1) of Finite-dimensional algebra of degree 2 over Finite Field of
↪size 3
```

```

sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,1,0], [0,0,0]]),
↪Matrix([[0,1,0], [0,0,0], [0,0,0]]), Matrix([[0,0,0], [0,0,0], [0,0,1]])])
sage: B.maximal_ideal()
Traceback (most recent call last):
...
ValueError: algebra is not local

```

maximal_ideals ()

Return a list consisting of all maximal ideals of `self`.

EXAMPLES:

```

sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1, 0], [0, 1]]),
↪Matrix([[0, 1], [0, 0]])])
sage: A.maximal_ideals()
[Ideal (e1) of Finite-dimensional algebra of degree 2 over Finite Field of
↪size 3]

sage: B = FiniteDimensionalAlgebra(QQ, [])
sage: B.maximal_ideals()
[]

```

ngens ()

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

EXAMPLES:

```

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

```

one ()

Return the multiplicative identity element of `self`, if it exists.

EXAMPLES:

```

sage: A = FiniteDimensionalAlgebra(QQ, [])
sage: A.one()
0

sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0], [0,1]]),
↪Matrix([[0,1], [-1,0]])])
sage: B.one()
e0

sage: C = FiniteDimensionalAlgebra(QQ, [Matrix([[0,0], [0,0]]),
↪Matrix([[0,0], [0,0]])])
sage: C.one()
Traceback (most recent call last):
...
TypeError: algebra is not unitary

sage: D = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,1,0], [0,0,1]]),
↪Matrix([[0,1,0], [0,0,0], [0,0,0]]), Matrix([[0,0,1], [0,0,0], [1,0,0]])])
sage: D.one()
e0

```

```

sage: E = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,1,0], [0,0,1]]),
↪Matrix([[0,1,0], [0,0,0], [0,0,0]]), Matrix([[0,1,0], [0,0,0], [1,0,0]]))
sage: E.one()
Traceback (most recent call last):
...
TypeError: algebra is not unitary

```

primary_decomposition ()

Return the primary decomposition of self.

Note: self must be unitary, commutative and associative.

OUTPUT:

- a list consisting of the quotient maps $\text{self} \rightarrow A$, with A running through the primary factors of self

EXAMPLES:

```

sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1, 0], [0, 1]]),
↪Matrix([[0, 1], [0, 0]]))
sage: A.primary_decomposition()
[Morphism from Finite-dimensional algebra of degree 2 over Finite Field of
↪size 3 to Finite-dimensional algebra of degree 2 over Finite Field of size
↪3 given by matrix [1 0]
[0 1]]

sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,1,0], [0,0,0]]),
↪Matrix([[0,1,0], [0,0,0], [0,0,0]]), Matrix([[0,0,0], [0,0,0], [0,0,1]]))
sage: B.primary_decomposition()
[Morphism from Finite-dimensional algebra of degree 3 over Rational Field to
↪Finite-dimensional algebra of degree 1 over Rational Field given by matrix
↪[0]
[0]
[1], Morphism from Finite-dimensional algebra of degree 3 over Rational Field
↪to Finite-dimensional algebra of degree 2 over Rational Field given by
↪matrix [1 0]
[0 1]
[0 0]]

```

quotient_map (ideal)

Return the quotient of self by ideal.

INPUT:

- ideal – a FiniteDimensionalAlgebraIdeal

OUTPUT:

- *FiniteDimensionalAlgebraMorphism*; the quotient homomorphism

EXAMPLES:

```

sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1, 0], [0, 1]]),
↪Matrix([[0, 1], [0, 0]]))
sage: q0 = A.quotient_map(A.zero_ideal())
sage: q0
Morphism from Finite-dimensional algebra of degree 2 over Finite Field of
↪size 3 to Finite-dimensional algebra of degree 2 over Finite Field of size
↪3 given by matrix

```

```

[1 0]
[0 1]
sage: q1 = A.quotient_map(A.ideal(A.gen(1)))
sage: q1
Morphism from Finite-dimensional algebra of degree 2 over Finite Field of
↳size 3 to Finite-dimensional algebra of degree 1 over Finite Field of size
↳3 given by matrix
[1]
[0]

```

random_element (*args, **kwargs)

Return a random element of self .

Optional input parameters are propagated to the `random_element` method of the underlying `VectorSpace` .

EXAMPLES:

```

sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1, 0], [0, 1]]),
↳Matrix([[0, 1], [0, 0]])])
sage: A.random_element() # random
e0 + 2*e1

sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,1,0], [0,0,0]]),
↳Matrix([[0,1,0], [0,0,0], [0,0,0]]), Matrix([[0,0,0], [0,0,0], [0,0,1]])])
sage: B.random_element(num_bound=1000) # random
215/981*e0 + 709/953*e1 + 931/264*e2

```

table ()

Return the multiplication table of self , as a list of matrices for right multiplication by the basis elements.

EXAMPLES:

```

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

```

3.2 Elements of Finite Algebras

class sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element. **FiniteDim**

Bases: sage.structure.element.AlgebraElement

Create an element of a *FiniteDimensionalAlgebra* using a multiplication table.

INPUT:

- *A* – a *FiniteDimensionalAlgebra* which will be the parent
- *elt* – vector, matrix or element of the base field (default: None)
- *check* – boolean (default: True); if False and *elt* is a matrix, assume that it is known to be the matrix of an element

If `elt` is a vector, it is interpreted as a vector of coordinates with respect to the given basis of A . If `elt` is a matrix, it is interpreted as a multiplication matrix with respect to this basis.

EXAMPLES:

```
sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1,0], [0,1]]), Matrix([[0,1],
↪[0,0]])])
sage: A(17)
2*e0
sage: A([1,1])
e0 + e1
```

characteristic_polynomial ()

Return the characteristic polynomial of `self`.

Note: This function just returns the characteristic polynomial of the matrix of right multiplication by `self`. This may not be a very meaningful invariant if the algebra is not unitary and associative.

EXAMPLES:

```
sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,1,0], [0,0,0]]),
↪Matrix([[0,1,0], [0,0,0], [0,0,0]]), Matrix([[0,0,0], [0,0,0], [0,0,1]])])
sage: B(0).characteristic_polynomial()
x^3
sage: b = B.random_element()
sage: f = b.characteristic_polynomial(); f # random
x^3 - 8*x^2 + 16*x
sage: f(b) == 0
True
```

inverse ()

Return the two-sided multiplicative inverse of `self`, if it exists.

This assumes that the algebra to which `self` belongs is associative.

Note: If an element of a finite-dimensional unitary associative algebra over a field admits a left inverse, then this is the unique left inverse, and it is also a right inverse.

EXAMPLES:

```
sage: C = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0], [0,1]]),
↪Matrix([[0,1], [-1,0]])])
sage: C([1,2]).inverse()
1/5*e0 - 2/5*e1
```

is_invertible ()

Return `True` if `self` has a two-sided multiplicative inverse.

This assumes that the algebra to which `self` belongs is associative.

Note: If an element of a unitary finite-dimensional algebra over a field admits a left inverse, then this is the unique left inverse, and it is also a right inverse.

EXAMPLES:

```

sage: C = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0], [0,1]]),
↳Matrix([[0,1], [-1,0]])])
sage: C([1,2]).is_invertible()
True
sage: C(0).is_invertible()
False

```

is_nilpotent ()

Return True if self is nilpotent.

EXAMPLES:

```

sage: C = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0], [0,1]]),
↳Matrix([[0,1], [0,0]])])
sage: C([1,0]).is_nilpotent()
False
sage: C([0,1]).is_nilpotent()
True

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

```

is_zerodivisor ()

Return True if self is a left or right zero-divisor.

EXAMPLES:

```

sage: C = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0], [0,1]]),
↳Matrix([[0,1], [0,0]])])
sage: C([1,0]).is_zerodivisor()
False
sage: C([0,1]).is_zerodivisor()
True

```

left_matrix ()

Return the matrix for multiplication by self from the left.

EXAMPLES:

```

sage: C = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,0,0], [0,0,0]]),
↳Matrix([[0,1,0], [0,0,0], [0,0,0]]), Matrix([[0,0,0], [0,1,0], [0,0,1]])])
sage: C([1,2,0]).left_matrix()
[1 0 0]
[0 1 0]
[0 2 0]

```

matrix ()

Return the matrix for multiplication by self from the right.

EXAMPLES:

```

sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,1,0], [0,0,0]]),
↳Matrix([[0,1,0], [0,0,0], [0,0,0]]), Matrix([[0,0,0], [0,0,0], [0,0,1]])])
sage: B(5).matrix()
[5 0 0]
[0 5 0]
[0 0 5]

```


minimal_polynomial ()

Return the minimal polynomial of `self`.

EXAMPLES:

```
sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,1,0], [0,0,0]]),
↪Matrix([[0,1,0], [0,0,0], [0,0,0]]), Matrix([[0,0,0], [0,0,0], [0,0,1]])])
sage: B(0).minimal_polynomial()
x
sage: b = B.random_element()
sage: f = b.minimal_polynomial(); f # random
x^3 + 1/2*x^2 - 7/16*x + 1/16
sage: f(b) == 0
True
```

monomial_coefficients (copy=True)

Return a dictionary whose keys are indices of basis elements in the support of `self` and whose values are the corresponding coefficients.

INPUT:

- `copy` – (default: `True`) if `self` is internally represented by a dictionary `d`, then make a copy of `d`; if `False`, then this can cause undesired behavior by mutating `d`

EXAMPLES:

```
sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0], [0,1]]),
↪Matrix([[0,1], [-1,0]])])
sage: elt = B(Matrix([[1,1], [-1,1]]))
sage: elt.monomial_coefficients()
{0: 1, 1: 1}
```

vector ()

Return `self` as a vector.

EXAMPLES:

```
sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,1,0], [0,0,0]]),
↪Matrix([[0,1,0], [0,0,0], [0,0,0]]), Matrix([[0,0,0], [0,0,0], [0,0,1]])])
sage: B(5).vector()
(5, 0, 5)
```

3.3 Ideals of Finite Algebras

class `sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_ideal.FiniteDimen`

Bases: `sage.rings.ideal.Ideal_generic`

An ideal of a *FiniteDimensionalAlgebra*.

INPUT:

- `A` – a finite-dimensional algebra
- `gens` – the generators of this ideal
- `given_by_matrix` – (default: `False`) whether the basis matrix is given by `gens`

EXAMPLES:

```
sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1, 0], [0, 1]]), Matrix([[0, 1],
↪[1], [0, 0]]))
sage: A.ideal(A([0,1]))
Ideal (e1) of Finite-dimensional algebra of degree 2 over Finite Field of size 3
```

basis_matrix ()

Return the echelonized matrix whose rows form a basis of self .

EXAMPLES:

```
sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1, 0], [0, 1]]), ↪
↪Matrix([[0, 1], [0, 0]]))
sage: I = A.ideal(A([1,1]))
sage: I.basis_matrix()
[1 0]
[0 1]
```

vector_space ()

Return self as a vector space.

EXAMPLES:

```
sage: A = FiniteDimensionalAlgebra(GF(3), [Matrix([[1, 0], [0, 1]]), ↪
↪Matrix([[0, 1], [0, 0]]))
sage: I = A.ideal(A([1,1]))
sage: I.vector_space()
Vector space of degree 2 and dimension 2 over Finite Field of size 3
Basis matrix:
[1 0]
[0 1]
```

3.4 Morphisms Between Finite Algebras

class sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism. **FiniteDi**

Bases: sage.rings.homset.RingHomset_generic

Set of morphisms between two finite-dimensional algebras.

zero ()

Construct the zero morphism of self .

EXAMPLES:

```
sage: A = FiniteDimensionalAlgebra(QQ, [Matrix([1])])
sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1, 0], [0, 1]]), Matrix([[0, 1],
↪[1], [0, 0]])])
sage: H = Hom(A, B)
sage: H.zero()
Morphism from Finite-dimensional algebra of degree 1 over Rational Field to
Finite-dimensional algebra of degree 2 over Rational Field given by matrix
[0 0]
```

class sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism. **FiniteDi**

Bases: sage.rings.morphism.RingHomomorphism_im_gens

Create a morphism between two *finite-dimensional algebras*.

INPUT:

- parent – the parent homset
- f – matrix of the underlying k -linear map
- unitary – boolean (default: True); if True and check is also True, raise a ValueError unless A and B are unitary and f respects unit elements
- check – boolean (default: True); check whether the given k -linear map really defines a (not necessarily unitary) k -algebra homomorphism

The algebras A and B must be defined over the same base field.

EXAMPLES:

```
sage: from sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_
      ↪morphism import FiniteDimensionalAlgebraMorphism
sage: A = FiniteDimensionalAlgebra(QQ, [Matrix([[1, 0], [0, 1]]), Matrix([[0, 1],
      ↪[0, 0]])])
sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([1])])
sage: H = Hom(A, B)
sage: f = H(Matrix([[1], [0]]))
sage: f.domain() is A
True
sage: f.codomain() is B
True
sage: f(A.basis()[0])
e
sage: f(A.basis()[1])
0
```

Todo

An example illustrating unitary flag.

inverse_image (I)

Return the inverse image of I under self.

INPUT:

- I – FiniteDimensionalAlgebraIdeal, an ideal of self.codomain()

OUTPUT:

– FiniteDimensionalAlgebraIdeal, the inverse image of I under self.

EXAMPLES:

```
sage: A = FiniteDimensionalAlgebra(QQ, [Matrix([[1, 0], [0, 1]]), Matrix([[0,
      ↪1], [0, 0]])])
sage: I = A.maximal_ideal()
```

```
sage: q = A.quotient_map(I)
sage: B = q.codomain()
sage: q.inverse_image(B.zero_ideal()) == I
True
```

matrix ()

Return the matrix of self .

EXAMPLES:

```
sage: A = FiniteDimensionalAlgebra(QQ, [Matrix([[1, 0], [0, 1]]), Matrix([[0, ↵
↵1], [0, 0]])])
sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([1])])
sage: M = Matrix([[1], [0]])
sage: H = Hom(A, B)
sage: f = H(M)
sage: f.matrix() == M
True
```

NAMED ASSOCIATIVE ALGEBRAS

4.1 Affine nilTemperley Lieb Algebra of type A

class sage.algebras.affine_nil_temperley_lieb. **AffineNilTemperleyLiebTypeA** (*n*,
R=Integer
Ring,
pre-
fix='a'

Bases: sage.combinat.free_module.CombinatorialFreeModule

Constructs the affine nilTemperley Lieb algebra of type $A_{n-1}^{(1)}$ as used in [Pos2005].

INPUT:

• *n* – a positive integer

The affine nilTemperley Lieb algebra is generated by a_i for $i = 0, 1, \dots, n-1$ subject to the relations $a_i a_i = a_i a_{i+1} a_i = a_{i+1} a_i a_{i+1} = 0$ and $a_i a_j = a_j a_i$ for $i - j \not\equiv \pm 1$, where the indices are taken modulo n .

EXAMPLES:

```
sage: A = AffineNilTemperleyLiebTypeA(4)
sage: a = A.algebra_generators(); a
Finite family {0: a0, 1: a1, 2: a2, 3: a3}
sage: a[1]*a[2]*a[0] == a[1]*a[0]*a[2]
True
sage: a[0]*a[3]*a[0]
0
sage: A.an_element()
2*a0 + 1 + 3*a1 + a0*a1*a2*a3
```

algebra_generator (*i*)

EXAMPLES:

```
sage: A = AffineNilTemperleyLiebTypeA(3)
sage: A.algebra_generator(1)
a1
sage: A = AffineNilTemperleyLiebTypeA(3, prefix = 't')
sage: A.algebra_generator(1)
t1
```

algebra_generators ()

Returns the generators a_i for $i = 0, 1, 2, \dots, n-1$.

EXAMPLES:

```

sage: A = AffineNilTemperleyLiebTypeA(3)
sage: a = A.algebra_generators(); a
Finite family {0: a0, 1: a1, 2: a2}
sage: a[1]
a1

```

has_no_braid_relation (*w, i*)

Assuming that w contains no relations of the form s_i^2 or $s_i s_{i+1} s_i$ or $s_i s_{i-1} s_i$, tests whether ws_i contains terms of this form.

EXAMPLES:

```

sage: A = AffineNilTemperleyLiebTypeA(5)
sage: W = A.weyl_group()
sage: s=W.simple_reflections()
sage: A.has_no_braid_relation(s[2]*s[1]*s[0]*s[4]*s[3],0)
False
sage: A.has_no_braid_relation(s[2]*s[1]*s[0]*s[4]*s[3],2)
True
sage: A.has_no_braid_relation(s[4],2)
True

```

index_set ()

EXAMPLES:

```

sage: A = AffineNilTemperleyLiebTypeA(3)
sage: A.index_set()
(0, 1, 2)

```

one_basis ()

Returns the unit of the underlying Weyl group, which index the one of this algebra, as per `AlgebrasWithBasis.ParentMethods.one_basis()`.

EXAMPLES:

```

sage: A = AffineNilTemperleyLiebTypeA(3)
sage: A.one_basis()
[1 0 0]
[0 1 0]
[0 0 1]
sage: A.one_basis() == A.weyl_group().one()
True
sage: A.one()
1

```

product_on_basis (*w, w1*)

Returns $a_w a_{w1}$, where w and $w1$ are in the Weyl group assuming that w does not contain any braid relations.

EXAMPLES:

```

sage: A = AffineNilTemperleyLiebTypeA(5)
sage: W = A.weyl_group()
sage: s = W.simple_reflections()
sage: [A.product_on_basis(s[1],x) for x in s]
[a1*a0, 0, a1*a2, a3*a1, a4*a1]

sage: a = A.algebra_generators()

```

```

sage: x = a[1] * a[2]
sage: x
a1*a2
sage: x * a[1]
0
sage: x * a[2]
0
sage: x * a[0]
a1*a2*a0

sage: [x * a[1] for x in a]
[a0*a1, 0, a2*a1, a3*a1, a4*a1]

sage: w = s[1]*s[2]*s[1]
sage: A.product_on_basis(w,s[1])
Traceback (most recent call last):
...
AssertionError

```

weyl_group ()
EXAMPLES:

```

sage: A = AffineNilTemperleyLiebTypeA(3)
sage: A.weyl_group()
Weyl Group of type ['A', 2, 1] (as a matrix group acting on the root space)

```

4.2 Diagram and Partition Algebras

AUTHORS:

- Mike Hansen (2007): Initial version
- Stephen Doty, Aaron Lauve, George H. Seelinger (2012): Implementation of partition, Brauer, Temperley–Lieb, and ideal partition algebras
- Stephen Doty, Aaron Lauve, George H. Seelinger (2015): Implementation of `*Diagram` classes and other methods to improve diagram algebras.

class `sage.combinat.diagram_algebras.AbstractPartitionDiagram (parent, d)`
Bases: `sage.combinat.set_partition.SetPartition`

Abstract base class for partition diagrams.

This class represents a single partition diagram, that is used as a basis key for a diagram algebra element. A partition diagram should be a partition of the set $\{1, \dots, k, -1, \dots, -k\}$. Each such set partition is regarded as a graph on nodes $\{1, \dots, k, -1, \dots, -k\}$ arranged in two rows, with nodes $1, \dots, k$ in the top row from left to right and with nodes $-1, \dots, -k$ in the bottom row from left to right, and an edge connecting two nodes if and only if the nodes lie in the same subset of the set partition.

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: pd = da.AbstractPartitionDiagrams(da.partition_diagrams, 2)
sage: pd1 = da.AbstractPartitionDiagram(pd, [[1,2],[-1,-2]])
sage: pd2 = da.AbstractPartitionDiagram(pd, [[1,2],[-1,-2]])
sage: pd1
{{-2, -1}, {1, 2}}

```

```

sage: pd1 == pd2
True
sage: pd1 == [[1,2],[-1,-2]]
True
sage: pd1 == ((-2,-1),(2,1))
True
sage: pd1 == SetPartition([[1,2],[-1,-2]])
True
sage: pd3 = da.AbstractPartitionDiagram(pd, [[1,-2],[-1,2]])
sage: pd1 == pd3
False
sage: pd4 = da.AbstractPartitionDiagram(pd, [[1,2],[3,4]])
Traceback (most recent call last):
...
ValueError: this does not represent two rows of vertices

```

base_diagram ()

Return the underlying implementation of the diagram.

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: pd = da.AbstractPartitionDiagrams(da.partition_diagrams, 2)
sage: pd([[1,2],[-1,-2]]).base_diagram() == ((-2,-1),(1,2))
True

```

check ()

Check the validity of the input for the diagram.

compose (other)

Compose *self* with *other*.

The composition of two diagrams X and Y is given by placing X on top of Y and removing all loops.

OUTPUT:

A tuple where the first entry is the composite diagram and the second entry is how many loop were removed.

Note: This is not really meant to be called directly, but it works to call it this way if desired.

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: pd = da.AbstractPartitionDiagrams(da.partition_diagrams, 2)
sage: pd([[1,2],[-1,-2]]).compose(pd([[1,2],[-1,-2]]))
(({-2, -1}, {1, 2}}, 1)

```

diagram ()

Return the underlying implementation of the diagram.

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: pd = da.AbstractPartitionDiagrams(da.partition_diagrams, 2)
sage: pd([[1,2],[-1,-2]]).base_diagram() == pd([[1,2],[-1,-2]]).diagram()
True

```


global_options (**args, **kws*)

Deprecated: Use `options()` instead. See [trac ticket #18555](#) for details.

options (**get_value, **set_value*)

Set and display the global options for Brauer diagram (algebras). If no parameters are set, then the function returns a copy of the options dictionary.

The options to diagram algebras can be accessed as the method `BrauerAlgebra.options` of `BrauerAlgebra` and related classes.

OPTIONS:

- `display` – (default: `normal`) Specifies how the Brauer diagrams should be printed
 - `compact` – Using the compact representation
 - `normal` – Using the normal representation

EXAMPLES:

```
sage: R.<q> = QQ[]
sage: BA = BrauerAlgebra(2, q)
sage: E = BA([[1, 2], [-1, -2]])
sage: E
B{{-2, -1}, {1, 2}}
sage: BrauerAlgebra.options.display="compact"
sage: E
B[12/12;]
sage: BrauerAlgebra.options._reset()
```

See `GlobalOptions` for more features of these options.

propagating_number ()

Return the propagating number of the diagram.

The propagating number is the number of blocks with both a positive and negative number.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.AbstractPartitionDiagrams(da.partition_diagrams, 2)
sage: d1 = pd([[1, -2], [2, -1]])
sage: d1.propagating_number()
2
sage: d2 = pd([[1, 2], [-2, -1]])
sage: d2.propagating_number()
0
```

class `sage.combinat.diagram_algebras.AbstractPartitionDiagrams` (*diagram_func*, *order*, *category=None*)

Bases: `sage.structure.parent.Parent`, `sage.structure.unique_representation.UniqueRepresentation`

This is a class that generates partition diagrams.

The primary use of this class is to serve as basis keys for diagram algebras, but diagrams also have properties in their own right. Furthermore, this class is meant to be extended to create more efficient containers.

INPUT:

- `diagram_func` – generator; a function that can create the type of diagram desired

- order – integer or integer +1/2; the order of the diagrams

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.AbstractPartitionDiagrams(da.partition_diagrams, 2)
sage: pd
Partition diagrams of order 2
sage: [i for i in pd]
[{{-2, -1, 1, 2}},
 {{-2, -1, 2}, {1}},
 {{-2, -1, 1}, {2}},
 {{-2}, {-1, 1, 2}},
 {{-2, 1, 2}, {-1}},
 {{-2, 1}, {-1, 2}},
 {{-2, 2}, {-1, 1}},
 {{-2, -1}, {1, 2}},
 {{-2, -1}, {1}, {2}},
 {{-2}, {-1, 2}, {1}},
 {{-2, 2}, {-1}, {1}},
 {{-2}, {-1, 1}, {2}},
 {{-2, 1}, {-1}, {2}},
 {{-2}, {-1}, {1, 2}},
 {{-2}, {-1}, {1}, {2}}]
sage: pd.an_element() in pd
True
sage: elm = pd([[1,2],[-1,-2]])
sage: elm in pd
True
```

Element

alias of *AbstractPartitionDiagram*

class sage.combinat.diagram_algebras. **BrauerAlgebra** (*k, q, base_ring, prefix*)

Bases: *sage.combinat.diagram_algebras.SubPartitionAlgebra*

A Brauer algebra.

The Brauer algebra of rank k is an algebra with basis indexed by the collection of set partitions of $\{1, \dots, k, -1, \dots, -k\}$ with block size 2.

This algebra is a subalgebra of the partition algebra. For more information, see *PartitionAlgebra*.

INPUT:

- k – rank of the algebra
- q – the deformation parameter q

OPTIONAL ARGUMENTS:

- base_ring – (default None) a ring containing q ; if None then just takes the parent of q
- prefix – (default "B") a label for the basis elements

EXAMPLES:

We now define the Brauer algebra of rank 2 with parameter x over \mathbb{Z} :

```
sage: R.<x> = ZZ[]
sage: B = BrauerAlgebra(2, x, R)
sage: B
Brauer Algebra of rank 2 with parameter x
```

```

over Univariate Polynomial Ring in x over Integer Ring
sage: B.basis()
Lazy family (Term map from Brauer diagrams of order 2 to Brauer Algebra
of rank 2 with parameter x over Univariate Polynomial Ring in x
over Integer Ring(i))_{i in Brauer diagrams of order 2}
sage: b = B.basis().list()
sage: b
[B{{-2, 1}, {-1, 2}}, B{{-2, 2}, {-1, 1}}, B{{-2, -1}, {1, 2}}]
sage: b[2]
B{{-2, -1}, {1, 2}}
sage: b[2]^2
x*B{{-2, -1}, {1, 2}}
sage: b[2]^5
x^4*B{{-2, -1}, {1, 2}}

```

Note, also that since the symmetric group algebra is contained in the Brauer algebra, there is also a conversion between the two.

```

sage: R.<x> = ZZ[]
sage: B = BrauerAlgebra(2, x, R)
sage: S = SymmetricGroupAlgebra(R, 2)
sage: S([2, 1]) * B([[1, -1], [2, -2]])
B{{-2, 1}, {-1, 2}}

```

`jucys_murphy(j)`

Return the j -th generalized Jucys-Murphy element of `self`.

The j -th Jucys-Murphy element of a Brauer algebra is simply the j -th Jucys-Murphy element of the symmetric group algebra with an extra $(z - 1)/2$ term, where z is the parameter of the Brauer algebra.

REFERENCES:

EXAMPLES:

```

sage: z = var('z')
sage: B = BrauerAlgebra(3, z)
sage: B.jucys_murphy(1)
(1/2*z-1/2)*B{{-3, 3}, {-2, 2}, {-1, 1}}
sage: B.jucys_murphy(3)
-B{{-3, -2}, {-1, 1}, {2, 3}} - B{{-3, -1}, {-2, 2}, {1, 3}}
+ B{{-3, 1}, {-2, 2}, {-1, 3}} + B{{-3, 2}, {-2, 3}, {-1, 1}}
+ (1/2*z-1/2)*B{{-3, 3}, {-2, 2}, {-1, 1}}

```

class `sage.combinat.diagram_algebras.BrauerDiagram` (*parent*, *d*)

Bases: `sage.combinat.diagram_algebras.AbstractPartitionDiagram`

A Brauer diagram.

A Brauer diagram for an integer k is a partition of the set $\{1, \dots, k, -1, \dots, -k\}$ with block size 2.

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(2)
sage: bd1 = bd([[1, 2], [-1, -2]])
sage: bd2 = bd([[1, 2, -1, -2]])
Traceback (most recent call last):
...
ValueError: all blocks must be of size 2

```

bijection_on_free_nodes (*two_line=False*)

Return the induced bijection - as a list of $(x, f(x))$ values - from the free nodes on the top at the Brauer diagram to the free nodes at the bottom of *self*.

OUTPUT:

If *two_line* is *True*, then the output is the induced bijection as a two-row list (*inputs, outputs*) .

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(3)
sage: elm = bd([[1, 2], [-2, -3], [3, -1]])
sage: elm.bijection_on_free_nodes()
[[3, -1]]
sage: elm2 = bd([[1, -2], [2, -3], [3, -1]])
sage: elm2.bijection_on_free_nodes(two_line=True)
[[1, 2, 3], [-2, -3, -1]]
```

check ()

Check the validity of the input for *self*.

involution_permutation_triple (*curt=True*)

Return the involution permutation triple of *self*.

From Graham-Lehrer (see [BrauerDiagrams](#)), a Brauer diagram is a triple (D_1, D_2, π) , where:

- D_1 is a partition of the top nodes;
- D_2 is a partition of the bottom nodes;
- π is the induced permutation on the free nodes.

INPUT:

- *curt* - (default: *True*) if *True*, then return bijection on free nodes as a one-line notation (standardized to look like a permutation), else, return the honest mapping, a list of pairs $(i, -j)$ describing the bijection on free nodes

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(3)
sage: elm = bd([[1, 2], [-2, -3], [3, -1]])
sage: elm.involution_permutation_triple()
[[ (1, 2) ], [ (-3, -2) ], [ 1 ]]
sage: elm.involution_permutation_triple(curt=False)
[[ (1, 2) ], [ (-3, -2) ], [[3, -1]]]
```

is_elementary_symmetric ()

Check if is elementary symmetric.

Let (D_1, D_2, π) be the Graham-Lehrer representation of the Brauer diagram d . We say d is *elementary symmetric* if $D_1 = D_2$ and π is the identity.

Todo

Come up with a better name?

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(3)
sage: elm = bd([[1,2],[-1,-2],[3,-3]])
sage: elm.is_elementary_symmetric()
True
sage: elm2 = bd([[1,2],[-1,-3],[3,-2]])
sage: elm2.is_elementary_symmetric()
False

```

perm ()

Return the induced bijection on the free nodes of `self` in one-line notation, re-indexed and treated as a permutation.

See also:

`bijection_on_free_nodes()`

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(3)
sage: elm = bd([[1,2],[-2,-3],[3,-1]])
sage: elm.perm()
[1]

```

class `sage.combinat.diagram_algebras.BrauerDiagrams (order, category=None)`

Bases: `sage.combinat.diagram_algebras.AbstractPartitionDiagrams`

This class represents all Brauer diagrams of integer or integer +1/2 order. For more information on Brauer diagrams, see *BrauerAlgebra*.

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(3)
sage: bd.an_element() in bd
True
sage: bd.cardinality() == len(bd.list())
True

```

These diagrams also come equipped with a compact representation based on their bipartition triple representation. See the `from_involution_permutation_triple()` method for more information.

```

sage: bd = da.BrauerDiagrams(3)
sage: bd.options.display="compact"
sage: bd.list()
[/;321],
[/;312],
[23/12;1],
[/;231],
[/;132],
[13/12;1],
[/;213],
[/;123],
[12/12;1],
[23/23;1],
[13/23;1],
[12/23;1],
[23/13;1],

```

```
[13/13;1],
[12/13;1]]
sage: bd.options._reset()
```

Element

alias of *BrauerDiagram*

cardinality ()

Return the cardinality of *self*.

The number of Brauer diagrams of integer order k is $(2k - 1)!!$.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(3)
sage: bd.cardinality()
15
```

from_involution_permutation_triple (D1_D2_pi)

Construct a Brauer diagram of *self* from an involution permutation triple.

A Brauer diagram can be represented as a triple where the first entry is a list of arcs on the top row of the diagram, the second entry is a list of arcs on the bottom row of the diagram, and the third entry is a permutation on the remaining nodes. This triple is called the *involution permutation triple*. For more information, see [GL1996].

INPUT:

- *D1_D2_pi* – a list or tuple where the first entry is a list of arcs on the top of the diagram, the second entry is a list of arcs on the bottom of the diagram, and the third entry is a permutation on the free nodes.

REFERENCES:

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(4)
sage: bd.from_involution_permutation_triple([[1,2]], [[3,4]], [2,1])
[{-4, -3}, {-2, 3}, {-1, 4}, {1, 2}]
```

global_options (*args, **kws)

Deprecated: Use *options()* instead. See [trac ticket #18555](#) for details.

options (*get_value, **set_value)

Set and display the global options for Brauer diagram (algebras). If no parameters are set, then the function returns a copy of the options dictionary.

The options to diagram algebras can be accessed as the method *BrauerAlgebra.options* of *BrauerAlgebra* and related classes.

OPTIONS:

- *display* – (default: *normal*) Specifies how the Brauer diagrams should be printed
 - *compact* – Using the compact representation
 - *normal* – Using the normal representation

EXAMPLES:

```

sage: R.<q> = QQ[]
sage: BA = BrauerAlgebra(2, q)
sage: E = BA([[1, 2], [-1, -2]])
sage: E
B{{-2, -1}, {1, 2}}
sage: BrauerAlgebra.options.display="compact"
sage: E
B[12/12;]
sage: BrauerAlgebra.options._reset()

```

See `GlobalOptions` for more features of these options.

symmetric_diagrams (*l=None, perm=None*)

Return the list of brauer diagrams with symmetric placement of *l* arcs, and with free nodes permuted according to *perm*.

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: bd = da.BrauerDiagrams(4)
sage: bd.symmetric_diagrams(l=1, perm=[2, 1])
[{{-4, -3}, {-2, 1}, {-1, 2}, {3, 4}},
 {{-4, -2}, {-3, 1}, {-1, 3}, {2, 4}},
 {{-4, 1}, {-3, -2}, {-1, 4}, {2, 3}},
 {{-4, -1}, {-3, 2}, {-2, 3}, {1, 4}},
 {{-4, 2}, {-3, -1}, {-2, 4}, {1, 3}},
 {{-4, 3}, {-3, 4}, {-2, -1}, {1, 2}}]

```

class `sage.combinat.diagram_algebras. DiagramAlgebra` (*k, q, base_ring, prefix, diagrams,*
category=None)

Bases: `sage.combinat.free_module.CombinatorialFreeModule`

Abstract class for diagram algebras and is not designed to be used directly. If used directly, the class could create an “algebra” that is not actually an algebra.

class `Element` (*M, x*)

Bases: `sage.combinat.free_module.CombinatorialFreeModuleElement`

An element of a diagram algebra.

This subclass provides a few additional methods for partition algebra elements. Most element methods are already implemented elsewhere.

diagram ()

Return the underlying diagram of `self` if `self` is a basis element. Raises an error if `self` is not a basis element.

EXAMPLES:

```

sage: R.<x> = ZZ[]
sage: P = PartitionAlgebra(2, x, R)
sage: elt = 3*P([[1, 2], [-2, -1]])
sage: elt.diagram()
{{-2, -1}, {1, 2}}

```

diagrams ()

Return the diagrams in the support of `self`.

EXAMPLES:

```

sage: R.<x> = ZZ[]
sage: P = PartitionAlgebra(2, x, R)
sage: elt = 3*P([[1,2],[-2,-1]]) + P([[1,2],[-2],[-1]])
sage: elt.diagrams()
[{{-2}, {-1}, {1, 2}}, {{-2, -1}, {1, 2}}]

```

`DiagramAlgebra.one_basis()`

The following constructs the identity element of `self`.

It is not called directly; instead one should use `DA.one()` if `DA` is a defined diagram algebra.

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: R.<x> = QQ[]
sage: D = da.DiagramAlgebra(2, x, R, 'P', da.PartitionDiagrams(2))
sage: D.one_basis()
[{{-2, 2}, {-1, 1}}]

```

`DiagramAlgebra.order()`

Return the order of `self`.

The order of a partition algebra is defined as half of the number of nodes in the diagrams.

EXAMPLES:

```

sage: q = var('q')
sage: PA = PartitionAlgebra(2, q)
sage: PA.order()
2

```

`DiagramAlgebra.product_on_basis(d1, d2)`

Return the product $D_{d_1} D_{d_2}$ by two basis diagrams.

`DiagramAlgebra.set_partitions()`

Return the collection of underlying set partitions indexing the basis elements of a given diagram algebra.

Todo

Is this really necessary?

class `sage.combinat.diagram_algebras.IdealDiagrams` (*order*)

Bases: `sage.combinat.diagram_algebras.AbstractPartitionDiagrams`

All “ideal” diagrams of integer or integer +1/2 order.

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: id = da.IdealDiagrams(3)
sage: id.an_element() in id
True
sage: id.cardinality() == len(id.list())
True

```

class `sage.combinat.diagram_algebras.PartitionAlgebra` (*k, q, base_ring, prefix*)

Bases: `sage.combinat.diagram_algebras.DiagramAlgebra`

A partition algebra.

A partition algebra of rank k over a given ground ring R is an algebra with (R -module) basis indexed by the collection of set partitions of $\{1, \dots, k, -1, \dots, -k\}$. Each such set partition can be represented by a graph on nodes $\{1, \dots, k, -1, \dots, -k\}$ arranged in two rows, with nodes $1, \dots, k$ in the top row from left to right and with nodes $-1, \dots, -k$ in the bottom row from left to right, and edges drawn such that the connected components of the graph are precisely the parts of the set partition. (This choice of edges is often not unique, and so there are often many graphs representing one and the same set partition; the representation nevertheless is useful and vivid. We often speak of “diagrams” to mean graphs up to such equivalence of choices of edges; of course, we could just as well speak of set partitions.)

There is not just one partition algebra of given rank over a given ground ring, but rather a whole family of them, indexed by the elements of R . More precisely, for every $q \in R$, the partition algebra of rank k over R with parameter q is defined to be the R -algebra with basis the collection of all set partitions of $\{1, \dots, k, -1, \dots, -k\}$, where the product of two basis elements is given by the rule

$$a \cdot b = q^N (a \circ b),$$

where $a \circ b$ is the composite set partition obtained by placing the diagram (i.e., graph) of a above the diagram of b , identifying the bottom row nodes of a with the top row nodes of b , and omitting any closed “loops” in the middle. The number N is the number of connected components formed by the omitted loops.

The parameter q is a deformation parameter. Taking $q = 1$ produces the semigroup algebra (over the base ring) of the partition monoid, in which the product of two set partitions is simply given by their composition.

The Iwahori–Hecke algebra of type A (with a single parameter) is naturally a subalgebra of the partition algebra.

The partition algebra is regarded as an example of a “diagram algebra” due to the fact that its natural basis is given by certain graphs often called diagrams.

An excellent reference for partition algebras and their various subalgebras (Brauer algebra, Temperley–Lieb algebra, etc) is the paper [\[HR2005\]](#).

INPUT:

- **k** – rank of the algebra
- **q** – the deformation parameter q

OPTIONAL ARGUMENTS:

- **base_ring** – (default `None`) a ring containing q ; if `None`, then Sage automatically chooses the parent of q
- **prefix** – (default `"P"`) a label for the basis elements

EXAMPLES:

The following shorthand simultaneously defines the univariate polynomial ring over the rationals as well as the variable x :

```
sage: R.<x> = PolynomialRing(QQ)
sage: R
Univariate Polynomial Ring in x over Rational Field
sage: x
x
sage: x.parent() is R
True
```

We now define the partition algebra of rank 2 with parameter x over \mathbb{Z} :

```
sage: R.<x> = ZZ[]
sage: P = PartitionAlgebra(2, x, R)
sage: P
```

```

Partition Algebra of rank 2 with parameter x
over Univariate Polynomial Ring in x over Integer Ring
sage: P.basis().list()
[P{{-2, -1, 1, 2}}, P{{-2, -1, 2}, {1}},
 P{{-2, -1, 1}, {2}}, P{{-2}, {-1, 1, 2}},
 P{{-2, 1, 2}, {-1}}, P{{-2, 1}, {-1, 2}},
 P{{-2, 2}, {-1, 1}}, P{{-2, -1}, {1, 2}},
 P{{-2, -1}, {1}, {2}}, P{{-2}, {-1, 2}, {1}},
 P{{-2, 2}, {-1}, {1}}, P{{-2}, {-1, 1}, {2}},
 P{{-2, 1}, {-1}, {2}}, P{{-2}, {-1}, {1, 2}},
 P{{-2}, {-1}, {1}, {2}}]
sage: E = P([[1, 2], [-2, -1]]); E
P{{-2, -1}, {1, 2}}
sage: E in P.basis().list()
True
sage: E^2
x*P{{-2, -1}, {1, 2}}
sage: E^5
x^4*P{{-2, -1}, {1, 2}}
sage: (P([[2, -2], [-1, 1]]) - 2*P([[1, 2], [-1, -2]]))^2
(4*x-4)*P{{-2, -1}, {1, 2}} + P{{-2, 2}, {-1, 1}}

```

One can work with partition algebras using a symbol for the parameter, leaving the base ring unspecified. This implies that the underlying base ring is Sage's symbolic ring.

```

sage: q = var('q')
sage: PA = PartitionAlgebra(2, q); PA
Partition Algebra of rank 2 with parameter q over Symbolic Ring
sage: PA([[1, 2], [-2, -1]])^2 == q*PA([[1, 2], [-2, -1]])
True
sage: (PA([[2, -2], [1, -1]]) - 2*PA([[2, -1], [1, 2]]))^2 == (4*q-4)*PA([[1, 2], [-2, -1]]) + PA([[2, -2], [1, -1]])
True

```

The identity element of the partition algebra is the set partition $\{\{1, -1\}, \{2, -2\}, \dots, \{k, -k\}\}$:

```

sage: P = PA.basis().list()
sage: PA.one()
P{{-2, 2}, {-1, 1}}
sage: PA.one()*P[7] == P[7]
True
sage: P[7]*PA.one() == P[7]
True

```

We now give some further examples of the use of the other arguments. One may wish to “specialize” the parameter to a chosen element of the base ring:

```

sage: R.<q> = RR[]
sage: PA = PartitionAlgebra(2, q, R, prefix='B')
sage: PA
Partition Algebra of rank 2 with parameter q over
Univariate Polynomial Ring in q over Real Field with 53 bits of precision
sage: PA([[1, 2], [-1, -2]])
1.000000000000000*B{{-2, -1}, {1, 2}}
sage: PA = PartitionAlgebra(2, 5, base_ring=ZZ, prefix='B')
sage: PA
Partition Algebra of rank 2 with parameter 5 over Integer Ring
sage: (PA([[2, -2], [1, -1]]) - 2*PA([[2, -1], [1, 2]]))^2 == 16*PA([[2, -1], [1, 2]]) + PA([[2, -2], [1, -1]])

```

```
True
```

REFERENCES:

class `sage.combinat.diagram_algebras.PartitionDiagrams (order, category=None)`
 Bases: `sage.combinat.diagram_algebras.AbstractPartitionDiagrams`

This class represents all partition diagrams of integer or integer +1/2 order.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.PartitionDiagrams(3)
sage: pd.an_element() in pd
True
sage: pd.cardinality() == len(pd.list())
True
```

cardinality ()

The cardinality of partition diagrams of integer order n is the $2n$ -th Bell number.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: pd = da.PartitionDiagrams(3)
sage: pd.cardinality()
203
```

class `sage.combinat.diagram_algebras.PlanarAlgebra (k, q, base_ring, prefix)`
 Bases: `sage.combinat.diagram_algebras.SubPartitionAlgebra`

A planar algebra.

The planar algebra of rank k is an algebra with basis indexed by the collection of all planar set partitions of $\{1, \dots, k, -1, \dots, -k\}$.

This algebra is thus a subalgebra of the partition algebra. For more information, see [PartitionAlgebra](#).

INPUT:

- k – rank of the algebra
- q – the deformation parameter q

OPTIONAL ARGUMENTS:

- `base_ring` – (default `None`) a ring containing q ; if `None` then just takes the parent of q
- `prefix` – (default `"Pl"`) a label for the basis elements

EXAMPLES:

We define the planar algebra of rank 2 with parameter x over \mathbb{Z} :

```
sage: R.<x> = ZZ[]
sage: Pl = PlanarAlgebra(2, x, R); Pl
Planar Algebra of rank 2 with parameter x over Univariate Polynomial Ring in x_
↳over Integer Ring
sage: Pl.basis().list()
[Pl{{-2, -1, 1, 2}}, Pl{{-2, -1, 2}, {1}},
 Pl{{-2, -1, 1}, {2}}, Pl{{-2}, {-1, 1, 2}},
 Pl{{-2, 1, 2}, {-1}}, Pl{{-2, 2}, {-1, 1}},
 Pl{{-2, -1}, {1, 2}}, Pl{{-2, -1}, {1}, {2}},
```

```

Pl{{-2}, {-1, 2}, {1}}, Pl{{-2, 2}, {-1}, {1}},
Pl{{-2}, {-1, 1}, {2}}, Pl{{-2, 1}, {-1}, {2}},
Pl{{-2}, {-1}, {1, 2}}, Pl{{-2}, {-1}, {1}, {2}}]
sage: E = Pl([[1, 2], [-1, -2]])
sage: E^2 == x*E
True
sage: E^5 == x^4*E
True

```

class `sage.combinat.diagram_algebras.PlanarDiagrams (order)`
 Bases: `sage.combinat.diagram_algebras.AbstractPartitionDiagrams`

All planar diagrams of integer or integer +1/2 order.

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: pld = da.PlanarDiagrams(3)
sage: pld.an_element() in pld
True
sage: pld.cardinality() == len(pld.list())
True

```

cardinality ()

Return the cardinality of self .

The number of all planar diagrams of order k is the $2k$ -th Catalan number.

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: pld = da.PlanarDiagrams(3)
sage: pld.cardinality()
132

```

class `sage.combinat.diagram_algebras.PropagatingIdeal (k, q, base_ring, prefix)`

Bases: `sage.combinat.diagram_algebras.SubPartitionAlgebra`

A propagating ideal.

The propagating ideal of rank k is a non-unital algebra with basis indexed by the collection of ideal set partitions of $\{1, \dots, k, -1, \dots, -k\}$. We say a set partition is *ideal* if its propagating number is less than k .

This algebra is a non-unital subalgebra and an ideal of the partition algebra. For more information, see [PartitionAlgebra](#).

EXAMPLES:

We now define the propagating ideal of rank 2 with parameter x over \mathbb{Z} :

```

sage: R.<x> = QQ[]
sage: I = PropagatingIdeal(2, x, R); I
Propagating Ideal of rank 2 with parameter x
over Univariate Polynomial Ring in x over Rational Field
sage: I.basis().list()
[I{{-2, -1, 1, 2}}, I{{-2, -1, 2}, {1}},
I{{-2, -1, 1}, {2}}, I{{-2}, {-1, 1, 2}},
I{{-2, 1, 2}, {-1}}, I{{-2, -1}, {1, 2}},
I{{-2, -1}, {1}, {2}}, I{{-2}, {-1, 2}, {1}},
I{{-2, 2}, {-1}, {1}}, I{{-2}, {-1, 1}, {2}},
I{{-2, 1}, {-1}, {2}}, I{{-2}, {-1}, {1, 2}},

```

```

I{{-2}, {-1}, {1}, {2}}]
sage: E = I([[1,2],[-1,-2]])
sage: E^2 == x*E
True
sage: E^5 == x^4*E
True

```

class `Element (M, x)`

Bases: `sage.combinat.diagram_algebras.DiagramAlgebra.Element`

An element of a propagating ideal.

We need to take care of exponents since we are not unital.

`PropagatingIdeal.one_basis ()`

The propagating ideal is a non-unital algebra, i.e. it does not have a multiplicative identity.

EXAMPLES:

```

sage: R.<q> = QQ[]
sage: I = PropagatingIdeal(2, q, R)
sage: I.one_basis()
Traceback (most recent call last):
...
ValueError: The ideal partition algebra is not unital
sage: I.one()
Traceback (most recent call last):
...
ValueError: The ideal partition algebra is not unital

```

class `sage.combinat.diagram_algebras.SubPartitionAlgebra (k, q, base_ring, prefix, diagrams, category=None)`

Bases: `sage.combinat.diagram_algebras.DiagramAlgebra`

A subalgebra of the partition algebra indexed by a subset of the diagrams.

ambient ()

Return the partition algebra `self` is a sub-algebra of.

EXAMPLES:

```

sage: x = var('x')
sage: BA = BrauerAlgebra(2, x)
sage: BA.ambient()
Partition Algebra of rank 2 with parameter x over Symbolic Ring

```

lift ()

Return the lift map from diagram subalgebra to the ambient space.

EXAMPLES:

```

sage: R.<x> = QQ[]
sage: BA = BrauerAlgebra(2, x, R)
sage: E = BA([[1,2],[-1,-2]])
sage: lifted = BA.lift(E); lifted
B{{-2, -1}, {1, 2}}
sage: lifted.parent() is BA.ambient()
True

```

retract (x)

Retract an appropriate partition algebra element to the corresponding element in the partition subalgebra.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: BA = BrauerAlgebra(2, x, R)
sage: PA = BA.ambient()
sage: E = PA([[1,2], [-1,-2]])
sage: BA.retract(E) in BA
True
```

class sage.combinat.diagram_algebras. **TemperleyLiebAlgebra** (*k, q, base_ring, prefix*)
 Bases: [sage.combinat.diagram_algebras.SubPartitionAlgebra](#)

A Temperley–Lieb algebra.

The Temperley–Lieb algebra of rank k is an algebra with basis indexed by the collection of planar set partitions of $\{1, \dots, k, -1, \dots, -k\}$ with block size 2.

This algebra is thus a subalgebra of the partition algebra. For more information, see [PartitionAlgebra](#).

INPUT:

- k – rank of the algebra
- q – the deformation parameter q

OPTIONAL ARGUMENTS:

- *base_ring* – (default `None`) a ring containing q ; if `None` then just takes the parent of q
- *prefix* – (default `"T"`) a label for the basis elements

EXAMPLES:

We define the Temperley–Lieb algebra of rank 2 with parameter x over \mathbb{Z} :

```
sage: R.<x> = ZZ[]
sage: T = TemperleyLiebAlgebra(2, x, R); T
Temperley-Lieb Algebra of rank 2 with parameter x
over Univariate Polynomial Ring in x over Integer Ring
sage: T.basis()
Lazy family (Term map from Temperleylieb diagrams of order 2
to Temperley-Lieb Algebra of rank 2 with parameter x
over Univariate Polynomial Ring in x over
Integer Ring(i))_{i in Temperleylieb diagrams of order 2}
sage: b = T.basis().list()
sage: b
[T{{-2, 2}, {-1, 1}}, T{{-2, -1}, {1, 2}}]
sage: b[1]
T{{-2, -1}, {1, 2}}
sage: b[1]^2 == x*b[1]
True
sage: b[1]^5 == x^4*b[1]
True
```

class sage.combinat.diagram_algebras. **TemperleyLiebDiagrams** (*order*)
 Bases: [sage.combinat.diagram_algebras.AbstractPartitionDiagrams](#)

All Temperley–Lieb diagrams of integer or integer $+1/2$ order.

For more information on Temperley–Lieb diagrams, see [TemperleyLiebAlgebra](#).

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: td = da.TemperleyLiebDiagrams(3)
sage: td.an_element() in td
True
sage: td.cardinality() == len(td.list())
True

```

cardinality ()

Return the cardinality of `self`.

The number of Temperley–Lieb diagrams of integer order k is the k -th Catalan number.

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: td = da.TemperleyLiebDiagrams(3)
sage: td.cardinality()
5

```

`sage.combinat.diagram_algebras.brauer_diagrams (k)`

Return a generator of all Brauer diagrams of order k .

A Brauer diagram of order k is a partition diagram of order k with block size 2.

INPUT:

- k – the order of the Brauer diagrams

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: [SetPartition(p) for p in da.brauer_diagrams(2)]
[{{-2, 1}, {-1, 2}}, {{-2, 2}, {-1, 1}}, {{-2, -1}, {1, 2}}]
sage: [SetPartition(p) for p in da.brauer_diagrams(5/2)]
[{{-3, 3}, {-2, 1}, {-1, 2}}, {{-3, 3}, {-2, 2}, {-1, 1}}, {{-3, 3}, {-2, -1},
↪ {1, 2}}]

```

`sage.combinat.diagram_algebras.ideal_diagrams (k)`

Return a generator of all “ideal” diagrams of order k .

An ideal diagram of order k is a partition diagram of order k with propagating number less than k .

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: [SetPartition(p) for p in da.ideal_diagrams(2)]
[{{-2, -1, 1, 2}}, {{-2, -1, 2}, {1}}, {{-2, -1, 1}, {2}}, {{-2}, {-1, 1, 2}},
{{-2, 1, 2}, {-1}}, {{-2, -1}, {1, 2}}, {{-2, -1}, {1}, {2}},
{{-2}, {-1, 2}, {1}}, {{-2, 2}, {-1}, {1}}, {{-2}, {-1, 1}, {2}}, {{-2, 1},
{-1}, {2}}, {{-2}, {-1}, {1, 2}}, {{-2}, {-1}, {1}, {2}}]
sage: [SetPartition(p) for p in da.ideal_diagrams(3/2)]
[{{-2, -1, 1, 2}}, {{-2, -1, 2}, {1}}, {{-2, 1, 2}, {-1}}, {{-2, 2}, {-1}, {1}}]

```

`sage.combinat.diagram_algebras.identity_set_partition (k)`

Return the identity set partition $\{\{1, -1\}, \dots, \{k, -k\}\}$

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: SetPartition(da.identity_set_partition(2))
{{-2, 2}, {-1, 1}}

```

`sage.combinat.diagram_algebras.is_planar (sp)`

Return True if the diagram corresponding to the set partition `sp` is planar; otherwise, return False .

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: da.is_planar( da.to_set_partition([[1,-2],[2,-1]]))
False
sage: da.is_planar( da.to_set_partition([[1,-1],[2,-2]]))
True
```

`sage.combinat.diagram_algebras.pair_to_graph (sp1, sp2)`

Return a graph consisting of the disjoint union of the graphs of set partitions `sp1` and `sp2` along with edges joining the bottom row (negative numbers) of `sp1` to the top row (positive numbers) of `sp2` .

The vertices of the graph `sp1` appear in the result as pairs $(k, 1)$, whereas the vertices of the graph `sp2` appear as pairs $(k, 2)$.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: sp1 = da.to_set_partition([[1,-2],[2,-1]])
sage: sp2 = da.to_set_partition([[1,-2],[2,-1]])
sage: g = da.pair_to_graph( sp1, sp2 ); g
Graph on 8 vertices

sage: g.vertices()
[(-2, 1), (-2, 2), (-1, 1), (-1, 2), (1, 1), (1, 2), (2, 1), (2, 2)]
sage: g.edges()
[((-2, 1), (1, 1), None), ((-2, 1), (2, 2), None),
 ((-2, 2), (1, 2), None), ((-1, 1), (1, 2), None),
 ((-1, 1), (2, 1), None), ((-1, 2), (2, 2), None)]
```

Another example which used to be wrong until [trac ticket #15958](#):

```
sage: sp3 = da.to_set_partition([[1, -1], [2], [-2]])
sage: sp4 = da.to_set_partition([[1], [-1], [2], [-2]])
sage: g = da.pair_to_graph( sp3, sp4 ); g
Graph on 8 vertices

sage: g.vertices()
[(-2, 1), (-2, 2), (-1, 1), (-1, 2), (1, 1), (1, 2), (2, 1), (2, 2)]
sage: g.edges()
[((-2, 1), (2, 2), None), ((-1, 1), (1, 1), None),
 ((-1, 1), (1, 2), None)]
```

`sage.combinat.diagram_algebras.partition_diagrams (k)`

Return a generator of all partition diagrams of order `k` .

A partition diagram of order $k \in \mathbb{Z}$ is a set partition of $\{1, \dots, k, -1, \dots, -k\}$. If we have $k - 1/2 \in \mathbb{Z}$, then a partition diagram of order $k \in 1/2\mathbb{Z}$ is a set partition of $\{1, \dots, k + 1/2, -1, \dots, -(k + 1/2)\}$ with $k + 1/2$ and $-(k + 1/2)$ in the same block. See [\[HR2005\]](#).

INPUT:

- `k` – the order of the partition diagrams

EXAMPLES:


```

sage: import sage.combinat.diagram_algebras as da
sage: [SetPartition(p) for p in da.partition_diagrams(2)]
[{{-2, -1, 1, 2}}, {{-2, -1, 2}, {1}}, {{-2, -1, 1}, {2}},
 {{-2}, {-1, 1, 2}}, {{-2, 1, 2}, {-1}}, {{-2, 1}, {-1, 2}},
 {{-2, 2}, {-1, 1}}, {{-2, -1}, {1, 2}}, {{-2, -1}, {1}, {2}},
 {{-2}, {-1, 2}, {1}}, {{-2, 2}, {-1}, {1}}, {{-2}, {-1, 1}, {2}},
 {{-2, 1}, {-1}, {2}}, {{-2}, {-1}, {1, 2}}, {{-2}, {-1}, {1}, {2}}]
sage: [SetPartition(p) for p in da.partition_diagrams(3/2)]
[{{-2, -1, 1, 2}}, {{-2, -1, 2}, {1}}, {{-2, 2}, {-1, 1}},
 {{-2, 1, 2}, {-1}}, {{-2, 2}, {-1}, {1}}]

```

`sage.combinat.diagram_algebras.planar_diagrams (k)`

Return a generator of all planar diagrams of order k .

A planar diagram of order k is a partition diagram of order k that has no crossings.

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: [SetPartition(p) for p in da.planar_diagrams(2)]
[{{-2, -1, 1, 2}}, {{-2, -1, 2}, {1}}, {{-2, -1, 1}, {2}},
 {{-2}, {-1, 1, 2}}, {{-2, 1, 2}, {-1}}, {{-2, 2}, {-1, 1}},
 {{-2, -1}, {1, 2}}, {{-2, -1}, {1}, {2}}, {{-2}, {-1, 2}, {1}},
 {{-2, 2}, {-1}, {1}}, {{-2}, {-1, 1}, {2}}, {{-2, 1}, {-1}, {2}},
 {{-2}, {-1}, {1, 2}}, {{-2}, {-1}, {1}, {2}}]
sage: [SetPartition(p) for p in da.planar_diagrams(3/2)]
[{{-2, -1, 1, 2}}, {{-2, -1, 2}, {1}}, {{-2, 2}, {-1, 1}},
 {{-2, 1, 2}, {-1}}, {{-2, 2}, {-1}, {1}}]

```

`sage.combinat.diagram_algebras.propagating_number (sp)`

Return the propagating number of the set partition sp .

The propagating number is the number of blocks with both a positive and negative number.

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: sp1 = da.to_set_partition([[1,-2],[2,-1]])
sage: sp2 = da.to_set_partition([[1,2],[-2,-1]])
sage: da.propagating_number(sp1)
2
sage: da.propagating_number(sp2)
0

```

`sage.combinat.diagram_algebras.set_partition_composition (sp1, sp2)`

Return a tuple consisting of the composition of the set partitions $sp1$ and $sp2$ and the number of components removed from the middle rows of the graph.

EXAMPLES:

```

sage: import sage.combinat.diagram_algebras as da
sage: sp1 = da.to_set_partition([[1,-2],[2,-1]])
sage: sp2 = da.to_set_partition([[1,-2],[2,-1]])
sage: p, c = da.set_partition_composition(sp1, sp2)
sage: (SetPartition(p), c) == (SetPartition(da.identity_set_partition(2)), 0)
True

```

`sage.combinat.diagram_algebras.temperley_lieb_diagrams (k)`

Return a generator of all Temperley–Lieb diagrams of order k .

A Temperley–Lieb diagram of order k is a partition diagram of order k with block size 2 and is planar.

INPUT:

- k – the order of the Temperley–Lieb diagrams

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: [SetPartition(p) for p in da.temperley_lieb_diagrams(2)]
[{{-2, 2}, {-1, 1}}, {{-2, -1}, {1, 2}}]
sage: [SetPartition(p) for p in da.temperley_lieb_diagrams(5/2)]
[{{-3, 3}, {-2, 2}, {-1, 1}}, {{-3, 3}, {-2, -1}, {1, 2}}]
```

`sage.combinat.diagram_algebras.to_Brauer_partition (l, k=None)`

Same as `to_set_partition()` but assumes omitted elements are connected straight through.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: f = lambda sp: SetPartition(da.to_Brauer_partition(sp))
sage: f([[1, 2], [-1, -2]]) == SetPartition([[1, 2], [-1, -2]])
True
sage: f([[1, 3], [-1, -3]]) == SetPartition([[1, 3], [-3, -1], [2, -2]])
True
sage: f([[1, -4], [-3, -1], [3, 4]]) == SetPartition([[-3, -1], [2, -2], [1, -4], [3, 4]])
True
sage: p = SetPartition([[1, 2], [-1, -2], [3, -3], [4, -4]])
sage: SetPartition(da.to_Brauer_partition([[1, 2], [-1, -2]], k=4)) == p
True
```

`sage.combinat.diagram_algebras.to_graph (sp)`

Return a graph representing the set partition `sp`.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: g = da.to_graph( da.to_set_partition([[1, -2], [2, -1]])); g
Graph on 4 vertices

sage: g.vertices()
[-2, -1, 1, 2]
sage: g.edges()
[(-2, 1, None), (-1, 2, None)]
```

`sage.combinat.diagram_algebras.to_set_partition (l, k=None)`

Convert a list of a list of numbers to a set partitions. Each list of numbers in the outer list specifies the numbers contained in one of the blocks in the set partition.

If k is specified, then the set partition will be a set partition of $\{1, \dots, k, -1, \dots, -k\}$. Otherwise, k will default to the minimum number needed to contain all of the specified numbers.

EXAMPLES:

```
sage: import sage.combinat.diagram_algebras as da
sage: f = lambda sp: SetPartition(da.to_set_partition(sp))
sage: f([[1, -1], [2, -2]]) == SetPartition(da.identity_set_partition(2))
True
```

4.3 Clifford Algebras

AUTHORS:

- Travis Scrimshaw (2013-09-06): Initial version

class sage.algebras.clifford_algebra. **CliffordAlgebra** (*Q*, *names*, *category=None*)
 Bases: sage.combinat.free_module.CombinatorialFreeModule

The Clifford algebra of a quadratic form.

Let $Q : V \rightarrow \mathbf{k}$ denote a quadratic form on a vector space V over a field \mathbf{k} . The Clifford algebra $Cl(V, Q)$ is defined as $T(V)/I_Q$ where $T(V)$ is the tensor algebra of V and I_Q is the two-sided ideal generated by all elements of the form $v \otimes v - Q(v)$ for all $v \in V$.

We abuse notation to denote the projection of a pure tensor $x_1 \otimes x_2 \otimes \cdots \otimes x_m \in T(V)$ onto $T(V)/I_Q = Cl(V, Q)$ by $x_1 \wedge x_2 \wedge \cdots \wedge x_m$. This is motivated by the fact that $Cl(V, Q)$ is the exterior algebra $\wedge V$ when $Q = 0$ (one can also think of a Clifford algebra as a quantization of the exterior algebra). See [ExteriorAlgebra](#) for the concept of an exterior algebra.

From the definition, a basis of $Cl(V, Q)$ is given by monomials of the form

$$\{e_{i_1} \wedge \cdots \wedge e_{i_k} \mid 1 \leq i_1 < \cdots < i_k \leq n\},$$

where $n = \dim(V)$ and where $\{e_1, e_2, \dots, e_n\}$ is any fixed basis of V . Hence

$$\dim(Cl(V, Q)) = \sum_{k=0}^n \binom{n}{k} = 2^n.$$

Note: The algebra $Cl(V, Q)$ is a $\mathbf{Z}/2\mathbf{Z}$ -graded algebra, but not (in general) \mathbf{Z} -graded (in a reasonable way).

This construction satisfies the following universal property. Let $i : V \rightarrow Cl(V, Q)$ denote the natural inclusion (which is an embedding). Then for every associative \mathbf{k} -algebra A and any \mathbf{k} -linear map $j : V \rightarrow A$ satisfying

$$j(v)^2 = Q(v) \cdot 1_A$$

for all $v \in V$, there exists a unique \mathbf{k} -algebra homomorphism $f : Cl(V, Q) \rightarrow A$ such that $f \circ i = j$. This property determines the Clifford algebra uniquely up to canonical isomorphism. The inclusion i is commonly used to identify V with a vector subspace of $Cl(V)$.

The Clifford algebra $Cl(V, Q)$ is a \mathbf{Z}_2 -graded algebra (where $\mathbf{Z}_2 = \mathbf{Z}/2\mathbf{Z}$); this grading is determined by placing all elements of V in degree 1. It is also an \mathbf{N} -filtered algebra, with the filtration too being defined by placing all elements of V in degree 1. The `degree()` gives the \mathbf{N} -filtration degree, and to get the super degree use instead `is_even_odd()`.

The Clifford algebra also can be considered as a covariant functor from the category of vector spaces equipped with quadratic forms to the category of algebras. In fact, if (V, Q) and (W, R) are two vector spaces endowed with quadratic forms, and if $g : W \rightarrow V$ is a linear map preserving the quadratic form, then we can define an algebra morphism $Cl(g) : Cl(W, R) \rightarrow Cl(V, Q)$ by requiring that it send every $w \in W$ to $g(w) \in V$. Since the quadratic form R on W is uniquely determined by the quadratic form Q on V (due to the assumption that g preserves the quadratic form), this fact can be rewritten as follows: If (V, Q) is a vector space with a quadratic form, and W is another vector space, and $\phi : W \rightarrow V$ is any linear map, then we obtain an algebra morphism $Cl(\phi) : Cl(W, \phi(Q)) \rightarrow Cl(V, Q)$ where $\phi(Q) = \phi^T \cdot Q \cdot \phi$ (we consider ϕ as a matrix) is the quadratic form Q pulled back to W . In fact, the map ϕ preserves the quadratic form because of

$$\phi(Q)(x) = x^T \cdot \phi^T \cdot Q \cdot \phi \cdot x = (\phi \cdot x)^T \cdot Q \cdot (\phi \cdot x) = Q(\phi(x)).$$

Hence we have $\phi(w)^2 = Q(\phi(w)) = \phi(Q)(w)$ for all $w \in W$.

REFERENCES:

- [Wikipedia article Clifford algebra](#)

INPUT:

- Q – a quadratic form
- `names` – (default: `'e'`) the generator names

EXAMPLES:

To create a Clifford algebra, all one needs to do is specify a quadratic form:

```
sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl = CliffordAlgebra(Q)
sage: Cl
The Clifford algebra of the Quadratic form in 3 variables
over Integer Ring with coefficients:
[ 1 2 3 ]
[ * 4 5 ]
[ * * 6 ]
```

We can also explicitly name the generators. In this example, the Clifford algebra we construct is an exterior algebra (since we choose the quadratic form to be zero):

```
sage: Q = QuadraticForm(ZZ, 4, [0]*10)
sage: Cl.<a,b,c,d> = CliffordAlgebra(Q)
sage: a*d
a*d
sage: d*c*b*a + a + 4*b*c
a*b*c*d + 4*b*c + a
```

Element

alias of *CliffordAlgebraElement*

algebra_generators ()

Return the algebra generators of `self`.

EXAMPLES:

```
sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.algebra_generators()
Finite family {'y': y, 'x': x, 'z': z}
```

center_basis ()

Return a list of elements which correspond to a basis for the center of `self`.

This assumes that the ground ring can be used to compute the kernel of a matrix.

See also:

supercenter_basis() , <http://math.stackexchange.com/questions/129183/center-of-clifford-algebra-depending-on-the-parity-of-dim-v>

Todo

Deprecate this in favor of a method called *center()* once subalgebras are properly implemented in Sage.

EXAMPLES:

```

sage: Q = QuadraticForm(QQ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Z = Cl.center_basis(); Z
(1, -2/5*x*y*z + x - 3/5*y + 2/5*z)
sage: all(z*b - b*z == 0 for z in Z for b in Cl.basis())
True

sage: Q = QuadraticForm(QQ, 3, [1,-2,-3, 4, 2, 1])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Z = Cl.center_basis(); Z
(1, -x*y*z + x + 3/2*y - z)
sage: all(z*b - b*z == 0 for z in Z for b in Cl.basis())
True

sage: Q = QuadraticForm(QQ, 2, [1,-2,-3])
sage: Cl.<x,y> = CliffordAlgebra(Q)
sage: Cl.center_basis()
(1,)

sage: Q = QuadraticForm(QQ, 2, [-1,1,-3])
sage: Cl.<x,y> = CliffordAlgebra(Q)
sage: Cl.center_basis()
(1,)

```

A degenerate case:

```

sage: Q = QuadraticForm(QQ, 3, [4,4,-4,1,-2,1])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.center_basis()
(1, x*y*z + x - 2*y - 2*z, x*y + x*z - 2*y*z)

```

The most degenerate case (the exterior algebra):

```

sage: Q = QuadraticForm(QQ, 3)
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.center_basis()
(1, x*y, x*z, y*z, x*y*z)

```

degree_on_basis (*m*)

Return the degree of the monomial indexed by *m*.

We are considering the Clifford algebra to be \mathbf{N} -filtered, and the degree of the monomial *m* is the length of *m*.

EXAMPLES:

```

sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.degree_on_basis((0,))
1
sage: Cl.degree_on_basis((0,1))
2

```

dimension ()

Return the rank of *self* as a free module.

Let *V* be a free *R*-module of rank *n*; then, $Cl(V, Q)$ is a free *R*-module of rank 2^n .

EXAMPLES:

```
sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.dimension()
8
```

free_module ()

Return the underlying free module V of `self`.

This is the free module on which the quadratic form that was used to construct `self` is defined.

EXAMPLES:

```
sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.free_module()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

gen (i)

Return the i -th standard generator of the algebra `self`.

This is the i -th basis vector of the vector space on which the quadratic form defining `self` is defined, regarded as an element of `self`.

EXAMPLES:

```
sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: [Cl.gen(i) for i in range(3)]
[x, y, z]
```

gens ()

Return the generators of `self` (as an algebra).

EXAMPLES:

```
sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.gens()
(x, y, z)
```

graded_algebra ()

Return the associated graded algebra of `self`.

EXAMPLES:

```
sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.graded_algebra()
The exterior algebra of rank 3 over Integer Ring
```

is_commutative ()

Check if `self` is a commutative algebra.

EXAMPLES:

```
sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.is_commutative()
False
```

lift_isometry (*m*, *names=None*)

Lift an invertible isometry *m* of the quadratic form of *self* to a Clifford algebra morphism.

Given an invertible linear map $m : V \rightarrow W$ (here represented by a matrix acting on column vectors), this method returns the algebra morphism $Cl(m)$ from $Cl(V, Q)$ to $Cl(W, m^{-1}(Q))$, where $Cl(V, Q)$ is the Clifford algebra *self* and where $m^{-1}(Q)$ is the pullback of the quadratic form Q to W along the inverse map $m^{-1} : W \rightarrow V$. See the documentation of [CliffordAlgebra](#) for how this pullback and the morphism $Cl(m)$ are defined.

INPUT:

- *m* – an isometry of the quadratic form of *self*
- *names* – (default: 'e') the names of the generators of the Clifford algebra of the codomain of (the map represented by) *m*

OUTPUT:

The algebra morphism $Cl(m)$ from *self* to $Cl(W, m^{-1}(Q))$.

EXAMPLES:

```
sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: m = matrix([[1,1,2],[0,1,1],[0,0,1]])
sage: phi = Cl.lift_isometry(m, 'abc')
sage: phi(x)
a
sage: phi(y)
a + b
sage: phi(x*y)
a*b + 1
sage: phi(x) * phi(y)
a*b + 1
sage: phi(z*y)
a*b - a*c - b*c
sage: phi(z) * phi(y)
a*b - a*c - b*c
sage: phi(x + z) * phi(y + z) == phi((x + z) * (y + z))
True
```

lift_module_morphism (*m*, *names=None*)

Lift the matrix *m* to an algebra morphism of Clifford algebras.

Given a linear map $m : W \rightarrow V$ (here represented by a matrix acting on column vectors), this method returns the algebra morphism $Cl(m) : Cl(W, m(Q)) \rightarrow Cl(V, Q)$, where $Cl(V, Q)$ is the Clifford algebra *self* and where $m(Q)$ is the pullback of the quadratic form Q to W . See the documentation of [CliffordAlgebra](#) for how this pullback and the morphism $Cl(m)$ are defined.

Note: This is a map into *self*.

INPUT:

- *m* – a matrix
- *names* – (default: 'e') the names of the generators of the Clifford algebra of the domain of (the map represented by) *m*

OUTPUT:

The algebra morphism $Cl(m)$ from $Cl(W, m(Q))$ to self .

EXAMPLES:

```
sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: m = matrix([[1,-1,-1],[0,1,-1],[1,1,1]])
sage: phi = Cl.lift_module_morphism(m, 'abc')
sage: phi
Generic morphism:
  From: The Clifford algebra of the Quadratic form in 3 variables over_
  ↪ Integer Ring with coefficients:
[ 10 17 3 ]
[ * 11 0 ]
[ * * 5 ]
  To:   The Clifford algebra of the Quadratic form in 3 variables over_
  ↪ Integer Ring with coefficients:
[ 1 2 3 ]
[ * 4 5 ]
[ * * 6 ]
sage: a,b,c = phi.domain().gens()
sage: phi(a)
x + z
sage: phi(b)
-x + y + z
sage: phi(c)
-x - y + z
sage: phi(a + 3*b)
-2*x + 3*y + 4*z
sage: phi(a) + 3*phi(b)
-2*x + 3*y + 4*z
sage: phi(a*b)
x*y + 2*x*z - y*z + 7
sage: phi(b*a)
-x*y - 2*x*z + y*z + 10
sage: phi(a*b + c)
x*y + 2*x*z - y*z - x - y + z + 7
sage: phi(a*b) + phi(c)
x*y + 2*x*z - y*z - x - y + z + 7
```

We check that the map is an algebra morphism:

```
sage: phi(a)*phi(b)
x*y + 2*x*z - y*z + 7
sage: phi(a*b)
x*y + 2*x*z - y*z + 7
sage: phi(a*a)
10
sage: phi(a)*phi(a)
10
sage: phi(b*a)
-x*y - 2*x*z + y*z + 10
sage: phi(b) * phi(a)
-x*y - 2*x*z + y*z + 10
sage: phi((a + b)*(a + c)) == phi(a + b) * phi(a + c)
True
```


We can also lift arbitrary linear maps:

```
sage: m = matrix([[1,1],[0,1],[1,1]])
sage: phi = Cl.lift_module_morphism(m, 'ab')
sage: a,b = phi.domain().gens()
sage: phi(a)
x + z
sage: phi(b)
x + y + z
sage: phi(a*b)
x*y - y*z + 15
sage: phi(a)*phi(b)
x*y - y*z + 15
sage: phi(b*a)
-x*y + y*z + 12
sage: phi(b)*phi(a)
-x*y + y*z + 12

sage: m = matrix([[1,1,1,2],[0,1,1,1],[0,1,1,1]])
sage: phi = Cl.lift_module_morphism(m, 'abcd')
sage: a,b,c,d = phi.domain().gens()
sage: phi(a)
x
sage: phi(b)
x + y + z
sage: phi(c)
x + y + z
sage: phi(d)
2*x + y + z
sage: phi(a*b*c + d*a)
-x*y - x*z + 21*x + 7
sage: phi(a*b*c*d)
21*x*y + 21*x*z + 42
```

ngens ()

Return the number of algebra generators of `self`.

EXAMPLES:

```
sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.ngens()
3
```

one_basis ()

Return the basis index of the element 1.

EXAMPLES:

```
sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.one_basis()
()
```

pseudoscalar ()

Return the unit pseudoscalar of `self`.

Given the basis e_1, e_2, \dots, e_n of the underlying R -module, the unit pseudoscalar is defined as $e_1 \cdot e_2 \cdots e_n$.

This depends on the choice of basis.

EXAMPLES:

```
sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.pseudoscalar()
x*y*z

sage: Q = QuadraticForm(ZZ, 0, [])
sage: Cl = CliffordAlgebra(Q)
sage: Cl.pseudoscalar()
1
```

REFERENCES:

- Wikipedia article [Classification_of_Clifford_algebras#Unit_pseudoscalar](#)

quadratic_form ()

Return the quadratic form of `self`.

This is the quadratic form used to define `self`. The quadratic form on `self` is yet to be implemented.

EXAMPLES:

```
sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.quadratic_form()
Quadratic form in 3 variables over Integer Ring with coefficients:
[ 1 2 3 ]
[ * 4 5 ]
[ * * 6 ]
```

supercenter_basis ()

Return a list of elements which correspond to a basis for the supercenter of `self`.

This assumes that the ground ring can be used to compute the kernel of a matrix.

See also:

`center_basis()`, <http://math.stackexchange.com/questions/129183/center-of-clifford-algebra-depending-on-the-parity-of-dim-v>

Todo

Deprecate this in favor of a method called `supercenter()` once subalgebras are properly implemented in Sage.

EXAMPLES:

```
sage: Q = QuadraticForm(QQ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: SZ = Cl.supercenter_basis(); SZ
(1,)
sage: all(z.supercommutator(b) == 0 for z in SZ for b in Cl.basis())
True

sage: Q = QuadraticForm(QQ, 3, [1,-2,-3, 4, 2, 1])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.supercenter_basis()
(1,)
```

```

sage: Q = QuadraticForm(QQ, 2, [1,-2,-3])
sage: Cl.<x,y> = CliffordAlgebra(Q)
sage: Cl.supercenter_basis()
(1,)

sage: Q = QuadraticForm(QQ, 2, [-1,1,-3])
sage: Cl.<x,y> = CliffordAlgebra(Q)
sage: Cl.supercenter_basis()
(1,)

```

Singular vectors of a quadratic form generate in the supercenter:

```

sage: Q = QuadraticForm(QQ, 3, [1/2,-2,4,256/249,3,-185/8])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.supercenter_basis()
(1, x + 249/322*y + 22/161*z)

sage: Q = QuadraticForm(QQ, 3, [4,4,-4,1,-2,1])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.supercenter_basis()
(1, x + 2*z, y + z, x*y + x*z - 2*y*z)

```

The most degenerate case:

```

sage: Q = QuadraticForm(QQ, 3)
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: Cl.supercenter_basis()
(1, x, y, z, x*y, x*z, y*z, x*y*z)

```

class sage.algebras.clifford_algebra. **CliffordAlgebraElement** (*M,x*)
 Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

An element in a Clifford algebra.

clifford_conjugate ()

Return the Clifford conjugate of *self* .

The Clifford conjugate of an element *x* of a Clifford algebra is defined as

$$\bar{x} := \alpha(x^t) = \alpha(x)^t$$

where α denotes the *reflection* automorphism and *t* the *transposition* .

EXAMPLES:

```

sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: elt = 5*x + y + x*z
sage: c = elt.conjugate(); c
-x*z - 5*x - y + 3
sage: c.conjugate() == elt
True

```

conjugate ()

Return the Clifford conjugate of *self* .

The Clifford conjugate of an element *x* of a Clifford algebra is defined as

$$\bar{x} := \alpha(x^t) = \alpha(x)^t$$

where α denotes the *reflection* automorphism and t the *transposition*.

EXAMPLES:

```
sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: elt = 5*x + y + x*z
sage: c = elt.conjugate(); c
-x*z - 5*x - y + 3
sage: c.conjugate() == elt
True
```

degree_negation ()

Return the image of the reflection automorphism on *self*.

The *reflection automorphism* of a Clifford algebra is defined as the linear endomorphism of this algebra which maps

$$x_1 \wedge x_2 \wedge \cdots \wedge x_m \mapsto (-1)^m x_1 \wedge x_2 \wedge \cdots \wedge x_m.$$

It is an algebra automorphism of the Clifford algebra.

degree_negation() is an alias for *reflection()*.

EXAMPLES:

```
sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: elt = 5*x + y + x*z
sage: r = elt.reflection(); r
x*z - 5*x - y
sage: r.reflection() == elt
True
```

list ()

Return the list of monomials and their coefficients in *self* (as a list of 2-tuples, each of which has the form (monomial, coefficient)).

EXAMPLES:

```
sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: elt = 5*x + y
sage: elt.list()
[( (0, ), 5), ( (1, ), 1)]
```

reflection ()

Return the image of the reflection automorphism on *self*.

The *reflection automorphism* of a Clifford algebra is defined as the linear endomorphism of this algebra which maps

$$x_1 \wedge x_2 \wedge \cdots \wedge x_m \mapsto (-1)^m x_1 \wedge x_2 \wedge \cdots \wedge x_m.$$

It is an algebra automorphism of the Clifford algebra.

degree_negation() is an alias for *reflection()*.

EXAMPLES:

```

sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: elt = 5*x + y + x*z
sage: r = elt.reflection(); r
x*z - 5*x - y
sage: r.reflection() == elt
True

```

supercommutator (x)

Return the supercommutator of `self` and `x`.

Let A be a superalgebra. The *supercommutator* of homogeneous elements $x, y \in A$ is defined by

$$[x, y] = xy - (-1)^{|x||y|}yx$$

and extended to all elements by linearity.

EXAMPLES:

```

sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: a = x*y - z
sage: b = x - y + y*z
sage: a.supercommutator(b)
-5*x*y + 8*x*z - 2*y*z - 6*x + 12*y - 5*z
sage: a.supercommutator(Cl.one())
0
sage: Cl.one().supercommutator(a)
0
sage: Cl.zero().supercommutator(a)
0
sage: a.supercommutator(Cl.zero())
0

sage: Q = QuadraticForm(ZZ, 2, [-1,1,-3])
sage: Cl.<x,y> = CliffordAlgebra(Q)
sage: [a.supercommutator(b) for a in Cl.basis() for b in Cl.basis()]
[0, 0, 0, 0, 0, -2, 1, -x - 2*y, 0, 1,
 -6, 6*x + y, 0, x + 2*y, -6*x - y, 0]
sage: [a*b-b*a for a in Cl.basis() for b in Cl.basis()]
[0, 0, 0, 0, 0, 0, 2*x*y - 1, -x - 2*y, 0,
 -2*x*y + 1, 0, 6*x + y, 0, x + 2*y, -6*x - y, 0]

```

Exterior algebras inherit from Clifford algebras, so supercommutators work as well. We verify the exterior algebra is supercommutative:

```

sage: E.<x,y,z,w> = ExteriorAlgebra(QQ)
sage: all(b1.supercommutator(b2) == 0
....:      for b1 in E.basis() for b2 in E.basis())
True

```

support ()

Return the support of `self`.

This is the list of all monomials which appear with nonzero coefficient in `self`.

EXAMPLES:

```

sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: elt = 5*x + y
sage: elt.support()
[(0,), (1,)]

```

transpose ()

Return the transpose of self .

The transpose is an anti-algebra involution of a Clifford algebra and is defined (using linearity) by

$$x_1 \wedge x_2 \wedge \cdots \wedge x_m \mapsto x_m \wedge \cdots \wedge x_2 \wedge x_1.$$

EXAMPLES:

```

sage: Q = QuadraticForm(ZZ, 3, [1,2,3,4,5,6])
sage: Cl.<x,y,z> = CliffordAlgebra(Q)
sage: elt = 5*x + y + x*z
sage: t = elt.transpose(); t
-x*z + 5*x + y + 3
sage: t.transpose() == elt
True
sage: Cl.one().transpose()
1

```

class sage.algebras.clifford_algebra. **ExteriorAlgebra** (*R*, *names*)

Bases: [sage.algebras.clifford_algebra.CliffordAlgebra](#)

An exterior algebra of a free module over a commutative ring.

Let V be a module over a commutative ring R . The exterior algebra (or Grassmann algebra) $\Lambda(V)$ of V is defined as the quotient of the tensor algebra $T(V)$ of V modulo the two-sided ideal generated by all tensors of the form $x \otimes x$ with $x \in V$. The multiplication on $\Lambda(V)$ is denoted by \wedge (so $v_1 \wedge v_2 \wedge \cdots \wedge v_n$ is the projection of $v_1 \otimes v_2 \otimes \cdots \otimes v_n$ onto $\Lambda(V)$) and called the “exterior product” or “wedge product”.

If V is a rank- n free R -module with a basis $\{e_1, \dots, e_n\}$, then $\Lambda(V)$ is the R -algebra noncommutatively generated by the n generators e_1, \dots, e_n subject to the relations $e_i^2 = 0$ for all i , and $e_i e_j = -e_j e_i$ for all $i < j$. As an R -module, $\Lambda(V)$ then has a basis $(\bigwedge_{i \in I} e_i)$ with I ranging over the subsets of $\{1, 2, \dots, n\}$ (where $\bigwedge_{i \in I} e_i$ is the wedge product of e_i for i running through all elements of I from smallest to largest), and hence is free of rank 2^n .

The exterior algebra of an R -module V can also be realized as the Clifford algebra of V for the quadratic form Q given by $Q(v) = 0$ for all vectors $v \in V$. See [CliffordAlgebra](#) for the notion of a Clifford algebra.

The exterior algebra of an R -module V is a connected \mathbf{Z} -graded Hopf superalgebra. It is commutative in the super sense (i.e., the odd elements anticommute and square to 0).

This class implements the exterior algebra $\Lambda(R^n)$ for n a nonnegative integer.

Warning: We initialize the exterior algebra as an object of the category of Hopf algebras, but this is not really correct, since it is a Hopf superalgebra with the odd-degree components forming the odd part. So use Hopf-algebraic methods with care!

INPUT:

- R – the base ring, or the free module whose exterior algebra is to be computed

- `names` – a list of strings to name the generators of the exterior algebra; this list can either have one entry only (in which case the generators will be called `e + '0'`, `e + '1'`, ..., `e + 'n-1'`, with `e` being said entry), or have `n` entries (in which case these entries will be used directly as names for the generators)
- `n` – the number of generators, i.e., the rank of the free module whose exterior algebra is to be computed (this doesn't have to be provided if it can be inferred from the rest of the input)

REFERENCES:

- [Wikipedia article Exterior_algebra](#)

class `Element` (`M`, `x`)

Bases: `sage.algebras.clifford_algebra.CliffordAlgebraElement`

An element of an exterior algebra.

antiderivation (`x`)

Return the interior product (also known as antiderivation) of `self` with respect to `x` (that is, the element $\iota_x(\text{self})$ of the exterior algebra).

If V is an R -module, and if α is a fixed element of V^* , then the *interior product* with respect to α is an R -linear map $i_\alpha: \Lambda(V) \rightarrow \Lambda(V)$, determined by the following requirements:

- $i_\alpha(v) = \alpha(v)$ for all $v \in V = \Lambda^1(V)$,
- it is a graded derivation of degree -1 : all x and y in $\Lambda(V)$ satisfy

$$i_\alpha(x \wedge y) = (i_\alpha x) \wedge y + (-1)^{\deg x} x \wedge (i_\alpha y).$$

It can be shown that this map i_α is graded of degree -1 (that is, sends $\Lambda^k(V)$ into $\Lambda^{k-1}(V)$ for every k).

When V is a finite free R -module, the interior product can also be defined by

$$(i_\alpha \omega)(u_1, \dots, u_k) = \omega(\alpha, u_1, \dots, u_k),$$

where $\omega \in \Lambda^k(V)$ is thought of as an alternating multilinear mapping from $V^* \times \dots \times V^*$ to R .

Since Sage is only dealing with exterior powers of modules of the form R^d for some nonnegative integer d , the element $\alpha \in V^*$ can be thought of as an element of V (by identifying the standard basis of $V = R^d$ with its dual basis). This is how α should be passed to this method.

We then extend the interior product to all $\alpha \in \Lambda(V^*)$ by

$$i_{\beta \wedge \gamma} = i_\gamma \circ i_\beta.$$

INPUT:

- `x` – element of (or coercing into) $\Lambda^1(V)$ (for example, an element of V); this plays the role of α in the above definition

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: x.interior_product(x)
1
sage: (x + x*y).interior_product(2*y)
-2*x
sage: (x*z + x*y*z).interior_product(2*y - x)
-2*x^2z - y^2z - z
sage: x.interior_product(E.one())
x
sage: E.one().interior_product(x)
0
sage: x.interior_product(E.zero())
0
```

```
sage: E.zero().interior_product(x)
0
```

REFERENCES:

- [Wikipedia article Exterior_algebra#Interior_product](#)

constant_coefficient ()

Return the constant coefficient of `self`.

Todo

Define a similar method for general Clifford algebras once the morphism to exterior algebras is implemented.

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: elt = 5*x + y + x*z + 10
sage: elt.constant_coefficient()
10
sage: x.constant_coefficient()
0
```

hodge_dual ()

Return the Hodge dual of `self`.

The Hodge dual of an element α of the exterior algebra is defined as $i_\alpha \sigma$, where σ is the volume form (`volume_form()`) and i_α denotes the antiderivation function with respect to α (see `interior_product()` for the definition of this).

Note: The Hodge dual of the Hodge dual of a homogeneous element p of $\Lambda(V)$ equals $(-1)^{k(n-k)}p$, where $n = \dim V$ and $k = \deg(p) = |p|$.

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: x.hodge_dual()
y^z
sage: (x*z).hodge_dual()
-y
sage: (x*y*z).hodge_dual()
1
sage: [a.hodge_dual().hodge_dual() for a in E.basis()]
[1, x, y, z, x^y, x^z, y^z, x^y^z]
sage: (x + x*y).hodge_dual()
y^z + z
sage: (x*z + x*y*z).hodge_dual()
-y + 1
sage: E = ExteriorAlgebra(QQ, 'wxyz')
sage: [a.hodge_dual().hodge_dual() for a in E.basis()]
[1, -w, -x, -y, -z, w^x, w^y, w^z, x^y, x^z, y^z,
-w^x^y, -w^x^z, -w^y^z, -x^y^z, w^x^y^z]
```

interior_product (x)

Return the interior product (also known as antiderivation) of `self` with respect to x (that is, the

element $\iota_x(\text{self})$ of the exterior algebra).

If V is an R -module, and if α is a fixed element of V^* , then the *interior product* with respect to α is an R -linear map $i_\alpha: \Lambda(V) \rightarrow \Lambda(V)$, determined by the following requirements:

- $i_\alpha(v) = \alpha(v)$ for all $v \in V = \Lambda^1(V)$,
- it is a graded derivation of degree -1 : all x and y in $\Lambda(V)$ satisfy
$$i_\alpha(x \wedge y) = (i_\alpha x) \wedge y + (-1)^{\deg x} x \wedge (i_\alpha y).$$

It can be shown that this map i_α is graded of degree -1 (that is, sends $\Lambda^k(V)$ into $\Lambda^{k-1}(V)$ for every k).

When V is a finite free R -module, the interior product can also be defined by

$$(i_\alpha \omega)(u_1, \dots, u_k) = \omega(\alpha, u_1, \dots, u_k),$$

where $\omega \in \Lambda^k(V)$ is thought of as an alternating multilinear mapping from $V^* \times \dots \times V^*$ to R .

Since Sage is only dealing with exterior powers of modules of the form R^d for some nonnegative integer d , the element $\alpha \in V^*$ can be thought of as an element of V (by identifying the standard basis of $V = R^d$ with its dual basis). This is how α should be passed to this method.

We then extend the interior product to all $\alpha \in \Lambda(V^*)$ by

$$i_{\beta \wedge \gamma} = i_\gamma \circ i_\beta.$$

INPUT:

- **x** – element of (or coercing into) $\Lambda^1(V)$ (for example, an element of V); this plays the role of α in the above definition

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: x.interior_product(x)
1
sage: (x + x*y).interior_product(2*y)
-2*x
sage: (x*z + x*y*z).interior_product(2*y - x)
-2*x^z - y^z - z
sage: x.interior_product(E.one())
x
sage: E.one().interior_product(x)
0
sage: x.interior_product(E.zero())
0
sage: E.zero().interior_product(x)
0
```

REFERENCES:

- [Wikipedia article Exterior algebra#Interior product](#)

scalar (*other*)

Return the standard scalar product of **self** with **other**.

The standard scalar product of $x, y \in \Lambda(V)$ is defined by $\langle x, y \rangle = \langle x^t y \rangle$, where $\langle a \rangle$ denotes the degree-0 term of a , and where x^t denotes the transpose (`transpose()`) of x .

Todo

Define a similar method for general Clifford algebras once the morphism to exterior algebras is implemented.

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: elt = 5*x + y + x*z
sage: elt.scalar(z + 2*x)
0
sage: elt.transpose() * (z + 2*x)
-2*x^y + 5*x^z + y^z
```

`ExteriorAlgebra. antipode_on_basis (m)`

Return the antipode on the basis element indexed by `m`.

Given a basis element ω , the antipode is defined by $S(\omega) = (-1)^{\deg(\omega)}\omega$.

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: E.antipode_on_basis(())
1
sage: E.antipode_on_basis((1,))
-y
sage: E.antipode_on_basis((1,2))
y^z
```

`ExteriorAlgebra. boundary (s_coeff)`

Return the boundary operator ∂ defined by the structure coefficients `s_coeff` of a Lie algebra.

For more on the boundary operator, see [ExteriorAlgebraBoundary](#).

INPUT:

- `s_coeff` – a dictionary whose keys are in $I \times I$, where I is the index set of the underlying vector space V , and whose values can be coerced into 1-forms (degree 1 elements) in E (usually, these values will just be elements of V)

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: E.boundary({(0,1): z, (1,2): x, (2,0): y})
Boundary endomorphism of The exterior algebra of rank 3 over Rational Field
```

`ExteriorAlgebra. coboundary (s_coeff)`

Return the coboundary operator d defined by the structure coefficients `s_coeff` of a Lie algebra.

For more on the coboundary operator, see [ExteriorAlgebraCoboundary](#).

INPUT:

- `s_coeff` – a dictionary whose keys are in $I \times I$, where I is the index set of the underlying vector space V , and whose values can be coerced into 1-forms (degree 1 elements) in E (usually, these values will just be elements of V)

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: E.coboundary({(0,1): z, (1,2): x, (2,0): y})
Coboundary endomorphism of The exterior algebra of rank 3 over Rational Field
```

`ExteriorAlgebra. coproduct_on_basis (a)`

Return the coproduct on the basis element indexed by `a`.

The coproduct is defined by

$$\Delta(e_{i_1} \wedge \cdots \wedge e_{i_m}) = \sum_{k=0}^m \sum_{\sigma \in Ush_{k,m-k}} (-1)^\sigma (e_{i_{\sigma(1)}} \wedge \cdots \wedge e_{i_{\sigma(k)}}) \otimes (e_{i_{\sigma(k+1)}} \wedge \cdots \wedge e_{i_{\sigma(m)}}),$$

where $Ush_{k,m-k}$ denotes the set of all $(k, m-k)$ -unshuffles (i.e., permutations in S_m which are increasing on the interval $\{1, 2, \dots, k\}$ and on the interval $\{k+1, k+2, \dots, k+m\}$).

Warning: This coproduct is a homomorphism of superalgebras, not a homomorphism of algebras!

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: E.coproduct_on_basis((0,))
1 # x + x # 1
sage: E.coproduct_on_basis((0,1))
1 # x^y + x # y + x^y # 1 - y # x
sage: E.coproduct_on_basis((0,1,2))
1 # x^y^z + x # y^z + x^y # z + x^y^z # 1
- x^z # y - y # x^z + y^z # x + z # x^y
```

`ExteriorAlgebra.counit (x)`

Return the counit of x .

The counit of an element ω of the exterior algebra is its constant coefficient.

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: elt = x*y - 2*x + 3
sage: E.counit(elt)
3
```

`ExteriorAlgebra.degree_on_basis (m)`

Return the degree of the monomial indexed by m .

The degree of m in the \mathbb{Z} -grading of `self` is defined to be the length of m .

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: E.degree_on_basis(())
0
sage: E.degree_on_basis((0,))
1
sage: E.degree_on_basis((0,1))
2
```

`ExteriorAlgebra.interior_product_on_basis (a, b)`

Return the interior product $\iota_b a$ of a with respect to b .

See `interior_product()` for more information.

In this method, a and b are supposed to be basis elements (see `interior_product()` for a method that computes interior product of arbitrary elements), and to be input as their keys.

This depends on the choice of basis of the vector space whose exterior algebra is `self`.

EXAMPLES:

```

sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: E.interior_product_on_basis((0,), (0,))
1
sage: E.interior_product_on_basis((0,2), (0,))
z
sage: E.interior_product_on_basis((1,), (0,2))
0
sage: E.interior_product_on_basis((0,2), (1,))
0
sage: E.interior_product_on_basis((0,1,2), (0,2))
-y

```

`ExteriorAlgebra.lift_morphism (phi, names=None)`

Lift the matrix `m` to an algebra morphism of exterior algebras.

Given a linear map $\phi : V \rightarrow W$ (here represented by a matrix acting on column vectors over the base ring of V), this method returns the algebra morphism $\Lambda(\phi) : \Lambda(V) \rightarrow \Lambda(W)$. This morphism is defined on generators $v_i \in \Lambda(V)$ by $v_i \mapsto \phi(v_i)$.

Note: This is the map going out of `self` as opposed to `lift_module_morphism()` for general Clifford algebras.

INPUT:

- `phi` – a linear map ϕ from V to W , encoded as a matrix
- `names` – (default: `'e'`) the names of the generators of the Clifford algebra of the domain of (the map represented by) `phi`

OUTPUT:

The algebra morphism $\Lambda(\phi)$ from `self` to $\Lambda(W)$.

EXAMPLES:

```

sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: phi = matrix([[0,1],[1,1],[1,2]]); phi
[0 1]
[1 1]
[1 2]
sage: L = E.lift_morphism(phi, ['a','b','c']); L
Generic morphism:
  From: The exterior algebra of rank 2 over Rational Field
  To:   The exterior algebra of rank 3 over Rational Field
sage: L(x)
b + c
sage: L(y)
a + b + 2*c
sage: L.on_basis()(1,)
a + b + 2*c
sage: p = L(E.one()); p
1
sage: p.parent()
The exterior algebra of rank 3 over Rational Field
sage: L(x*y)
-a^b - a^c + b^c
sage: L(x)*L(y)
-a^b - a^c + b^c

```

```

sage: L(x + y)
a + 2*b + 3*c
sage: L(x) + L(y)
a + 2*b + 3*c
sage: L(1/2*x + 2)
1/2*b + 1/2*c + 2
sage: L(E(3))
3

sage: psi = matrix([[1, -3/2]]); psi
[ 1 -3/2]
sage: Lp = E.lift_morphism(psi, ['a']); Lp
Generic morphism:
  From: The exterior algebra of rank 2 over Rational Field
  To:   The exterior algebra of rank 1 over Rational Field
sage: Lp(x)
a
sage: Lp(y)
-3/2*a
sage: Lp(x + 2*y + 3)
-2*a + 3

```

ExteriorAlgebra. **lifted_bilinear_form** (*M*)

Return the bilinear form on the exterior algebra `self = $\Lambda(V)$` which is obtained by lifting the bilinear form f on V given by the matrix M .

Let V be a module over a commutative ring R , and let $f : V \times V \rightarrow R$ be a bilinear form on V . Then, a bilinear form $\Lambda(f) : \Lambda(V) \times \Lambda(V) \rightarrow R$ on $\Lambda(V)$ can be canonically defined as follows: For every $n \in \mathbb{N}$, $m \in \mathbb{N}$, $v_1, v_2, \dots, v_n, w_1, w_2, \dots, w_m \in V$, we define

$$\Lambda(f)(v_1 \wedge v_2 \wedge \dots \wedge v_n, w_1 \wedge w_2 \wedge \dots \wedge w_m) := \begin{cases} 0, & \text{if } n \neq m; \\ \det G, & \text{if } n = m \end{cases},$$

where G is the $n \times m$ -matrix whose (i, j) -th entry is $f(v_i, w_j)$. This bilinear form $\Lambda(f)$ is known as the bilinear form on $\Lambda(V)$ obtained by lifting the bilinear form f . Its restriction to the 1-st homogeneous component V of $\Lambda(V)$ is f .

The bilinear form $\Lambda(f)$ is symmetric if f is.

INPUT:

- M – a matrix over the same base ring as `self`, whose (i, j) -th entry is $f(e_i, e_j)$, where (e_1, e_2, \dots, e_N) is the standard basis of the module V for which `self = $\Lambda(V)$` (so that $N = \dim(V)$), and where f is the bilinear form which is to be lifted.

OUTPUT:

A bivariate function which takes two elements p and q of `self` to $\Lambda(f)(p, q)$.

Note: This takes a bilinear form on V as matrix, and returns a bilinear form on `self` as a function in two arguments. We do not return the bilinear form as a matrix since this matrix can be huge and one often needs just a particular value.

Todo

Implement a class for bilinear forms and rewrite this method to use that class.

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: M = Matrix(QQ, [[1, 2, 3], [2, 3, 4], [3, 4, 5]])
sage: Eform = E.lifted_bilinear_form(M)
sage: Eform
Bilinear Form from The exterior algebra of rank 3 over Rational
Field (+) The exterior algebra of rank 3 over Rational Field to
Rational Field
sage: Eform(x*y, y*z)
-1
sage: Eform(x*y, y)
0
sage: Eform(x*(y+z), y*z)
-3
sage: Eform(x*(y+z), y*(z+x))
0
sage: N = Matrix(QQ, [[3, 1, 7], [2, 0, 4], [-1, -3, -1]])
sage: N.determinant()
-8
sage: Eform = E.lifted_bilinear_form(N)
sage: Eform(x, E.one())
0
sage: Eform(x, x*z*y)
0
sage: Eform(E.one(), E.one())
1
sage: Eform(E.zero(), E.one())
0
sage: Eform(x, y)
1
sage: Eform(z, y)
-3
sage: Eform(x*z, y*z)
20
sage: Eform(x+x*y+x*y*z, z+z*y+z*y*x)
11
```

Todo

Another way to compute this bilinear form seems to be to map x and y to the appropriate Clifford algebra and there compute $x^t y$, then send the result back to the exterior algebra and return its constant coefficient. Or something like this. Once the maps to the Clifford and back are implemented, check if this is faster.

`ExteriorAlgebra.volume_form()`

Return the volume form of self.

Given the basis e_1, e_2, \dots, e_n of the underlying R -module, the volume form is defined as $e_1 \wedge e_2 \wedge \dots \wedge e_n$.

This depends on the choice of basis.

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: E.volume_form()
x^y^z
```

`class sage.algebras.clifford_algebra.ExteriorAlgebraBoundary (E, s_coeff)`

Bases: `sage.algebras.clifford_algebra.ExteriorAlgebraDifferential`

The boundary ∂ of an exterior algebra $\Lambda(L)$ defined by the structure coefficients of L .

Let L be a Lie algebra. We give the exterior algebra $E = \Lambda(L)$ a chain complex structure by considering a differential $\partial : \Lambda^{k+1}(L) \rightarrow \Lambda^k(L)$ defined by

$$\partial(x_1 \wedge x_2 \wedge \cdots \wedge x_{k+1}) = \sum_{i < j} (-1)^{i+j+1} [x_i, x_j] \wedge x_1 \wedge \cdots \wedge \hat{x}_i \wedge \cdots \wedge \hat{x}_j \wedge \cdots \wedge x_{k+1}$$

where \hat{x}_i denotes a missing index. The corresponding homology is the Lie algebra homology.

INPUT:

- `E` – an exterior algebra of a vector space L
- `s_coeff` – a dictionary whose keys are in $I \times I$, where I is the index set of the basis of the vector space L , and whose values can be coerced into 1-forms (degree 1 elements) in E ; this dictionary will be used to define the Lie algebra structure on L (indeed, the i -th coordinate of the Lie bracket of the j -th and k -th basis vectors of L for $j < k$ is set to be the value at the key (j, k) if this key appears in `s_coeff`, or otherwise the negated of the value at the key (k, j))

Warning: The values of `s_coeff` are supposed to be coercible into 1-forms in E ; but they can also be dictionaries themselves (in which case they are interpreted as giving the coordinates of vectors in L). In the interest of speed, these dictionaries are not sanitized or checked.

Warning: For any two distinct elements i and j of I , the dictionary `s_coeff` must have only one of the pairs (i, j) and (j, i) as a key. This is not checked.

EXAMPLES:

We consider the differential given by Lie algebra given by the cross product \times of \mathbf{R}^3 :

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: par = E.boundary({(0,1): z, (1,2): x, (2,0): y})
sage: par(x)
0
sage: par(x*y)
z
sage: par(x*y*z)
0
sage: par(x+y-y*z+x*y)
-x + z
sage: par(E.zero())
0
```

We check that $\partial \circ \partial = 0$:

```
sage: p2 = par * par
sage: all(p2(b) == 0 for b in E.basis())
True
```

Another example: the Lie algebra \mathfrak{sl}_2 , which has a basis e, f, h satisfying $[h, e] = 2e$, $[h, f] = -2f$, and $[e, f] = h$:

```

sage: E.<e,f,h> = ExteriorAlgebra(QQ)
sage: par = E.boundary({(0,1): h, (2,1): -2*f, (2,0): 2*e})
sage: par(E.zero())
0
sage: par(e)
0
sage: par(e*f)
h
sage: par(f*h)
2*f
sage: par(h*f)
-2*f
sage: C = par.chain_complex(); C
Chain complex with at most 4 nonzero terms over Rational Field
sage: ascii_art(C)
          [ 0 -2  0]          [0]
          [ 0  0  2]          [0]
    [0 0 0]    [ 1  0  0]    [0]
0 <-- C_0 <----- C_1 <----- C_2 <---- C_3 <-- 0
sage: C.homology()
{0: Vector space of dimension 1 over Rational Field,
 1: Vector space of dimension 0 over Rational Field,
 2: Vector space of dimension 0 over Rational Field,
 3: Vector space of dimension 1 over Rational Field}

```

Over the integers:

```

sage: C = par.chain_complex(R=ZZ); C
Chain complex with at most 4 nonzero terms over Integer Ring
sage: ascii_art(C)
          [ 0 -2  0]          [0]
          [ 0  0  2]          [0]
    [0 0 0]    [ 1  0  0]    [0]
0 <-- C_0 <----- C_1 <----- C_2 <---- C_3 <-- 0
sage: C.homology()
{0: Z, 1: C2 x C2, 2: 0, 3: Z}

```

REFERENCES:

- [Wikipedia article Exterior_algebra#Lie_algebra_homology](#)

chain_complex (*R=None*)

Return the chain complex over *R* determined by *self*.

INPUT:

- *R* – the base ring; the default is the base ring of the exterior algebra

EXAMPLES:

```

sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: par = E.boundary({(0,1): z, (1,2): x, (2,0): y})
sage: C = par.chain_complex(); C
Chain complex with at most 4 nonzero terms over Rational Field
sage: ascii_art(C)
          [ 0  0  1]          [0]
          [ 0 -1  0]          [0]
    [0 0 0]    [ 1  0  0]    [0]
0 <-- C_0 <----- C_1 <----- C_2 <---- C_3 <-- 0

```


class sage.algebras.clifford_algebra. **ExteriorAlgebraCoboundary** (*E*, *s_coeff*)
 Bases: *sage.algebras.clifford_algebra.ExteriorAlgebraDifferential*

The coboundary d of an exterior algebra $\Lambda(L)$ defined by the structure coefficients of a Lie algebra L .

Let L be a Lie algebra. We endow its exterior algebra $E = \Lambda(L)$ with a cochain complex structure by considering a differential $d : \Lambda^k(L) \rightarrow \Lambda^{k+1}(L)$ defined by

$$dx_i = \sum_{j < k} s_{jk}^i x_j x_k,$$

where (x_1, x_2, \dots, x_n) is a basis of L , and where s_{jk}^i is the x_i -coordinate of the Lie bracket $[x_j, x_k]$.

The corresponding cohomology is the Lie algebra cohomology of L .

This can also be thought of as the exterior derivative, in which case the resulting cohomology is the de Rham cohomology of a manifold whose exterior algebra of differential forms is E .

INPUT:

- *E* – an exterior algebra of a vector space L
- *s_coeff* – a dictionary whose keys are in $I \times I$, where I is the index set of the basis of the vector space L , and whose values can be coerced into 1-forms (degree 1 elements) in E ; this dictionary will be used to define the Lie algebra structure on L (indeed, the i -th coordinate of the Lie bracket of the j -th and k -th basis vectors of L for $j < k$ is set to be the value at the key (j, k) if this key appears in *s_coeff*, or otherwise the negated of the value at the key (k, j))

Warning: For any two distinct elements i and j of I , the dictionary *s_coeff* must have only one of the pairs (i, j) and (j, i) as a key. This is not checked.

EXAMPLES:

We consider the differential coming from the Lie algebra given by the cross product \times of \mathbf{R}^3 :

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: d = E.coboundary({(0,1): z, (1,2): x, (2,0): y})
sage: d(x)
y^z
sage: d(y)
-x^z
sage: d(x+y-y*z)
-x^z + y^z
sage: d(x*y)
0
sage: d(E.one())
0
sage: d(E.zero())
0
```

We check that $d \circ d = 0$:

```
sage: d2 = d * d
sage: all(d2(b) == 0 for b in E.basis())
True
```

Another example: the Lie algebra \mathfrak{sl}_2 , which has a basis e, f, h satisfying $[h, e] = 2e$, $[h, f] = -2f$, and $[e, f] = h$:

```

sage: E.<e,f,h> = ExteriorAlgebra(QQ)
sage: d = E.coboundary({(0,1): h, (2,1): -2*f, (2,0): 2*e})
sage: d(E.zero())
0
sage: d(e)
-2*e^h
sage: d(f)
2*f^h
sage: d(h)
e^f
sage: d(e*f)
0
sage: d(f*h)
0
sage: d(e*h)
0
sage: C = d.chain_complex(); C
Chain complex with at most 4 nonzero terms over Rational Field
sage: ascii_art(C)
          [ 0  0  1]          [0]
          [-2  0  0]          [0]
    [0 0 0]    [ 0  2  0]    [0]
0 <-- C_3 <----- C_2 <----- C_1 <---- C_0 <-- 0
sage: C.homology()
{0: Vector space of dimension 1 over Rational Field,
 1: Vector space of dimension 0 over Rational Field,
 2: Vector space of dimension 0 over Rational Field,
 3: Vector space of dimension 1 over Rational Field}

```

Over the integers:

```

sage: C = d.chain_complex(R=ZZ); C
Chain complex with at most 4 nonzero terms over Integer Ring
sage: ascii_art(C)
          [ 0  0  1]          [0]
          [-2  0  0]          [0]
    [0 0 0]    [ 0  2  0]    [0]
0 <-- C_3 <----- C_2 <----- C_1 <---- C_0 <-- 0
sage: C.homology()
{0: Z, 1: 0, 2: C2 x C2, 3: Z}

```

REFERENCES:

- [Wikipedia article Exterior_algebra#Differential_geometry](#)

chain_complex (*R=None*)

Return the chain complex over *R* determined by *self*.

INPUT:

- *R* – the base ring; the default is the base ring of the exterior algebra

EXAMPLES:

```

sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: d = E.coboundary({(0,1): z, (1,2): x, (2,0): y})
sage: C = d.chain_complex(); C
Chain complex with at most 4 nonzero terms over Rational Field
sage: ascii_art(C)
          [ 0  0  1]          [0]

```

		[0 -1 0]	[0]
[0 0 0]		[1 0 0]	[0]
0 <-- C_3 <-----	C_2 <-----	C_1 <----	C_0 <-- 0

class sage.algebras.clifford_algebra. **ExteriorAlgebraDifferential** (*E, s_coeff*)
Bases: sage.modules.with_basis.morphism.ModuleMorphismByLinearity ,
sage.structure.unique_representation.UniqueRepresentation

Internal class to store the data of a boundary or coboundary of an exterior algebra $\Lambda(L)$ defined by the structure coefficients of a Lie algebra L .

See [ExteriorAlgebraBoundary](#) and [ExteriorAlgebraCoboundary](#) for the actual classes, which inherit from this.

Warning: This is not a general class for differentials on the exterior algebra.

homology (*deg=None, **kws*)
Return the homology determined by *self*.

EXAMPLES:

```
sage: E.<x,y,z> = ExteriorAlgebra(QQ)
sage: par = E.boundary({(0,1): z, (1,2): x, (2,0): y})
sage: par.homology()
{0: Vector space of dimension 1 over Rational Field,
 1: Vector space of dimension 0 over Rational Field,
 2: Vector space of dimension 0 over Rational Field,
 3: Vector space of dimension 1 over Rational Field}
sage: d = E.coboundary({(0,1): z, (1,2): x, (2,0): y})
sage: d.homology()
{0: Vector space of dimension 1 over Rational Field,
 1: Vector space of dimension 0 over Rational Field,
 2: Vector space of dimension 0 over Rational Field,
 3: Vector space of dimension 1 over Rational Field}
```

4.4 Cluster algebras

This file constructs cluster algebras using the Parent-Element framework. The implementation mainly utilizes structural theorems from [FZ2007].

The key points being used here are these:

- cluster variables are parametrized by their g-vectors;
- g-vectors (together with c-vectors) provide a self-standing model for the combinatorics behind any cluster algebra;
- each cluster variable in any cluster algebra can be computed, by the separation of additions formula, from its g-vector and F-polynomial.

Accordingly this file provides three classes:

- [ClusterAlgebra](#)
- [ClusterAlgebraSeed](#)
- [ClusterAlgebraElement](#)

`ClusterAlgebra`, constructed as a subobject of `sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing`, is the frontend of this implementation. It provides all the algebraic features (like ring morphisms), it computes cluster variables, it is responsible for controlling the exploration of the exchange graph and serves as the repository for all the data recursively computed so far. In particular, all g-vectors and all F-polynomials of known cluster variables as well as a mutation path by which they can be obtained are recorded. In the optic of efficiency, this implementation does not store directly the exchange graph nor the exchange relations. Both of these could be added to `ClusterAlgebra` with minimal effort.

`ClusterAlgebraSeed` provides the combinatorial backbone for `ClusterAlgebra`. It is an auxiliary class and therefore its instances should **not** be directly created by the user. Rather it should be accessed via `ClusterAlgebra.current_seed()` and `ClusterAlgebra.initial_seed()`. The task of performing current seed mutations is delegated to this class. Seeds are considered equal if they have the same parent cluster algebra and they can be obtained from each other by a permutation of their data (i.e. if they coincide as unlabelled seeds). Cluster algebras whose initial seeds are equal in the above sense are not considered equal but are endowed with coercion maps to each other. More generally, a cluster algebra is endowed with coercion maps from any cluster algebra which is obtained by freezing a collection of initial cluster variables and/or permuting both cluster variables and coefficients.

`ClusterAlgebraElement` is a thin wrapper around `sage.rings.polynomial.laurent_polynomial.LaurentPolynomial` providing all the functions specific to cluster variables. Elements of a cluster algebra with principal coefficients have special methods and these are grouped in the subclass `PrincipalClusterAlgebraElement`.

One more remark about this implementation. Instances of `ClusterAlgebra` are built by identifying the initial cluster variables with the generators of `ClusterAlgebra.ambient()`. In particular, this forces a specific embedding into the ambient field of rational expressions. In view of this, although cluster algebras themselves are independent of the choice of initial seed, `ClusterAlgebra.mutate_initial()` is forced to return a different instance of `ClusterAlgebra`. At the moment there is no coercion implemented among the two instances but this could in principle be added to `ClusterAlgebra.mutate_initial()`.

REFERENCES:

- [FZ2007]
- [LLZ2014]
- [NZ2012]

AUTHORS:

- Dylan Rupel (2015-06-15): initial version
- Salvatore Stella (2015-06-15): initial version

EXAMPLES:

We begin by creating a simple cluster algebra and printing its initial exchange matrix:

```
sage: A = ClusterAlgebra(['A', 2]); A
A Cluster Algebra with cluster variables x0, x1 and no coefficients over Integer Ring
sage: A.b_matrix()
[ 0  1]
[-1  0]
```

A is of finite type so we can explore all its exchange graph:

```
sage: A.explore_to_depth(infinity)
```

and get all its g-vectors, F-polynomials, and cluster variables:

```
sage: A.g_vectors_so_far()
[(0, 1), (0, -1), (1, 0), (-1, 1), (-1, 0)]
```

```
sage: A.F_polynomials_so_far()
[1, u1 + 1, 1, u0 + 1, u0*u1 + u0 + 1]
sage: A.cluster_variables_so_far()
[x1, (x0 + 1)/x1, x0, (x1 + 1)/x0, (x0 + x1 + 1)/(x0*x1)]
```

Simple operations among cluster variables behave as expected:

```
sage: s = A.cluster_variable((0, -1)); s
(x0 + 1)/x1
sage: t = A.cluster_variable((-1, 1)); t
(x1 + 1)/x0
sage: t + s
(x0^2 + x1^2 + x0 + x1)/(x0*x1)
sage: _.parent() == A
True
sage: t - s
(-x0^2 + x1^2 - x0 + x1)/(x0*x1)
sage: _.parent() == A
True
sage: t*s
(x0*x1 + x0 + x1 + 1)/(x0*x1)
sage: _.parent() == A
True
sage: t/s
(x1^2 + x1)/(x0^2 + x0)
sage: _.parent() == A
False
```

Division is not guaranteed to yield an element of A so it returns an element of $A.ambient().fraction_field()$ instead:

```
sage: (t/s).parent() == A.ambient().fraction_field()
True
```

We can compute denominator vectors of any element of A :

```
sage: (t*s).d_vector()
(1, 1)
```

Since we are in rank 2 and we do not have coefficients we can compute the greedy element associated to any denominator vector:

```
sage: A.rank() == 2 and A.coefficients() == ()
True
sage: A.greedy_element((1, 1))
(x0 + x1 + 1)/(x0*x1)
sage: _ == t*s
False
```

not surprising since there is no cluster in A containing both t and s :

```
sage: seeds = A.seeds(mutating_F=False)
sage: [ S for S in seeds if (0, -1) in S and (-1, 1) in S ]
[]
```

indeed:

```
sage: A.greedy_element((1, 1)) == A.cluster_variable((-1, 0))
True
```

Disabling F-polynomials in the computation just done was redundant because we already explored the whole exchange graph before. Though in different circumstances it could have saved us considerable time.

g-vectors and F-polynomials can be computed from elements of A only if A has principal coefficients at the initial seed:

```
sage: (t*s).g_vector()
Traceback (most recent call last):
...
AttributeError: 'ClusterAlgebra_with_category.element_class' object has no attribute
↳ 'g_vector'
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True)
sage: A.explore_to_depth(infinity)
sage: s = A.cluster_variable((0, -1)); s
(x0*y1 + 1)/x1
sage: t = A.cluster_variable((-1, 1)); t
(x1 + y0)/x0
sage: (t*s).g_vector()
(-1, 0)
sage: (t*s).F_polynomial()
u0*u1 + u0 + u1 + 1
sage: (t*s).is_homogeneous()
True
sage: (t+s).is_homogeneous()
False
sage: (t+s).homogeneous_components()
{(-1, 1): (x1 + y0)/x0, (0, -1): (x0*y1 + 1)/x1}
```

Each cluster algebra is endowed with a reference to a current seed; it could be useful to assign a name to it:

```
sage: A = ClusterAlgebra(['F', 4])
sage: len(A.g_vectors_so_far())
4
sage: A.current_seed()
The initial seed of a Cluster Algebra with cluster variables x0, x1, x2, x3
and no coefficients over Integer Ring
sage: A.current_seed() == A.initial_seed()
True
sage: S = A.current_seed()
sage: S.b_matrix()
[ 0  1  0  0]
[-1  0 -1  0]
[ 0  2  0  1]
[ 0  0 -1  0]
sage: S.g_matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
sage: S.cluster_variables()
[x0, x1, x2, x3]
```

and use S to walk around the exchange graph of A :

```

sage: S.mutate(0); S
The seed of a Cluster Algebra with cluster variables x0, x1, x2, x3
and no coefficients over Integer Ring obtained from the initial
by mutating in direction 0
sage: S.b_matrix()
[ 0 -1  0  0]
[ 1  0 -1  0]
[ 0  2  0  1]
[ 0  0 -1  0]
sage: S.g_matrix()
[-1  0  0  0]
[ 1  1  0  0]
[ 0  0  1  0]
[ 0  0  0  1]
sage: S.cluster_variables()
[(x1 + 1)/x0, x1, x2, x3]
sage: S.mutate('sinks'); S
The seed of a Cluster Algebra with cluster variables x0, x1, x2, x3
and no coefficients over Integer Ring obtained from the initial
by mutating along the sequence [0, 2]
sage: S.mutate([2, 3, 2, 1, 0]); S
The seed of a Cluster Algebra with cluster variables x0, x1, x2, x3
and no coefficients over Integer Ring obtained from the initial
by mutating along the sequence [0, 3, 2, 1, 0]
sage: S.g_vectors()
[(0, 1, -2, 0), (-1, 2, -2, 0), (0, 1, -1, 0), (0, 0, 0, -1)]
sage: S.cluster_variable(3)
(x2 + 1)/x3

```

Walking around by mutating `S` updates the informations stored in `A` :

```

sage: len(A.g_vectors_so_far())
10
sage: A.current_seed().path_from_initial_seed()
[0, 3, 2, 1, 0]
sage: A.current_seed() == S
True

```

Starting from `A.initial_seed()` still records data in `A` but does not update `A.current_seed()` :

```

sage: S1 = A.initial_seed()
sage: S1.mutate([2, 1, 3])
sage: len(A.g_vectors_so_far())
11
sage: S1 == A.current_seed()
False

```

Since `ClusterAlgebra` inherits from `UniqueRepresentation`, computed data is shared across instances:

```

sage: A1 = ClusterAlgebra(['F', 4])
sage: A1 is A
True
sage: len(A1.g_vectors_so_far())
11

```

It can be useful, at times to forget all computed data. Because of `UniqueRepresentation` this cannot be achieved by simply creating a new instance; instead it has to be manually triggered by:

```
sage: A.clear_computed_data()
sage: len(A.g_vectors_so_far())
4
```

Given a cluster algebra A we may be looking for a specific cluster variable:

```
sage: A = ClusterAlgebra(['E', 8, 1])
sage: A.find_g_vector((-1, 1, -1, 1, -1, 1, 0, 0, 1), depth=2)
sage: A.find_g_vector((-1, 1, -1, 1, -1, 1, 0, 0, 1))
[0, 1, 2, 4, 3]
```

This also performs mutations of F-polynomials:

```
sage: A.F_polynomial((-1, 1, -1, 1, -1, 1, 0, 0, 1))
u0*u1*u2*u3*u4 + u0*u1*u2*u4 + u0*u2*u3*u4 + u0*u1*u2 + u0*u2*u4
+ u2*u3*u4 + u0*u2 + u0*u4 + u2*u4 + u0 + u2 + u4 + 1
```

which might not be a good idea in algebras that are too big. One workaround is to first disable F-polynomials and then recompute only the desired mutations:

```
sage: A.reset_exploring_iterator(mutating_F=False) # long time
sage: A.find_g_vector((-1, 1, -2, 2, -1, 1, -1, 1, 1)) # long time
[1, 0, 2, 6, 5, 4, 3, 8, 1]
sage: A.current_seed().mutate(_) # long time
sage: A.F_polynomial((-1, 1, -2, 2, -1, 1, -1, 1, 1)) # long time
u0*u1^2*u2^2*u3*u4*u5*u6*u8 +
...
2*u2 + u4 + u6 + 1
```

We can manually freeze cluster variables and get coercions in between the two algebras:

```
sage: A = ClusterAlgebra(['F', 4]); A
A Cluster Algebra with cluster variables x0, x1, x2, x3 and no coefficients
over Integer Ring
sage: A1 = ClusterAlgebra(A.b_matrix().matrix_from_columns([0, 1, 2]), coefficient_
↪prefix='x'); A1
A Cluster Algebra with cluster variables x0, x1, x2 and coefficient x3
over Integer Ring
sage: A.has_coerce_map_from(A1)
True
```

and we also have an immersion of $A.\text{base}()$ into A and of A into $A.\text{ambient}()$:

```
sage: A.has_coerce_map_from(A.base())
True
sage: A.ambient().has_coerce_map_from(A)
True
```

but there is currently no coercion in between algebras obtained by mutating at the initial seed:

```
sage: A1 = A.mutate_initial(0); A1
A Cluster Algebra with cluster variables x0, x1, x2, x3 and no coefficients
over Integer Ring
sage: A.b_matrix() == A1.b_matrix()
False
sage: map(lambda (X, Y): X.has_coerce_map_from(Y), [(A, A1), (A1, A)])
[False, False]
```



```
class sage.algebras.cluster_algebra.ClusterAlgebra ( Q, **kwargs)
    Bases: sage.structure.parent.Parent, sage.structure.unique_representation.UniqueRepresentation
```

A Cluster Algebra.

INPUT:

- `data` – some data defining a cluster algebra; it can be anything that can be parsed by `ClusterQuiver`
- `scalars` – a ring (default \mathbf{Z}); the scalars over which the cluster algebra is defined
- `cluster_variable_prefix` – string (default `'x'`); it needs to be a valid variable name
- `cluster_variable_names` – a list of strings; each element needs to be a valid variable name; supersedes `cluster_variable_prefix`
- `coefficient_prefix` – string (default `'y'`); it needs to be a valid variable name.
- `coefficient_names` – a list of strings; each element needs to be a valid variable name; supersedes `cluster_variable_prefix`
- `principal_coefficients` – bool (default `False`); supersedes any coefficient defined by `data`

ALGORITHM:

The implementation is mainly based on [FZ2007] and [NZ2012].

EXAMPLES:

```
sage: B = matrix([(0, 1, 0, 0), (-1, 0, -1, 0), (0, 1, 0, 1), (0, 0, -2, 0), (-1, 0, 0, 0), (0, -1, 0, 0)])
sage: A = ClusterAlgebra(B); A
A Cluster Algebra with cluster variables x0, x1, x2, x3
and coefficients y0, y1 over Integer Ring
sage: A.gens()
(x0, x1, x2, x3, y0, y1)
sage: A = ClusterAlgebra(['A', 2]); A
A Cluster Algebra with cluster variables x0, x1 and no coefficients
over Integer Ring
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True); A.gens()
(x0, x1, y0, y1)
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True, coefficient_
    ↪prefix='x'); A.gens()
(x0, x1, x2, x3)
sage: A = ClusterAlgebra(['A', 3], principal_coefficients=True, cluster_variable_
    ↪names=['a', 'b', 'c']); A.gens()
(a, b, c, y0, y1, y2)
sage: A = ClusterAlgebra(['A', 3], principal_coefficients=True, cluster_variable_
    ↪names=['a', 'b'])
Traceback (most recent call last):
...
ValueError: cluster_variable_names should be a list of 3 valid variable names
sage: A = ClusterAlgebra(['A', 3], principal_coefficients=True, coefficient_
    ↪names=['a', 'b', 'c']); A.gens()
(x0, x1, x2, a, b, c)
sage: A = ClusterAlgebra(['A', 3], principal_coefficients=True, coefficient_
    ↪names=['a', 'b'])
Traceback (most recent call last):
...
ValueError: coefficient_names should be a list of 3 valid variable names
```

F_polynomial (*g_vector*)

Return the F-polynomial with g-vector *g_vector* if it has been found.

INPUT:

- *g_vector* – tuple; the g-vector of the F-polynomial to return

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2])
sage: A.clear_computed_data()
sage: A.F_polynomial((-1, 1))
Traceback (most recent call last):
...
KeyError: 'the g-vector (-1, 1) has not been found yet'
sage: A.initial_seed().mutate(0, mutating_F=False)
sage: A.F_polynomial((-1, 1))
Traceback (most recent call last):
...
KeyError: 'the F-polynomial with g-vector (-1, 1) has not been computed yet;
you can compute it by mutating from the initial seed along the sequence [0]'
sage: A.initial_seed().mutate(0)
sage: A.F_polynomial((-1, 1))
u0 + 1
```

F_polynomials ()

Return an iterator producing all the F_polynomials of *self*.

ALGORITHM:

This method does not use the caching framework provided by *self*, but recomputes all the F-polynomials from scratch. On the other hand it stores the results so that other methods like *F_polynomials_so_far()* can access them afterwards.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: len(list(A.F_polynomials()))
9
```

F_polynomials_so_far ()

Return a list of the F-polynomials encountered so far.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2])
sage: A.clear_computed_data()
sage: A.current_seed().mutate(0)
sage: A.F_polynomials_so_far()
[1, 1, u0 + 1]
```

ambient ()

Return the Laurent polynomial ring containing *self*.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True)
sage: A.ambient()
Multivariate Laurent Polynomial Ring in x0, x1, y0, y1 over Integer Ring
```

b_matrix ()

Return the initial exchange matrix of `self`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2])
sage: A.b_matrix()
[ 0  1]
[-1  0]
```

clear_computed_data ()

Clear the cache of computed g-vectors and F-polynomials and reset both the current seed and the exploring iterator.

EXAMPLES:: sage: A = ClusterAlgebra(['A', 2]) sage: A.clear_computed_data() sage: A.g_vectors_so_far() [(0, 1), (1, 0)] sage: A.current_seed().mutate([1, 0]) sage: A.g_vectors_so_far() [(0, 1), (0, -1), (1, 0), (-1, 0)] sage: A.clear_computed_data() sage: A.g_vectors_so_far() [(0, 1), (1, 0)]

cluster_fan (depth=+Infinity)

Return the cluster fan (the fan of g-vectors) of `self`.

INPUT:

- `depth` – a positive integer or infinity (default `infinity`); the maximum depth at which to compute

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2])
sage: A.cluster_fan()
Rational polyhedral fan in 2-d lattice N
```

cluster_variable (g_vector)

Return the cluster variable with g-vector `g_vector` if it has been found.

INPUT:

- `g_vector` – tuple; the g-vector of the cluster variable to return

ALGORITHM:

This function computes cluster variables from their g-vectors and F-polynomials using the “separation of additions” formula of Theorem 3.7 in [FZ2007].

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2])
sage: A.initial_seed().mutate(0)
sage: A.cluster_variable((-1, 1))
(x1 + 1)/x0
```

cluster_variables ()

Return an iterator producing all the cluster variables of `self`.

ALGORITHM:

This method does not use the caching framework provided by `self`, but recomputes all the cluster variables from scratch. On the other hand it stores the results so that other methods like `cluster_variables_so_far()` can access them afterwards.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: len(list(A.cluster_variables()))
9
```

cluster_variables_so_far ()

Return a list of the cluster variables encountered so far.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2])
sage: A.clear_computed_data()
sage: A.current_seed().mutate(0)
sage: A.cluster_variables_so_far()
[x1, x0, (x1 + 1)/x0]
```

coefficient (j)

Return the j -th coefficient of `self`.

INPUT:

- j – an integer in `range(self.parent().rank())`; the index of the coefficient to return

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True)
sage: A.coefficient(0)
y0
```

coefficient_names ()

Return the list of coefficient names.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: A.coefficient_names()
()
sage: A1 = ClusterAlgebra(['B', 2], principal_coefficients=True)
sage: A1.coefficient_names()
('y0', 'y1')
sage: A2 = ClusterAlgebra(['C', 3], principal_coefficients=True, coefficient_
↳ prefix='x')
sage: A2.coefficient_names()
('x3', 'x4', 'x5')
```

coefficients ()

Return the list of coefficients of `self`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True)
sage: A.coefficients()
(y0, y1)
sage: A1 = ClusterAlgebra(['B', 2])
sage: A1.coefficients()
()
```

contains_seed (seed)

Test if `seed` is a seed of `self`.

INPUT:

•seed – a *ClusterAlgebraSeed*

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True); A
A Cluster Algebra with cluster variables x0, x1 and coefficients y0, y1 over
↳ Integer Ring
sage: S = copy(A.current_seed())
sage: A.contains_seed(S)
True
```

current_seed ()

Return the current seed of self.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2])
sage: A.clear_computed_data()
sage: A.current_seed()
The initial seed of a Cluster Algebra with cluster variables x0, x1
and no coefficients over Integer Ring
```

explore_to_depth (depth)

Explore the exchange graph of self up to distance depth from the initial seed.

INPUT:

•depth – a positive integer or infinity; the maximum depth at which to stop searching

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 4])
sage: A.explore_to_depth(infinity)
sage: len(A.g_vectors_so_far())
14
```

find_g_vector (g_vector, depth=+Infinity)

Return a mutation sequence to obtain a seed containing the g-vector g_vector from the initial seed.

INPUT:

•g_vector – a tuple: the g-vector to find

•depth – a positive integer or infinity (default infinity); the maximum distance from self.current_seed to reach

OUTPUT:

This function returns a list of integers if it can find g_vector, otherwise it returns None. If the exploring iterator stops, it means that the algebra is of finite type and g_vector is not the g-vector of any cluster variable. In this case the function resets the iterator and raises an error.

EXAMPLES:

```
sage: A = ClusterAlgebra(['G', 2], principal_coefficients=True)
sage: A.clear_computed_data()
sage: A.find_g_vector((-2, 3), depth=2)
sage: A.find_g_vector((-2, 3), depth=3)
[0, 1, 0]
sage: A.find_g_vector((1, 1), depth=3)
sage: A.find_g_vector((1, 1), depth=4)
Traceback (most recent call last):
```

```
...
ValueError: (1, 1) is not the g-vector of any cluster variable of a
Cluster Algebra with cluster variables x0, x1 and coefficients y0, y1
over Integer Ring
```

g_vectors (*mutating_F=True*)

Return an iterator producing all the g-vectors of *self* .

INPUT:

- *mutating_F* – bool (default `True`); whether to compute F-polynomials; disable this for speed considerations

ALGORITHM:

This method does not use the caching framework provided by *self* , but recomputes all the g-vectors from scratch. On the other hand it stores the results so that other methods like *g_vectors_so_far()* can access them afterwards.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: len(list(A.g_vectors()))
9
```

g_vectors_so_far ()

Return a list of the g-vectors of cluster variables encountered so far.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2])
sage: A.clear_computed_data()
sage: A.current_seed().mutate(0)
sage: A.g_vectors_so_far()
[(0, 1), (1, 0), (-1, 1)]
```

gens ()

Return the list of initial cluster variables and coefficients of *self* .

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True)
sage: A.gens()
(x0, x1, y0, y1)
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True, coefficient_
↳ prefix='x')
sage: A.gens()
(x0, x1, x2, x3)
```

greedy_element (*d_vector*)

Return the greedy element with denominator vector *d_vector* .

INPUT:

- *d_vector* – tuple of 2 integers; the denominator vector of the element to compute

ALGORITHM:

This implements greedy elements of a rank 2 cluster algebra using Equation (1.5) from [LLZ2014].

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', [1, 1], 1])
sage: A.greedy_element((1, 1))
(x0^2 + x1^2 + 1)/(x0*x1)
```

initial_cluster_variable (j)

Return the j -th initial cluster variable of `self`.

INPUT:

- j – an integer in `range(self.parent().rank())`; the index of the cluster variable to return

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True)
sage: A.initial_cluster_variable(0)
x0
```

initial_cluster_variable_names ()

Return the list of initial cluster variable names.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True)
sage: A.initial_cluster_variable_names()
('x0', 'x1')
sage: A1 = ClusterAlgebra(['B', 2], cluster_variable_prefix='a')
sage: A1.initial_cluster_variable_names()
('a0', 'a1')
```

initial_cluster_variables ()

Return the list of initial cluster variables of `self`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True)
sage: A.initial_cluster_variables()
(x0, x1)
```

initial_seed ()

Return the initial seed of `self`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2])
sage: A.initial_seed()
The initial seed of a Cluster Algebra with cluster variables x0, x1 and no_
↪coefficients over Integer Ring
```

lift (x)

Return x as an element of `ambient()`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True)
sage: x = A.cluster_variable((1, 0))
sage: A.lift(x).parent()
Multivariate Laurent Polynomial Ring in x0, x1, y0, y1 over Integer Ring
```

lower_bound ()

Return the lower bound associated to `self`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['F', 4])
sage: A.lower_bound()
Traceback (most recent call last):
...
NotImplementedError: not implemented yet
```

mutate_initial (*direction*)

Return the cluster algebra obtained by mutating *self* at the initial seed.

INPUT:

- *direction* – in which direction(s) to mutate, it can be:
 - an integer in `range(self.rank())` to mutate in one direction only
 - an iterable of such integers to mutate along a sequence
 - a string “sinks” or “sources” to mutate at all sinks or sources simultaneously

ALGORITHM:

This function computes data for the new algebra from known data for the old algebra using Equation (4.2) from [NZ2012] for g-vectors, and Equation (6.21) from [FZ2007] for F-polynomials. The exponent h in the formula for F-polynomials is $-\min(0, \text{old_g_vect}[k])$ due to [NZ2012] Proposition 4.2.

EXAMPLES:

```
sage: A = ClusterAlgebra(['F', 4])
sage: A.explore_to_depth(infinity)
sage: B = A.b_matrix()
sage: B.mutate(0)
sage: A1 = ClusterAlgebra(B)
sage: A1.explore_to_depth(infinity)
sage: A2 = A1.mutate_initial(0)
sage: A2._F_poly_dict == A._F_poly_dict
True
```

Check that we did not mess up the original algebra because of UniqueRepresentation:

```
sage: A = ClusterAlgebra(['A', 2])
sage: A.mutate_initial(0) is A
False
```

rank ()

Return the rank of *self*, i.e. the number of cluster variables in any seed.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True); A
A Cluster Algebra with cluster variables x0, x1
and coefficients y0, y1 over Integer Ring
sage: A.rank()
2
```

reset_current_seed ()

Reset the value reported by `current_seed()` to `initial_seed()`.

EXAMPLES:


```

sage: A = ClusterAlgebra(['A', 2])
sage: A.clear_computed_data()
sage: A.current_seed().mutate([1, 0])
sage: A.current_seed() == A.initial_seed()
False
sage: A.reset_current_seed()
sage: A.current_seed() == A.initial_seed()
True

```

reset_exploring_iterator (*mutating_F=True*)

Reset the iterator used to explore *self* .

INPUT:

- *mutating_F* – bool (default *True*); whether to also compute F-polynomials; disable this for speed considerations

EXAMPLES:

```

sage: A = ClusterAlgebra(['A', 4])
sage: A.clear_computed_data()
sage: A.reset_exploring_iterator(mutating_F=False)
sage: A.explore_to_depth(infinity)
sage: len(A.g_vectors_so_far())
14
sage: len(A.F_polynomials_so_far())
4

```

retract (*x*)

Return *x* as an element of *self* .

EXAMPLES:

```

sage: A = ClusterAlgebra(['A', 2], principal_coefficients=True)
sage: L = A.ambient()
sage: x = L.gen(0)
sage: A.retract(x).parent()
A Cluster Algebra with cluster variables x0, x1 and coefficients y0, y1 over_
↪ Integer Ring

```

scalars ()

Return the ring of scalars over which *self* is defined.

EXAMPLES:

```

sage: A = ClusterAlgebra(['A', 2])
sage: A.scalars()
Integer Ring

```

seeds (***kwargs*)

Return an iterator running over seeds of *self* .

INPUT:

- *from_current_seed* – bool (default *False*); whether to start the iterator from *current_seed()* or *initial_seed()*
- *mutating_F* – bool (default *True*); whether to compute F-polynomials also; disable this for speed considerations

- `allowed_directions` – iterable of integers (default `range(self.rank())`); the directions in which to mutate
- `depth` – a positive integer or infinity (default `infinity`); the maximum depth at which to stop searching
- `catch_keyboard_interrupt` – bool (default `False`); whether to catch `KeyboardInterrupt` and return it rather than raising an exception – this allows the iterator returned by this method to be resumed after being interrupted

ALGORITHM:

This function traverses the exchange graph in a breadth-first search.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 4])
sage: A.clear_computed_data()
sage: seeds = A.seeds(allowed_directions=[3, 0, 1])
sage: _ = list(seeds)
sage: A.g_vectors_so_far()
[(-1, 0, 0, 0),
 (1, 0, 0, 0),
 (0, 0, 0, 1),
 (0, -1, 0, 0),
 (0, 0, 1, 0),
 (0, 1, 0, 0),
 (-1, 1, 0, 0),
 (0, 0, 0, -1)]
```

set_current_seed (seed)

Set the value reported by `current_seed()` to `seed`, if it makes sense.

INPUT:

- `seed` – a `ClusterAlgebraSeed`

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2])
sage: A.clear_computed_data()
sage: S = copy(A.current_seed())
sage: S.mutate([0, 1, 0])
sage: A.current_seed() == S
False
sage: A.set_current_seed(S)
sage: A.current_seed() == S
True
sage: A1 = ClusterAlgebra(['B', 2])
sage: A.set_current_seed(A1.initial_seed())
Traceback (most recent call last):
...
ValueError: This is not a seed in this cluster algebra
```

theta_basis_element (g_vector)

Return the element of the theta basis with g-vector `g_vector`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['F', 4])
sage: A.theta_basis_element((1, 0, 0, 0))
```

```
Traceback (most recent call last):
...
NotImplementedError: not implemented yet
```

upper_bound ()

Return the upper bound associated to `self`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['F', 4])
sage: A.upper_bound()
Traceback (most recent call last):
...
NotImplementedError: not implemented yet
```

upper_cluster_algebra ()

Return the upper cluster algebra associated to `self`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['F', 4])
sage: A.upper_cluster_algebra()
Traceback (most recent call last):
...
NotImplementedError: not implemented yet
```

class `sage.algebras.cluster_algebra.ClusterAlgebraElement`

Bases: `sage.structure.element_wrapper.ElementWrapper`

An element of a cluster algebra.

d_vector ()

Return the denominator vector of `self` as a tuple of integers.

EXAMPLES:

```
sage: A = ClusterAlgebra(['F', 4], principal_coefficients=True)
sage: A.current_seed().mutate([0, 2, 1])
sage: x = A.cluster_variable((-1, 2, -2, 2)) * A.cluster_variable((0, 0, 0, 1))**2
sage: x.d_vector()
(1, 1, 2, -2)
```

class `sage.algebras.cluster_algebra.ClusterAlgebraSeed (B, C, G, parent, **kwargs)`

Bases: `sage.structure.sage_object.SageObject`

A seed in a Cluster Algebra.

INPUT:

- `B` – a skew-symmetrizable integer matrix
- `C` – the matrix of `c`-vectors of `self`
- `G` – the matrix of `g`-vectors of `self`
- `parent` – `ClusterAlgebra`; the algebra to which the seed belongs
- `path` – list (default `[]`); the mutation sequence from the initial seed of `parent` to `self`

Warning: Seeds should **not** be created manually: no test is performed to assert that they are built from consistent data nor that they really are seeds of `parent`. If you create seeds with inconsistent data all sort of things can go wrong, even `__eq__()` is no longer guaranteed to give correct answers. Use at your own risk.

F_polynomial (*j*)

Return the *j*-th F-polynomial of `self`.

INPUT:

- *j* – an integer in `range(self.parent().rank())`; the index of the F-polynomial to return

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: S = A.initial_seed()
sage: S.F_polynomial(0)
1
```

F_polynomials ()

Return all the F-polynomials of `self`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: S = A.initial_seed()
sage: S.F_polynomials()
[1, 1, 1]
```

b_matrix ()

Return the exchange matrix of `self`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: S = A.initial_seed()
sage: S.b_matrix()
[ 0  1  0]
[-1  0 -1]
[ 0  1  0]
```

c_matrix ()

Return the matrix whose columns are the c-vectors of `self`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: S = A.initial_seed()
sage: S.c_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
```

c_vector (*j*)

Return the *j*-th c-vector of `self`.

INPUT:

- *j* – an integer in `range(self.parent().rank())`; the index of the c-vector to return

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: S = A.initial_seed()
sage: S.c_vector(0)
(1, 0, 0)
sage: S.mutate(0)
sage: S.c_vector(0)
(-1, 0, 0)
sage: S.c_vector(1)
(1, 1, 0)
```

c_vectors ()

Return all the c-vectors of `self`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: S = A.initial_seed()
sage: S.c_vectors()
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]
```

cluster_variable (j)

Return the `j`-th cluster variable of `self`.

INPUT:

- `j` – an integer in `range(self.parent().rank())`; the index of the cluster variable to return

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: S = A.initial_seed()
sage: S.cluster_variable(0)
x0
sage: S.mutate(0)
sage: S.cluster_variable(0)
(x1 + 1)/x0
```

cluster_variables ()

Return all the cluster variables of `self`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 3])
sage: S = A.initial_seed()
sage: S.cluster_variables()
[x0, x1, x2]
```

depth ()

Return the length of a mutation sequence from the initial seed of `parent()` to `self`.

Warning: This is the length of the mutation sequence returned by `path_from_initial_seed()`, which need not be the shortest possible.

EXAMPLES:

```

sage: A = ClusterAlgebra(['A', 2])
sage: S1 = A.initial_seed()
sage: S1.mutate([0, 1, 0, 1])
sage: S1.depth()
4
sage: S2 = A.initial_seed()
sage: S2.mutate(1)
sage: S2.depth()
1
sage: S1 == S2
True

```

g_matrix ()

Return the matrix whose columns are the g-vectors of self .

EXAMPLES:

```

sage: A = ClusterAlgebra(['A', 3])
sage: S = A.initial_seed()
sage: S.g_matrix()
[1 0 0]
[0 1 0]
[0 0 1]

```

g_vector (j)

Return the j -th g-vector of self .

INPUT:

- j – an integer in range (self.parent().rank()) ; the index of the g-vector to return

EXAMPLES:

```

sage: A = ClusterAlgebra(['A', 3])
sage: S = A.initial_seed()
sage: S.g_vector(0)
(1, 0, 0)

```

g_vectors ()

Return all the g-vectors of self .

EXAMPLES:

```

sage: A = ClusterAlgebra(['A', 3])
sage: S = A.initial_seed()
sage: S.g_vectors()
[(1, 0, 0), (0, 1, 0), (0, 0, 1)]

```

mutate (direction, **kwargs)

Mutate self .

INPUT:

- direction – in which direction(s) to mutate, it can be:
 - an integer in range (self.rank()) to mutate in one direction only
 - an iterable of such integers to mutate along a sequence
 - a string “sinks” or “sources” to mutate at all sinks or sources simultaneously
- inplace – bool (default True); whether to mutate in place or to return a new object

• `mutating_F` – bool (default `True`); whether to compute F-polynomials while mutating

Note: While knowing F-polynomials is essential to computing cluster variables, the process of mutating them is quite slow. If you care only about combinatorial data like g-vectors and c-vectors, setting `mutating_F=False` yields significant benefits in terms of speed.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2])
sage: S = A.initial_seed()
sage: S.mutate(0); S
The seed of a Cluster Algebra with cluster variables x0, x1
and no coefficients over Integer Ring obtained from the initial
by mutating in direction 0
sage: S.mutate(5)
Traceback (most recent call last):
...
ValueError: cannot mutate in direction 5
```

parent ()

Return the parent of `self`.

EXAMPLES:

```
sage: A = ClusterAlgebra(['B', 3])
sage: A.current_seed().parent() == A
True
```

path_from_initial_seed ()

Return a mutation sequence from the initial seed of `parent()` to `self`.

Warning: This is the path used to compute `self` and it does not have to be the shortest possible.

EXAMPLES:

```
sage: A = ClusterAlgebra(['A', 2])
sage: S1 = A.initial_seed()
sage: S1.mutate([0, 1, 0, 1])
sage: S1.path_from_initial_seed()
[0, 1, 0, 1]
sage: S2 = A.initial_seed()
sage: S2.mutate(1)
sage: S2.path_from_initial_seed()
[1]
sage: S1 == S2
True
```

class `sage.algebras.cluster_algebra.PrincipalClusterAlgebraElement`

Bases: `sage.algebras.cluster_algebra.ClusterAlgebraElement`

An element in a cluster algebra with principle coefficients.

F_polynomial ()

Return the F-polynomial of `self`.

EXAMPLES:

```

sage: A = ClusterAlgebra(['B', 2], principal_coefficients=True)
sage: S = A.initial_seed()
sage: S.mutate([0, 1, 0])
sage: S.cluster_variable(0).F_polynomial() == S.F_polynomial(0)
True
sage: sum(A.initial_cluster_variables()).F_polynomial()
Traceback (most recent call last):
...
ValueError: this element is not homogeneous

```

g_vector ()

Return the g-vector of self .

EXAMPLES:

```

sage: A = ClusterAlgebra(['B', 2], principal_coefficients=True)
sage: A.cluster_variable((1, 0)).g_vector() == (1, 0)
True
sage: sum(A.initial_cluster_variables()).g_vector()
Traceback (most recent call last):
...
ValueError: this element is not homogeneous

```

homogeneous_components ()

Return a dictionary of the homogeneous components of self .

OUTPUT:

A dictionary whose keys are homogeneous degrees and whose values are the summands of self of the given degree.

EXAMPLES:

```

sage: A = ClusterAlgebra(['B', 2], principal_coefficients=True)
sage: x = A.cluster_variable((1, 0)) + A.cluster_variable((0, 1))
sage: x.homogeneous_components()
{(0, 1): x1, (1, 0): x0}

```

is_homogeneous ()

Return True if self is a homogeneous element of self.parent() .

EXAMPLES:

```

sage: A = ClusterAlgebra(['B', 2], principal_coefficients=True)
sage: A.cluster_variable((1, 0)).is_homogeneous()
True
sage: x = A.cluster_variable((1, 0)) + A.cluster_variable((0, 1))
sage: x.is_homogeneous()
False

```

4.5 Descent Algebras

AUTHORS:

- Travis Scrimshaw (2013-07-28): Initial version


```
class sage.combinat.descent_algebra.DescentAlgebra ( R, n)
    Bases: sage.structure.unique_representation.UniqueRepresentation,
           sage.structure.parent.Parent
```

Solomon's descent algebra.

The descent algebra Σ_n over a ring R is a subalgebra of the symmetric group algebra RS_n . (The product in the latter algebra is defined by $(pq)(i) = q(p(i))$ for any two permutations p and q in S_n and every $i \in \{1, 2, \dots, n\}$. The algebra Σ_n inherits this product.)

There are three bases currently implemented for Σ_n :

- the standard basis D_S of (sums of) descent classes, indexed by subsets S of $\{1, 2, \dots, n-1\}$,
- the subset basis B_p , indexed by compositions p of n ,
- the idempotent basis I_p , indexed by compositions p of n , which is used to construct the mutually orthogonal idempotents of the symmetric group algebra.

The idempotent basis is only defined when R is a \mathbb{Q} -algebra.

We follow the notations and conventions in [GR1989], apart from the order of multiplication being different from the one used in that article. Schocker's exposition [Schocker2004], in turn, uses the same order of multiplication as we are, but has different notations for the bases.

INPUT:

- R – the base ring
- n – a nonnegative integer

REFERENCES:

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: D = DA.D(); D
Descent algebra of 4 over Rational Field in the standard basis
sage: B = DA.B(); B
Descent algebra of 4 over Rational Field in the subset basis
sage: I = DA.I(); I
Descent algebra of 4 over Rational Field in the idempotent basis
sage: basis_B = B.basis()
sage: elt = basis_B[Composition([1,2,1])] + 4*basis_B[Composition([1,3])]; elt
B[1, 2, 1] + 4*B[1, 3]
sage: D(elt)
5*D{} + 5*D{1} + D{1, 3} + D{3}
sage: I(elt)
7/6*I[1, 1, 1, 1] + 2*I[1, 1, 2] + 3*I[1, 2, 1] + 4*I[1, 3]
```

As syntactic sugar, one can use the notation $D[i, \dots, 1]$ to construct elements of the basis; note that for the empty set one must use $D[[]]$ due to Python's syntax:

```
sage: D[[]] + D[2] + 2*D[1,2]
D{} + 2*D{1, 2} + D{2}
```

The same syntax works for the other bases:

```
sage: I[1,2,1] + 3*I[4] + 2*I[3,1]
I[1, 2, 1] + 2*I[3, 1] + 3*I[4]
```

class B (*alg*, *prefix*='B')

Bases: `sage.combinat.free_module.CombinatorialFreeModule`,
`sage.misc.bindable_class.BindableClass`

The subset basis of a descent algebra (indexed by compositions).

The subset basis $(B_S)_{S \subseteq \{1,2,\dots,n-1\}}$ of Σ_n is formed by

$$B_S = \sum_{T \subseteq S} D_T,$$

where $(D_S)_{S \subseteq \{1,2,\dots,n-1\}}$ is the *standard basis*. However it is more natural to index the subset basis by compositions of n under the bijection $\{i_1, i_2, \dots, i_k\} \mapsto (i_1, i_2 - i_1, i_3 - i_2, \dots, i_k - i_{k-1}, n - i_k)$ (where $i_1 < i_2 < \dots < i_k$), which is what Sage uses to index the basis.

The basis element B_p is denoted Ξ^p in [Schocker2004].

By using compositions of n , the product $B_p B_q$ becomes a sum over the non-negative-integer matrices M with row sum p and column sum q . The summand corresponding to M is B_c , where c is the composition obtained by reading M row-by-row from left-to-right and top-to-bottom and removing all zeroes. This multiplication rule is commonly called “Solomon’s Mackey formula”.

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: B = DA.B()
sage: list(B.basis())
[B[1, 1, 1, 1], B[1, 1, 2], B[1, 2, 1], B[1, 3],
 B[2, 1, 1], B[2, 2], B[3, 1], B[4]]
```

one_basis ()

Return the identity element which is the composition $[n]$, as per `AlgebrasWithBasis.ParentMethods.one_basis`.

EXAMPLES:

```
sage: DescentAlgebra(QQ, 4).B().one_basis()
[4]
sage: DescentAlgebra(QQ, 0).B().one_basis()
[]
sage: all( U * DescentAlgebra(QQ, 3).B().one() == U
....:      for U in DescentAlgebra(QQ, 3).B().basis() )
True
```

product_on_basis (*p*, *q*)

Return $B_p B_q$, where p and q are compositions of n .

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: B = DA.B()
sage: p = Composition([1,2,1])
sage: q = Composition([3,1])
sage: B.product_on_basis(p, q)
B[1, 1, 1, 1] + 2*B[1, 2, 1]
```

to_D_basis (*p*)

Return B_p as a linear combination of D -basis elements.

EXAMPLES:

```

sage: DA = DescentAlgebra(QQ, 4)
sage: B = DA.B()
sage: D = DA.D()
sage: list(map(D, B.basis())) # indirect doctest
[D{} + D{1} + D{1, 2} + D{1, 2, 3}
 + D{1, 3} + D{2} + D{2, 3} + D{3},
 D{} + D{1} + D{1, 2} + D{2},
 D{} + D{1} + D{1, 3} + D{3},
 D{} + D{1},
 D{} + D{2} + D{2, 3} + D{3},
 D{} + D{2},
 D{} + D{3},
 D{}]

```

to_I_basis (p)

Return B_p as a linear combination of I -basis elements.

This is done using the formula

$$B_p = \sum_{q \leq p} \frac{1}{\mathbf{k}!(q, p)} I_q,$$

where \leq is the refinement order and $\mathbf{k}!(q, p)$ is defined as follows: When $q \leq p$, we can write q as a concatenation $q_{(1)}q_{(2)} \cdots q_{(k)}$ with each $q_{(i)}$ being a composition of the i -th entry of p , and then we set $\mathbf{k}!(q, p)$ to be $l(q_{(1)})!l(q_{(2)})! \cdots l(q_{(k)})!$, where $l(r)$ denotes the number of parts of any composition r .

EXAMPLES:

```

sage: DA = DescentAlgebra(QQ, 4)
sage: B = DA.B()
sage: I = DA.I()
sage: list(map(I, B.basis())) # indirect doctest
[I[1, 1, 1, 1],
 1/2*I[1, 1, 1, 1] + I[1, 1, 2],
 1/2*I[1, 1, 1, 1] + I[1, 2, 1],
 1/6*I[1, 1, 1, 1] + 1/2*I[1, 1, 2] + 1/2*I[1, 2, 1] + I[1, 3],
 1/2*I[1, 1, 1, 1] + I[2, 1, 1],
 1/4*I[1, 1, 1, 1] + 1/2*I[1, 1, 2] + 1/2*I[2, 1, 1] + I[2, 2],
 1/6*I[1, 1, 1, 1] + 1/2*I[1, 2, 1] + 1/2*I[2, 1, 1] + I[3, 1],
 1/24*I[1, 1, 1, 1] + 1/6*I[1, 1, 2] + 1/6*I[1, 2, 1]
 + 1/2*I[1, 3] + 1/6*I[2, 1, 1] + 1/2*I[2, 2] + 1/2*I[3, 1] + I[4]]

```

to_nsym (p)

Return B_p as an element in $NSym$, the non-commutative symmetric functions.

This maps B_p to S_p where S denotes the Complete basis of $NSym$.

EXAMPLES:

```

sage: B = DescentAlgebra(QQ, 4).B()
sage: S = NonCommutativeSymmetricFunctions(QQ).Complete()
sage: list(map(S, B.basis())) # indirect doctest
[S[1, 1, 1, 1],
 S[1, 1, 2],
 S[1, 2, 1],
 S[1, 3],
 S[2, 1, 1],
 S[2, 2],

```

```
S[3, 1],
S[4]]
```

class DescentAlgebra . **D** (*alg*, *prefix*='D')

Bases: sage.combinat.free_module.CombinatorialFreeModule ,
sage.misc.bindable_class.BindableClass

The standard basis of a descent algebra.

This basis is indexed by $S \subseteq \{1, 2, \dots, n-1\}$, and the basis vector indexed by S is the sum of all permutations, taken in the symmetric group algebra RS_n , whose descent set is S . We denote this basis vector by D_S .

Occasionally this basis appears in literature but indexed by compositions of n rather than subsets of $\{1, 2, \dots, n-1\}$. The equivalence between these two indexings is owed to the bijection from the power set of $\{1, 2, \dots, n-1\}$ to the set of all compositions of n which sends every subset $\{i_1, i_2, \dots, i_k\}$ of $\{1, 2, \dots, n-1\}$ (with $i_1 < i_2 < \dots < i_k$) to the composition $(i_1, i_2 - i_1, \dots, i_k - i_{k-1}, n - i_k)$.

The basis element corresponding to a composition p (or to the subset of $\{1, 2, \dots, n-1\}$) is denoted Δ^p in [Schocker2004].

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: D = DA.D()
sage: list(D.basis())
[D{}, D{1}, D{2}, D{3}, D{1, 2}, D{1, 3}, D{2, 3}, D{1, 2, 3}]

sage: DA = DescentAlgebra(QQ, 0)
sage: D = DA.D()
sage: list(D.basis())
[D{}]
```

one_basis ()

Return the identity element, as per AlgebrasWithBasis.ParentMethods.one_basis .

EXAMPLES:

```
sage: DescentAlgebra(QQ, 4).D().one_basis()
()
sage: DescentAlgebra(QQ, 0).D().one_basis()
()

sage: all( U * DescentAlgebra(QQ, 3).D().one() == U
....:      for U in DescentAlgebra(QQ, 3).D().basis() )
True
```

product_on_basis (S, T)

Return $D_S D_T$, where S and T are subsets of $[n-1]$.

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: D = DA.D()
sage: D.product_on_basis((1, 3), (2,))
D{} + D{1} + D{1, 2} + 2*D{1, 2, 3} + D{1, 3} + D{2} + D{2, 3} + D{3}
```

to_B_basis (S)

Return D_S as a linear combination of B_p -basis elements.

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: D = DA.D()
sage: B = DA.B()
sage: list(map(B, D.basis())) # indirect doctest
[B[4],
 B[1, 3] - B[4],
 B[2, 2] - B[4],
 B[3, 1] - B[4],
 B[1, 1, 2] - B[1, 3] - B[2, 2] + B[4],
 B[1, 2, 1] - B[1, 3] - B[3, 1] + B[4],
 B[2, 1, 1] - B[2, 2] - B[3, 1] + B[4],
 B[1, 1, 1, 1] - B[1, 1, 2] - B[1, 2, 1] + B[1, 3]
 - B[2, 1, 1] + B[2, 2] + B[3, 1] - B[4]]
```

to_symmetric_group_algebra_on_basis (S)

Return D_S as a linear combination of basis elements in the symmetric group algebra.

EXAMPLES:

```
sage: D = DescentAlgebra(QQ, 4).D()
sage: [D.to_symmetric_group_algebra_on_basis(tuple(b))
....:  for b in Subsets(3)]
[[1, 2, 3, 4],
 [2, 1, 3, 4] + [3, 1, 2, 4] + [4, 1, 2, 3],
 [1, 3, 2, 4] + [1, 4, 2, 3] + [2, 3, 1, 4]
 + [2, 4, 1, 3] + [3, 4, 1, 2],
 [1, 2, 4, 3] + [1, 3, 4, 2] + [2, 3, 4, 1],
 [3, 2, 1, 4] + [4, 2, 1, 3] + [4, 3, 1, 2],
 [2, 1, 4, 3] + [3, 1, 4, 2] + [3, 2, 4, 1]
 + [4, 1, 3, 2] + [4, 2, 3, 1],
 [1, 4, 3, 2] + [2, 4, 3, 1] + [3, 4, 2, 1],
 [4, 3, 2, 1]]
```

class DescentAlgebra. **I** (alg, prefix='I')

Bases: sage.combinat.free_module.CombinatorialFreeModule, sage.misc.bindable_class.BindableClass

The idempotent basis of a descent algebra.

The idempotent basis $(I_p)_{p \models n}$ is a basis for Σ_n whenever the ground ring is a \mathbf{Q} -algebra. One way to compute it is using the formula (Theorem 3.3 in [GR1989])

$$I_p = \sum_{q \leq p} \frac{(-1)^{l(q)-l(p)}}{\mathbf{k}(q,p)} B_q,$$

where \leq is the refinement order and $l(r)$ denotes the number of parts of any composition r , and where $\mathbf{k}(q,p)$ is defined as follows: When $q \leq p$, we can write q as a concatenation $q_{(1)}q_{(2)} \cdots q_{(k)}$ with each $q_{(i)}$ being a composition of the i -th entry of p , and then we set $\mathbf{k}(q,p)$ to be the product $l(q_{(1)})l(q_{(2)}) \cdots l(q_{(k)})$.

Let $\lambda(p)$ denote the partition obtained from a composition p by sorting. This basis is called the idempotent basis since for any q such that $\lambda(p) = \lambda(q)$, we have:

$$I_p I_q = s(\lambda) I_p$$

where λ denotes $\lambda(p) = \lambda(q)$, and where $s(\lambda)$ is the stabilizer of λ in S_n . (This is part of Theorem 4.2 in [GR1989].)

It is also straightforward to compute the idempotents E_λ for the symmetric group algebra by the formula (Theorem 3.2 in [GR1989]):

$$E_\lambda = \frac{1}{k!} \sum_{\lambda(p)=\lambda} I_p.$$

Note: The basis elements are not orthogonal idempotents.

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: I = DA.I()
sage: list(I.basis())
[I[1, 1, 1, 1], I[1, 1, 2], I[1, 2, 1], I[1, 3], I[2, 1, 1], I[2, 2], I[3,
↪ 1], I[4]]
```

idempotent (*la*)

Return the idempotent corresponding to the partition *la* of *n*.

EXAMPLES:

```
sage: I = DescentAlgebra(QQ, 4).I()
sage: E = I.idempotent([3,1]); E
1/2*I[1, 3] + 1/2*I[3, 1]
sage: E*E == E
True
sage: E2 = I.idempotent([2,1,1]); E2
1/6*I[1, 1, 2] + 1/6*I[1, 2, 1] + 1/6*I[2, 1, 1]
sage: E2*E2 == E2
True
sage: E*E2 == I.zero()
True
```

one ()

Return the identity element, which is $B_{[n]}$, in the *I* basis.

EXAMPLES:

```
sage: DescentAlgebra(QQ, 4).I().one()
1/24*I[1, 1, 1, 1] + 1/6*I[1, 1, 2] + 1/6*I[1, 2, 1]
+ 1/2*I[1, 3] + 1/6*I[2, 1, 1] + 1/2*I[2, 2]
+ 1/2*I[3, 1] + I[4]
sage: DescentAlgebra(QQ, 0).I().one()
I[]
```

one_basis ()

The element 1 is not (generally) a basis vector in the *I* basis, thus this returns a `TypeError`.

EXAMPLES:

```
sage: DescentAlgebra(QQ, 4).I().one_basis()
Traceback (most recent call last):
...
TypeError: 1 is not a basis element in the I basis.
```

product_on_basis (*p*, *q*)

Return $I_p I_q$, where *p* and *q* are compositions of *n*.

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: I = DA.I()
sage: p = Composition([1,2,1])
sage: q = Composition([3,1])
sage: I.product_on_basis(p, q)
0
sage: I.product_on_basis(p, p)
2*I[1, 2, 1]
```

to_B_basis (*p*)

Return I_p as a linear combination of B -basis elements.

This is computed using the formula (Theorem 3.3 in [GR1989])

$$I_p = \sum_{q \leq p} \frac{(-1)^{l(q)-l(p)}}{\mathbf{k}(q,p)} B_q,$$

where \leq is the refinement order and $l(r)$ denotes the number of parts of any composition r , and where $\mathbf{k}(q,p)$ is defined as follows: When $q \leq p$, we can write q as a concatenation $q_{(1)}q_{(2)} \cdots q_{(k)}$ with each $q_{(i)}$ being a composition of the i -th entry of p , and then we set $\mathbf{k}(q,p)$ to be $l(q_{(1)})l(q_{(2)}) \cdots l(q_{(k)})$.

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: B = DA.B()
sage: I = DA.I()
sage: list(map(B, I.basis())) # indirect doctest
[B[1, 1, 1, 1],
 -1/2*B[1, 1, 1, 1] + B[1, 1, 2],
 -1/2*B[1, 1, 1, 1] + B[1, 2, 1],
 1/3*B[1, 1, 1, 1] - 1/2*B[1, 1, 2] - 1/2*B[1, 2, 1] + B[1, 3],
 -1/2*B[1, 1, 1, 1] + B[2, 1, 1],
 1/4*B[1, 1, 1, 1] - 1/2*B[1, 1, 2] - 1/2*B[2, 1, 1] + B[2, 2],
 1/3*B[1, 1, 1, 1] - 1/2*B[1, 2, 1] - 1/2*B[2, 1, 1] + B[3, 1],
 -1/4*B[1, 1, 1, 1] + 1/3*B[1, 1, 2] + 1/3*B[1, 2, 1]
 - 1/2*B[1, 3] + 1/3*B[2, 1, 1] - 1/2*B[2, 2]
 - 1/2*B[3, 1] + B[4]]
```

DescentAlgebra.a_realization ()

Return a particular realization of self (the B -basis).

EXAMPLES:

```
sage: DA = DescentAlgebra(QQ, 4)
sage: DA.a_realization()
Descent algebra of 4 over Rational Field in the subset basis
```

class sage.combinat.descent_algebra. **DescentAlgebraBases** (*base*)

Bases: sage.categories.realizations.Category_realization_of_parent

The category of bases of a descent algebra.

class **ElementMethods**

to_symmetric_group_algebra ()

Return self in the symmetric group algebra.

EXAMPLES:

```

sage: B = DescentAlgebra(QQ, 4).B()
sage: B[1,3].to_symmetric_group_algebra()
[1, 2, 3, 4] + [2, 1, 3, 4] + [3, 1, 2, 4] + [4, 1, 2, 3]
sage: I = DescentAlgebra(QQ, 4).I()
sage: elt = I(B[1,3])
sage: elt.to_symmetric_group_algebra()
[1, 2, 3, 4] + [2, 1, 3, 4] + [3, 1, 2, 4] + [4, 1, 2, 3]

```

class DescentAlgebraBases. **ParentMethods**

is_commutative ()

Return whether this descent algebra is commutative.

EXAMPLES:

```

sage: B = DescentAlgebra(QQ, 4).B()
sage: B.is_commutative()
False
sage: B = DescentAlgebra(QQ, 1).B()
sage: B.is_commutative()
True

```

is_field (*proof=True*)

Return whether this descent algebra is a field.

EXAMPLES:

```

sage: B = DescentAlgebra(QQ, 4).B()
sage: B.is_field()
False
sage: B = DescentAlgebra(QQ, 1).B()
sage: B.is_field()
True

```

to_symmetric_group_algebra ()

Morphism from self to the symmetric group algebra.

EXAMPLES:

```

sage: D = DescentAlgebra(QQ, 4).D()
sage: D.to_symmetric_group_algebra(D[1,3])
[2, 1, 4, 3] + [3, 1, 4, 2] + [3, 2, 4, 1] + [4, 1, 3, 2] + [4, 2, 3, 1]
sage: B = DescentAlgebra(QQ, 4).B()
sage: B.to_symmetric_group_algebra(B[1,2,1])
[1, 2, 3, 4] + [1, 2, 4, 3] + [1, 3, 4, 2] + [2, 1, 3, 4]
+ [2, 1, 4, 3] + [2, 3, 4, 1] + [3, 1, 2, 4] + [3, 1, 4, 2]
+ [3, 2, 4, 1] + [4, 1, 2, 3] + [4, 1, 3, 2] + [4, 2, 3, 1]

```

to_symmetric_group_algebra_on_basis (*S*)

Return the basis element index by *S* as a linear combination of basis elements in the symmetric group algebra.

EXAMPLES:

```

sage: B = DescentAlgebra(QQ, 3).B()
sage: [B.to_symmetric_group_algebra_on_basis(c)
....: for c in Compositions(3)]
[[1, 2, 3] + [1, 3, 2] + [2, 1, 3]

```



```

+ [2, 3, 1] + [3, 1, 2] + [3, 2, 1],
[1, 2, 3] + [2, 1, 3] + [3, 1, 2],
[1, 2, 3] + [1, 3, 2] + [2, 3, 1],
[1, 2, 3]]
sage: I = DescentAlgebra(QQ, 3).I()
sage: [I.to_symmetric_group_algebra_on_basis(c)
....:  for c in Compositions(3)]
[[[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1]
+ [3, 1, 2] + [3, 2, 1],
1/2*[1, 2, 3] - 1/2*[1, 3, 2] + 1/2*[2, 1, 3]
- 1/2*[2, 3, 1] + 1/2*[3, 1, 2] - 1/2*[3, 2, 1],
1/2*[1, 2, 3] + 1/2*[1, 3, 2] - 1/2*[2, 1, 3]
+ 1/2*[2, 3, 1] - 1/2*[3, 1, 2] - 1/2*[3, 2, 1],
1/3*[1, 2, 3] - 1/6*[1, 3, 2] - 1/6*[2, 1, 3]
- 1/6*[2, 3, 1] - 1/6*[3, 1, 2] + 1/3*[3, 2, 1]]

```

`DescentAlgebraBases.super_categories()`

The super categories of self.

EXAMPLES:

```

sage: from sage.combinat.descent_algebra import DescentAlgebraBases
sage: DA = DescentAlgebra(QQ, 4)
sage: bases = DescentAlgebraBases(DA)
sage: bases.super_categories()
[Category of finite dimensional algebras with basis over Rational Field,
Category of realizations of Descent algebra of 4 over Rational Field]

```

4.6 Hall Algebras

AUTHORS:

- Travis Scrimshaw (2013-10-17): Initial version

class `sage.algebras.hall_algebra.HallAlgebra` (*base_ring*, *q*, *prefix='H'*)

Bases: `sage.combinat.free_module.CombinatorialFreeModule`

The (classical) Hall algebra.

The (classical) Hall algebra over a commutative ring R with a parameter $q \in R$ is defined to be the free R -module with basis (I_λ) , where λ runs over all integer partitions. The algebra structure is given by a product defined by

$$I_\mu \cdot I_\lambda = \sum_{\nu} P_{\mu,\lambda}^{\nu}(q) I_\nu,$$

where $P_{\mu,\lambda}^{\nu}$ is a Hall polynomial (see `hall_polynomial()`). The unity of this algebra is I_\emptyset .

The (classical) Hall algebra is also known as the Hall-Steinitz algebra.

We can define an R -algebra isomorphism Φ from the R -algebra of symmetric functions (see `SymmetricFunctions`) to the (classical) Hall algebra by sending the r -th elementary symmetric function e_r to $q^{r(r-1)/2} I_{(1^r)}$ for every positive integer r . This isomorphism used to transport the Hopf algebra structure from the R -algebra of symmetric functions to the Hall algebra, thus making the latter a connected graded Hopf algebra. If λ is a partition, then the preimage of the basis element I_λ under this isomorphism is $q^{n(\lambda)} P_\lambda(x; q^{-1})$, where P_λ denotes the λ -th Hall-Littlewood P -function, and where $n(\lambda) = \sum_i (i-1)\lambda_i$.

See section 2.3 in [Sch2006], and sections II.2 and III.3 in [Macdonald1995] (where our I_λ is called u_λ).

EXAMPLES:

```
sage: R.<q> = ZZ[]
sage: H = HallAlgebra(R, q)
sage: H[2,1]*H[1,1]
H[3, 2] + (q+1)*H[3, 1, 1] + (q^2+q)*H[2, 2, 1] + (q^4+q^3+q^2)*H[2, 1, 1, 1]
sage: H[2]*H[2,1]
H[4, 1] + q*H[3, 2] + (q^2-1)*H[3, 1, 1] + (q^3+q^2)*H[2, 2, 1]
sage: H[3]*H[1,1]
H[4, 1] + q^2*H[3, 1, 1]
sage: H[3]*H[2,1]
H[5, 1] + q*H[4, 2] + (q^2-1)*H[4, 1, 1] + q^3*H[3, 2, 1]
```

We can rewrite the Hall algebra in terms of monomials of the elements $I_{(1^r)}$:

```
sage: I = H.monomial_basis()
sage: H(I[2,1,1])
H[3, 1] + (q+1)*H[2, 2] + (2*q^2+2*q+1)*H[2, 1, 1]
+ (q^5+2*q^4+3*q^3+3*q^2+2*q+1)*H[1, 1, 1, 1]
sage: I(H[2,1,1])
I[3, 1] + (-q^3-q^2-q-1)*I[4]
```

The isomorphism between the Hall algebra and the symmetric functions described above is implemented as a coercion:

```
sage: R = PolynomialRing(ZZ, 'q').fraction_field()
sage: q = R.gen()
sage: H = HallAlgebra(R, q)
sage: e = SymmetricFunctions(R).e()
sage: e(H[1,1,1])
1/q^3*e[3]
```

We can also do computations with any special value of q , such as 0 or 1 or (most commonly) a prime power. Here is an example using a prime:

```
sage: H = HallAlgebra(ZZ, 2)
sage: H[2,1]*H[1,1]
H[3, 2] + 3*H[3, 1, 1] + 6*H[2, 2, 1] + 28*H[2, 1, 1, 1]
sage: H[3,1]*H[2]
H[5, 1] + H[4, 2] + 6*H[3, 3] + 3*H[4, 1, 1] + 8*H[3, 2, 1]
sage: H[2,1,1]*H[3,1]
H[5, 2, 1] + 2*H[4, 3, 1] + 6*H[4, 2, 2] + 7*H[5, 1, 1, 1]
+ 19*H[4, 2, 1, 1] + 24*H[3, 3, 1, 1] + 48*H[3, 2, 2, 1]
+ 105*H[4, 1, 1, 1, 1] + 224*H[3, 2, 1, 1, 1]
sage: I = H.monomial_basis()
sage: H(I[2,1,1])
H[3, 1] + 3*H[2, 2] + 13*H[2, 1, 1] + 105*H[1, 1, 1, 1]
sage: I(H[2,1,1])
I[3, 1] - 15*I[4]
```

If q is set to 1, the coercion to the symmetric functions sends I_λ to m_λ :

```
sage: H = HallAlgebra(QQ, 1)
sage: H[2,1] * H[2,1]
H[4, 2] + 2*H[3, 3] + 2*H[4, 1, 1] + 2*H[3, 2, 1] + 6*H[2, 2, 2] + 4*H[2, 2, 1, 1]
sage: m = SymmetricFunctions(QQ).m()
sage: m[2,1] * m[2,1]
4*m[2, 2, 1, 1] + 6*m[2, 2, 2] + 2*m[3, 2, 1] + 2*m[3, 3] + 2*m[4, 1, 1] + m[4, 2]
```

```
sage: m(H[3,1])
m[3, 1]
```

We can set q to 0 (but should keep in mind that we don't get the Schur functions this way):

```
sage: H = HallAlgebra(QQ, 0)
sage: H[2,1] * H[2,1]
H[4, 2] + H[3, 3] + H[4, 1, 1] - H[3, 2, 1] - H[3, 1, 1, 1]
```

class Element (M, x)

Bases: `sage.combinat.free_module.CombinatorialFreeModuleElement`

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

scalar (y)

Return the scalar product of `self` and y .

The scalar product is given by

$$(I_\lambda, I_\mu) = \delta_{\lambda, \mu} \frac{1}{a_\lambda},$$

where a_λ is given by

$$a_\lambda = q^{|\lambda|+2n(\lambda)} \prod_k \prod_{i=1}^{l_k} (1 - q^{-i})$$

where $n(\lambda) = \sum_i (i-1)\lambda_i$ and $\lambda = (1^{l_1}, 2^{l_2}, \dots, m^{l_m})$.

Note that a_λ can be interpreted as the number of automorphisms of a certain object in a category corresponding to λ . See Lemma 2.8 in [Sch2006] for details.

EXAMPLES:

```
sage: R.<q> = ZZ[]
sage: H = HallAlgebra(R, q)
sage: H[1].scalar(H[1])
1/(q - 1)
sage: H[2].scalar(H[2])
1/(q^2 - q)
sage: H[2,1].scalar(H[2,1])
1/(q^5 - 2*q^4 + q^3)
sage: H[1,1,1,1].scalar(H[1,1,1,1])
1/(q^16 - q^15 - q^14 + 2*q^11 - q^8 - q^7 + q^6)
sage: H.an_element().scalar(H.an_element())
(4*q^2 + 9)/(q^2 - q)
```

`HallAlgebra. antipode_on_basis` (la)

Return the antipode of the basis element indexed by la .

EXAMPLES:

```
sage: R = PolynomialRing(ZZ, 'q').fraction_field()
sage: q = R.gen()
sage: H = HallAlgebra(R, q)
sage: H.antipode_on_basis(Partition([1,1]))
1/q*H[2] + 1/q*H[1, 1]
sage: H.antipode_on_basis(Partition([2]))
```

```

-1/q*H[2] + ((q^2-1)/q)*H[1, 1]

sage: R.<q> = LaurentPolynomialRing(ZZ)
sage: H = HallAlgebra(R, q)
sage: H.antipode_on_basis(Partition([1,1]))
(q^-1)*H[2] + (q^-1)*H[1, 1]
sage: H.antipode_on_basis(Partition([2]))
-(q^-1)*H[2] - (q^-1-q)*H[1, 1]

```

HallAlgebra. **coproduct_on_basis** (*la*)

Return the coproduct of the basis element indexed by *la*.

EXAMPLES:

```

sage: R = PolynomialRing(ZZ, 'q').fraction_field()
sage: q = R.gen()
sage: H = HallAlgebra(R, q)
sage: H.coproduct_on_basis(Partition([1,1]))
H[] # H[1, 1] + 1/q*H[1] # H[1] + H[1, 1] # H[]
sage: H.coproduct_on_basis(Partition([2]))
H[] # H[2] + ((q-1)/q)*H[1] # H[1] + H[2] # H[]
sage: H.coproduct_on_basis(Partition([2,1]))
H[] # H[2, 1] + ((q^2-1)/q^2)*H[1] # H[1, 1] + 1/q*H[1] # H[2]
+ ((q^2-1)/q^2)*H[1, 1] # H[1] + 1/q*H[2] # H[1] + H[2, 1] # H[]

sage: R.<q> = LaurentPolynomialRing(ZZ)
sage: H = HallAlgebra(R, q)
sage: H.coproduct_on_basis(Partition([2]))
H[] # H[2] - (q^-1-1)*H[1] # H[1] + H[2] # H[]
sage: H.coproduct_on_basis(Partition([2,1]))
H[] # H[2, 1] - (q^-2-1)*H[1] # H[1, 1] + (q^-1)*H[1] # H[2]
- (q^-2-1)*H[1, 1] # H[1] + (q^-1)*H[2] # H[1] + H[2, 1] # H[]

```

HallAlgebra. **counit** (*x*)

Return the counit of the element *x*.

EXAMPLES:

```

sage: R = PolynomialRing(ZZ, 'q').fraction_field()
sage: q = R.gen()
sage: H = HallAlgebra(R, q)
sage: H.counit(H.an_element())
2

```

HallAlgebra. **monomial_basis** ()

Return the basis of the Hall algebra given by monomials in the $I_{(1^r)}$.

EXAMPLES:

```

sage: R.<q> = ZZ[]
sage: H = HallAlgebra(R, q)
sage: H.monomial_basis()
Hall algebra with q=q over Univariate Polynomial Ring in q over
Integer Ring in the monomial basis

```

HallAlgebra. **one_basis** ()

Return the index of the basis element 1.

EXAMPLES:

```
sage: R.<q> = ZZ[]
sage: H = HallAlgebra(R, q)
sage: H.one_basis()
[]
```

`HallAlgebra.product_on_basis (mu, la)`

Return the product of the two basis elements indexed by `mu` and `la`.

EXAMPLES:

```
sage: R.<q> = ZZ[]
sage: H = HallAlgebra(R, q)
sage: H.product_on_basis(Partition([1,1]), Partition([1]))
H[2, 1] + (q^2+q+1)*H[1, 1, 1]
sage: H.product_on_basis(Partition([2,1]), Partition([1,1]))
H[3, 2] + (q+1)*H[3, 1, 1] + (q^2+q)*H[2, 2, 1] + (q^4+q^3+q^2)*H[2, 1, 1, 1]
sage: H.product_on_basis(Partition([3,2]), Partition([2,1]))
H[5, 3] + (q+1)*H[4, 4] + q*H[5, 2, 1] + (2*q^2-1)*H[4, 3, 1]
+ (q^3+q^2)*H[4, 2, 2] + (q^4+q^3)*H[3, 3, 2]
+ (q^4-q^2)*H[4, 2, 1, 1] + (q^5+q^4-q^3-q^2)*H[3, 3, 1, 1]
+ (q^6+q^5)*H[3, 2, 2, 1]
sage: H.product_on_basis(Partition([3,1,1]), Partition([2,1]))
H[5, 2, 1] + q*H[4, 3, 1] + (q^2-1)*H[4, 2, 2]
+ (q^3+q^2)*H[3, 3, 2] + (q^2+q+1)*H[5, 1, 1, 1]
+ (2*q^3+q^2-q-1)*H[4, 2, 1, 1] + (q^4+2*q^3+q^2)*H[3, 3, 1, 1]
+ (q^5+q^4)*H[3, 2, 2, 1] + (q^6+q^5+q^4-q^2-q-1)*H[4, 1, 1, 1, 1]
+ (q^7+q^6+q^5)*H[3, 2, 1, 1, 1]
```

class `sage.algebras.hall_algebra.HallAlgebraMonomials (base_ring, q, prefix='I')`

Bases: `sage.combinat.free_module.CombinatorialFreeModule`

The classical Hall algebra given in terms of monomials in the $I_{(1^r)}$.

We first associate a monomial $I_{(1^{r_1})}I_{(1^{r_2})}\cdots I_{(1^{r_k})}$ with the composition (r_1, r_2, \dots, r_k) . However since $I_{(1^r)}$ commutes with $I_{(1^s)}$, the basis is indexed by partitions.

EXAMPLES:

We use the fraction field of $\mathbf{Z}[q]$ for our initial example:

```
sage: R = PolynomialRing(ZZ, 'q').fraction_field()
sage: q = R.gen()
sage: H = HallAlgebra(R, q)
sage: I = H.monomial_basis()
```

We check that the basis conversions are mutually inverse:

```
sage: all(H(I(H[p])) == H[p] for i in range(7) for p in Partitions(i))
True
sage: all(I(H(I[p])) == I[p] for i in range(7) for p in Partitions(i))
True
```

Since Laurent polynomials are sufficient, we run the same check with the Laurent polynomial ring $\mathbf{Z}[q, q^{-1}]$:

```
sage: R.<q> = LaurentPolynomialRing(ZZ)
sage: H = HallAlgebra(R, q)
sage: I = H.monomial_basis()
sage: all(H(I(H[p])) == H[p] for i in range(6) for p in Partitions(i)) # long time
True
```

```
sage: all(I(H(I[p])) == I[p] for i in range(6) for p in Partitions(i)) # long time
True
```

We can also convert to the symmetric functions. The natural basis corresponds to the Hall-Littlewood basis (up to a renormalization and an inversion of the q parameter), and this basis corresponds to the elementary basis (up to a renormalization):

```
sage: Sym = SymmetricFunctions(R)
sage: e = Sym.e()
sage: e(I[2,1])
(q^-1)*e[2, 1]
sage: e(I[4,2,2,1])
(q^-8)*e[4, 2, 2, 1]
sage: HLP = Sym.hall_littlewood(q).P()
sage: H(I[2,1])
H[2, 1] + (1+q+q^2)*H[1, 1, 1]
sage: HLP(e[2,1])
(1+q+q^2)*HLP[1, 1, 1] + HLP[2, 1]
sage: all( e(H[lam]) == q**sum([i * x for i, x in enumerate(lam)])
.....:      * e(HLP[lam]).map_coefficients(lambda p: p(q**(-1)))
.....:      for lam in Partitions(4) )
True
```

We can also do computations using a prime power:

```
sage: H = HallAlgebra(ZZ, 3)
sage: I = H.monomial_basis()
sage: i_elt = I[2,1]*I[1,1]; i_elt
I[2, 1, 1, 1]
sage: H(i_elt)
H[4, 1] + 7*H[3, 2] + 37*H[3, 1, 1] + 136*H[2, 2, 1]
+ 1495*H[2, 1, 1, 1] + 62920*H[1, 1, 1, 1, 1]
```

class Element (M, x)

Bases: sage.combinat.free_module.CombinatorialFreeModuleElement

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

scalar (y)

Return the scalar product of `self` and y .

The scalar product is computed by converting into the natural basis.

EXAMPLES:

```
sage: R.<q> = ZZ[]
sage: I = HallAlgebra(R, q).monomial_basis()
sage: I[1].scalar(I[1])
1/(q - 1)
sage: I[2].scalar(I[2])
1/(q^4 - q^3 - q^2 + q)
sage: I[2,1].scalar(I[2,1])
(2*q + 1)/(q^6 - 2*q^5 + 2*q^3 - q^2)
sage: I[1,1,1,1].scalar(I[1,1,1,1])
24/(q^4 - 4*q^3 + 6*q^2 - 4*q + 1)
sage: I.an_element().scalar(I.an_element())
(4*q^4 - 4*q^2 + 9)/(q^4 - q^3 - q^2 + q)
```

HallAlgebraMonomials.**antipode_on_basis** (*a*)

Return the antipode of the basis element indexed by *a* .

EXAMPLES:

```
sage: R = PolynomialRing(ZZ, 'q').fraction_field()
sage: q = R.gen()
sage: I = HallAlgebra(R, q).monomial_basis()
sage: I.antipode_on_basis(Partition([1]))
-I[1]
sage: I.antipode_on_basis(Partition([2]))
1/q*I[1, 1] - I[2]
sage: I.antipode_on_basis(Partition([2,1]))
-1/q*I[1, 1, 1] + I[2, 1]

sage: R.<q> = LaurentPolynomialRing(ZZ)
sage: I = HallAlgebra(R, q).monomial_basis()
sage: I.antipode_on_basis(Partition([2,1]))
-(q^-1)*I[1, 1, 1] + I[2, 1]
```

HallAlgebraMonomials.**coproduct_on_basis** (*a*)

Return the coproduct of the basis element indexed by *a* .

EXAMPLES:

```
sage: R = PolynomialRing(ZZ, 'q').fraction_field()
sage: q = R.gen()
sage: I = HallAlgebra(R, q).monomial_basis()
sage: I.coproduct_on_basis(Partition([1]))
I[] # I[1] + I[1] # I[]
sage: I.coproduct_on_basis(Partition([2]))
I[] # I[2] + 1/q*I[1] # I[1] + I[2] # I[]
sage: I.coproduct_on_basis(Partition([2,1]))
I[] # I[2, 1] + 1/q*I[1] # I[1, 1] + I[1] # I[2]
+ 1/q*I[1, 1] # I[1] + I[2] # I[1] + I[2, 1] # I[]

sage: R.<q> = LaurentPolynomialRing(ZZ)
sage: I = HallAlgebra(R, q).monomial_basis()
sage: I.coproduct_on_basis(Partition([2,1]))
I[] # I[2, 1] + (q^-1)*I[1] # I[1, 1] + I[1] # I[2]
+ (q^-1)*I[1, 1] # I[1] + I[2] # I[1] + I[2, 1] # I[]
```

HallAlgebraMonomials.**counit** (*x*)

Return the counit of the element *x* .

EXAMPLES:

```
sage: R = PolynomialRing(ZZ, 'q').fraction_field()
sage: q = R.gen()
sage: I = HallAlgebra(R, q).monomial_basis()
sage: I.counit(I.an_element())
2
```

HallAlgebraMonomials.**one_basis** ()

Return the index of the basis element 1.

EXAMPLES:

```
sage: R.<q> = ZZ[]
sage: I = HallAlgebra(R, q).monomial_basis()
```

```
sage: I.one_basis()
[]
```

`HallAlgebraMonomials.product_on_basis (a, b)`

Return the product of the two basis elements indexed by `a` and `b`.

EXAMPLES:

```
sage: R.<q> = ZZ[]
sage: I = HallAlgebra(R, q).monomial_basis()
sage: I.product_on_basis(Partition([4,2,1]), Partition([3,2,1]))
I[4, 3, 2, 2, 1, 1]
```

`sage.algebras.hall_algebra.transpose_cmp (x, y)`

Compare partitions `x` and `y` in transpose dominance order.

We say partitions μ and λ satisfy $\mu \prec \lambda$ in transpose dominance order if for all $i \geq 1$ we have:

$$l_1 + 2l_2 + \cdots + (i-1)l_{i-1} + i(l_i + l_{i+1} + \cdots) \leq m_1 + 2m_2 + \cdots + (i-1)m_{i-1} + i(m_i + m_{i+1} + \cdots),$$

where l_k denotes the number of appearances of k in λ , and m_k denotes the number of appearances of k in μ .

Equivalently, $\mu \prec \lambda$ if the conjugate of the partition μ dominates the conjugate of the partition λ .

Since this is a partial ordering, we fallback to lex ordering $\mu <_L \lambda$ if we cannot compare in the transpose order.

EXAMPLES:

```
sage: from sage.algebras.hall_algebra import transpose_cmp
sage: transpose_cmp(Partition([4,3,1]), Partition([3,2,2,1]))
-1
sage: transpose_cmp(Partition([2,2,1]), Partition([3,2]))
1
sage: transpose_cmp(Partition([4,1,1]), Partition([4,1,1]))
0
```

4.7 Iwahori-Hecke Algebras

AUTHORS:

- Daniel Bump, Nicolas Thiery (2010): Initial version
- Brant Jones, Travis Scrimshaw, Andrew Mathas (2013): Moved into the category framework and implemented the Kazhdan-Lusztig C and C' bases

class `sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra (W, q1, q2, base_ring)`

Bases: `sage.structure.parent.Parent`, `sage.structure.unique_representation.UniqueRepresentation`

The Iwahori-Hecke algebra of the Coxeter group W with the specified parameters.

INPUT:

- W – a Coxeter group or Cartan type
- $q1$ – a parameter

OPTIONAL ARGUMENTS:

- `q2` – (default `-1`) another parameter
- `base_ring` – (default `q1.parent()`) a ring containing `q1` and `q2`

The Iwahori-Hecke algebra [Iwa1964] is a deformation of the group algebra of a Weyl group or, more generally, a Coxeter group. These algebras are defined by generators and relations and they depend on a deformation parameter q . Taking $q = 1$, as in the following example, gives a ring isomorphic to the group algebra of the corresponding Coxeter group.

Let (W, S) be a Coxeter system and let R be a commutative ring containing elements q_1 and q_2 . Then the *Iwahori-Hecke algebra* $H = H_{q_1, q_2}(W, S)$ of (W, S) with parameters q_1 and q_2 is the unital associative algebra with generators $\{T_s \mid s \in S\}$ and relations:

$$(T_s - q_1)(T_s - q_2) = 0$$

$$T_r T_s T_r \cdots = T_s T_r T_s \cdots,$$

where the number of terms on either side of the second relations (the braid relations) is the order of rs in the Coxeter group W , for $r, s \in S$.

Iwahori-Hecke algebras are fundamental in many areas of mathematics, ranging from the representation theory of Lie groups and quantum groups, to knot theory and statistical mechanics. For more information see, for example, [KL79], [HKP2010], [Jon1987] and [Wikipedia article Iwahori-Hecke_algebra](#).

Bases

A reduced expression for an element $w \in W$ is any minimal length word $w = s_1 \cdots s_k$, with $s_i \in S$. If $w = s_1 \cdots s_k$ is a reduced expression for w then Matsumoto's Monoid Lemma implies that $T_w = T_{s_1} \cdots T_{s_k}$ depends on w and not on the choice of reduced expressions. Moreover, $\{T_w \mid w \in W\}$ is a basis for the Iwahori-Hecke algebra H and

$$T_s T_w = \begin{cases} T_{sw}, & \text{if } \ell(sw) = \ell(w) + 1, \\ (q_1 + q_2)T_w - q_1 q_2 T_{sw}, & \text{if } \ell(sw) = \ell(w) - 1. \end{cases}$$

The T -basis of H is implemented for any choice of parameters `q_1` and `q_2`:

```
sage: R.<u,v> = LaurentPolynomialRing(ZZ, 2)
sage: H = IwahoriHeckeAlgebra('A3', u,v)
sage: T = H.T()
sage: T[1]
T[1]
sage: T[1, 2, 1] + T[2]
T[1, 2, 1] + T[2]
sage: T[1] * T[1, 2, 1]
(u+v)*T[1, 2, 1] + (-u*v)*T[2, 1]
sage: T[1]^(-1)
(-u^(-1)*v^(-1))*T[1] + (v^(-1)+u^(-1))
```

Working over the Laurent polynomial ring $Z[q^{\pm 1/2}]$ Kazhdan and Lusztig proved that there exist two distinguished bases $\{C'_w \mid w \in W\}$ and $\{C_w \mid w \in W\}$ of H which are uniquely determined by the properties that they are invariant under the bar involution on H and have triangular transitions matrices with polynomial entries of a certain form with the T -basis; see [KL79] for a precise statement.

It turns out that the Kazhdan-Lusztig bases can be defined (by specialization) in H whenever $-q_1 q_2$ is a square in the base ring. The Kazhdan-Lusztig bases are implemented inside H whenever $-q_1 q_2$ has a square root:

```
sage: H = IwahoriHeckeAlgebra('A3', u^2, -v^2)
sage: T=H.T(); Cp= H.Cp(); C=H.C()
```

```

sage: T(Cp[1])
(u^-1*v^-1)*T[1] + (u^-1*v)
sage: T(C[1])
(u^-1*v^-1)*T[1] + (-u*v^-1)
sage: Cp(C[1])
doctest:...: DeprecationWarning: the 'cmp' keyword is deprecated, use 'key'
→instead
See http://trac.sagemath.org/21043 for details.
Cp[1] + (-u*v^-1-u^-1*v)
sage: elt = Cp[2]*Cp[3]+C[1]; elt
Cp[2,3] + Cp[1] + (-u*v^-1-u^-1*v)
sage: c = C(elt); c
C[2,3] + C[1] + (u*v^-1+u^-1*v)*C[2] + (u*v^-1+u^-1*v)*C[3] + (u^2*v^-2+2+u^-2*v^
→2)
sage: t = T(c); t
(u^-2*v^-2)*T[2,3] + (u^-1*v^-1)*T[1] + (u^-2)*T[2] + (u^-2)*T[3] + (-u*v^-1+u^-
→2*v^2)
sage: Cp(t)
Cp[2,3] + Cp[1] + (-u*v^-1-u^-1*v)
sage: Cp(c)
Cp[2,3] + Cp[1] + (-u*v^-1-u^-1*v)

```

The conversions to and from the Kazhdan-Lusztig bases are done behind the scenes whenever the Kazhdan-Lusztig bases are well-defined. Once a suitable Iwahori-Hecke algebra is defined they will work without further intervention.

For example, with the “standard parameters”, so that $(T_r - q^2)(T_r + 1) = 0$:

```

sage: R.<q> = LaurentPolynomialRing(ZZ)
sage: H = IwahoriHeckeAlgebra('A3', q^2)
sage: T=H.T(); Cp=H.Cp(); C=H.C()
sage: C(T[1])
q*C[1] + q^2
sage: elt = Cp(T[1,2,1]); elt
q^3*Cp[1,2,1] - q^2*Cp[1,2] - q^2*Cp[2,1] + q*Cp[1] + q*Cp[2] - 1
sage: C(elt)
q^3*C[1,2,1] + q^4*C[1,2] + q^4*C[2,1] + q^5*C[1] + q^5*C[2] + q^6

```

With the “normalized presentation”, so that $(T_r - q)(T_r + q^{-1}) = 0$:

```

sage: R.<q> = LaurentPolynomialRing(ZZ)
sage: H = IwahoriHeckeAlgebra('A3', q, -q^-1)
sage: T=H.T(); Cp=H.Cp(); C=H.C()
sage: C(T[1])
C[1] + q
sage: elt = Cp(T[1,2,1]); elt
Cp[1,2,1] - (q^-1)*Cp[1,2] - (q^-1)*Cp[2,1] + (q^-2)*Cp[1] + (q^-2)*Cp[2] - (q^-3)
sage: C(elt)
C[1,2,1] + q*C[1,2] + q*C[2,1] + q^2*C[1] + q^2*C[2] + q^3

```

In the group algebra, so that $(T_r - 1)(T_r + 1) = 0$:

```

sage: H = IwahoriHeckeAlgebra('A3', 1)
sage: T=H.T(); Cp=H.Cp(); C=H.C()
sage: C(T[1])
C[1] + 1
sage: Cp(T[1,2,1])
Cp[1,2,1] - Cp[1,2] - Cp[2,1] + Cp[1] + Cp[2] - 1

```

```
sage: C(_)
C[1,2,1] + C[1,2] + C[2,1] + C[1] + C[2] + 1
```

On the other hand, if the Kazhdan-Lusztig bases are not well-defined (when $-q_1q_2$ is not a square), attempting to use the Kazhdan-Lusztig bases triggers an error:

```
sage: R.<q>=LaurentPolynomialRing(ZZ)
sage: H = IwahoriHeckeAlgebra('A3', q)
sage: C=H.C()
Traceback (most recent call last):
...
ValueError: The Kazhdan_Lusztig bases are defined only when -q_1*q_2 is a square
```

We give an example in affine type:

```
sage: R.<v> = LaurentPolynomialRing(ZZ)
sage: H = IwahoriHeckeAlgebra(['A',2,1], v^2)
sage: T=H.T(); Cp=H.Cp(); C=H.C()
sage: C(T[1,0,2])
v^3*C[1,0,2] + v^4*C[1,0] + v^4*C[0,2] + v^4*C[1,2]
+ v^5*C[0] + v^5*C[2] + v^5*C[1] + v^6
sage: Cp(T[1,0,2])
v^3*Cp[1,0,2] - v^2*Cp[1,0] - v^2*Cp[0,2] - v^2*Cp[1,2]
+ v*Cp[0] + v*Cp[2] + v*Cp[1] - 1
sage: T(C[1,0,2])
(v^-3)*T[1,0,2] - (v^-1)*T[1,0] - (v^-1)*T[0,2] - (v^-1)*T[1,2]
+ v*T[0] + v*T[2] + v*T[1] - v^3
sage: T(Cp[1,0,2])
(v^-3)*T[1,0,2] + (v^-3)*T[1,0] + (v^-3)*T[0,2] + (v^-3)*T[1,2]
+ (v^-3)*T[0] + (v^-3)*T[2] + (v^-3)*T[1] + (v^-3)
```

EXAMPLES:

We start by creating a Iwahori-Hecke algebra together with the three bases for these algebras that are currently supported:

```
sage: R.<v> = LaurentPolynomialRing(QQ, 'v')
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: T = H.T()
sage: C = H.C()
sage: Cp = H.Cp()
```

It is also possible to define these three bases quickly using the `inject_shorthands()` method.

Next we create our generators for the T -basis and do some basic computations and conversions between the bases:

```
sage: T1,T2,T3 = T.algebra_generators()
sage: T1 == T[1]
True
sage: T1*T2 == T[1,2]
True
sage: T1 + T2
T[1] + T[2]
sage: T1*T1
-(1-v^2)*T[1] + v^2
sage: (T1 + T2)*T3 + T1*T1 - (v + v^-1)*T2
T[3,1] + T[2,3] - (1-v^2)*T[1] - (v^-1+v)*T[2] + v^2
```

```

sage: Cp(T1)
v*Cp[1] - 1
sage: Cp((v^1 - 1)*T1*T2 - T3)
-(v^2-v^3)*Cp[1,2] + (v-v^2)*Cp[1] + (v-v^2)*Cp[2] - v*Cp[3] + v
sage: C(T1)
v*C[1] + v^2
sage: p = C(T2*T3 - v*T1); p
v^2*C[2,3] - v^2*C[1] + v^3*C[2] + v^3*C[3] - (v^3-v^4)
sage: Cp(p)
v^2*Cp[2,3] - v^2*Cp[1] - v*Cp[2] - v*Cp[3] + (1+v)
sage: Cp(T2*T3 - v*T1)
v^2*Cp[2,3] - v^2*Cp[1] - v*Cp[2] - v*Cp[3] + (1+v)

```

In addition to explicitly creating generators, we have two shortcuts to basis elements. The first is by using elements of the underlying Coxeter group, the other is by using reduced words:

```

sage: s1,s2,s3 = H.coxeter_group().gens()
sage: T[s1*s2*s1*s3] == T[1,2,1,3]
True
sage: T[1,2,1,3] == T1*T2*T1*T3
True

```

Todo

Implement multi-parameter Iwahori-Hecke algebras together with their Kazhdan-Lusztig bases. That is, Iwahori-Hecke algebras with (possibly) different parameters for each conjugacy class of simple reflections in the underlying Coxeter group.

Todo

When given “generic parameters” we should return the generic Iwahori-Hecke algebra with these parameters and allow the user to work inside this algebra rather than doing calculations behind the scenes in a copy of the generic Iwahori-Hecke algebra. The main problem is that it is not clear how to recognise when the parameters are “generic”.

class *A* (*IHAlgebra*, *prefix=None*)

Bases: `sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra._Basis`

The *A*-basis of an Iwahori-Hecke algebra.

The *A*-basis of the Iwahori-Hecke algebra is the simplest basis that is invariant under the Goldman involution $\#$, up to sign. For w in the underlying Coxeter group define:

$$A_w = T_w + (-1)^{\ell(w)} T_w^\# = T_w + (-1)^{\ell(w)} T_{w^{-1}}^{-1}$$

This gives a basis of the Iwahori-Hecke algebra whenever 2 is a unit in the base ring. The *A*-basis induces a $\mathbb{Z}/2\mathbb{Z}$ -grading on the Iwahori-Hecke algebra.

The *A*-basis is a basis only when 2 is invertible. An error is raised whenever 2 is not a unit in the base ring.

EXAMPLES:

```

sage: R.<v> = LaurentPolynomialRing(QQ, 'v')
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: A=H.A(); T=H.T()
sage: T(A[1])

```

```

T[1] + (1/2-1/2*v^2)
sage: T(A[1,2])
T[1,2] + (1/2-1/2*v^2)*T[1] + (1/2-1/2*v^2)*T[2] + (1/2-v^2+1/2*v^4)
sage: A[1]*A[2]
A[1,2] - (1/4-1/2*v^2+1/4*v^4)

```

goldman_involution_on_basis (*w*)

Return the effect of applying the Goldman involution to the basis element `self[w]`.

This function is not intended to be called directly. Instead, use `goldman_involution()`.

EXAMPLES:

```

sage: R.<v> = LaurentPolynomialRing(QQ, 'v')
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: A=H.A()
sage: s=H.coxeter_group().simple_reflection(1)
sage: A.goldman_involution_on_basis(s)
-A[1]
sage: A[1,2].goldman_involution()
A[1,2]

```

to_T_basis (*w*)

Return the *A*-basis element `self[w]` as a linear combination of *T*-basis elements.

EXAMPLES:

```

sage: R.<v> = LaurentPolynomialRing(QQ)
sage: H = IwahoriHeckeAlgebra('A3', v**2); A=H.A(); T=H.T()
sage: s=H.coxeter_group().simple_reflection(1)
sage: A.to_T_basis(s)
T[1] + (1/2-1/2*v^2)
sage: T(A[1,2])
T[1,2] + (1/2-1/2*v^2)*T[1] + (1/2-1/2*v^2)*T[2] + (1/2-v^2+1/2*v^4)
sage: A(T[1,2])
A[1,2] - (1/2-1/2*v^2)*A[1] - (1/2-1/2*v^2)*A[2]

```

class `IwahoriHeckeAlgebra.B` (*IHAlgebra*, *prefix=None*)

Bases: `sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra._Basis`

The *B*-basis of an Iwahori-Hecke algebra.

The *B*-basis is the unique basis of the Iwahori-Hecke algebra that is invariant under the Goldman involution, up to sign, and invariant under the Kazhdan-Lusztig bar involution. In the generic case, the *B*-basis becomes the group basis of the group algebra of the Coxeter group the *B*-basis upon setting the Hecke parameters equal to 1. If *w* is an element of the corresponding Coxeter group then the *B*-basis element B_w is uniquely determined by the conditions that $B_w^\# = (-1)^{\ell(w)} B_w$, where $\#$ is the Goldman involution and

$$B_w = T_w + \sum_{v < w} b_{vw}(q) T_v$$

where $b_{vw}(q) \neq 0$ only if $v < w$ in the Bruhat order and $\ell(v) \not\equiv \ell(w) \pmod{2}$.

This gives a basis of the Iwahori-Hecke algebra whenever 2 is a unit in the base ring. The *B*-basis induces a $\mathbb{Z}/2\mathbb{Z}$ -grading on the Iwahori-Hecke algebra. The *B*-basis elements are also invariant under the Kazhdan-Lusztig bar involution and hence are related to the Kazhdan-Lusztig bases.

The *B*-basis is a basis only when 2 is invertible. An error is raised whenever 2 is not a unit in the base ring.

EXAMPLES:

```
sage: R.<v> = LaurentPolynomialRing(QQ, 'v')
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: A=H.A(); T=H.T(); Cp=H.Cp()
sage: T(A[1])
T[1] + (1/2-1/2*v^2)
sage: T(A[1,2])
T[1,2] + (1/2-1/2*v^2)*T[1] + (1/2-1/2*v^2)*T[2] + (1/2-v^2+1/2*v^4)
sage: A[1]*A[2]
A[1,2] - (1/4-1/2*v^2+1/4*v^4)
sage: Cp(A[1]*A[2])
v^2*Cp[1,2] - (1/2*v+1/2*v^3)*Cp[1] - (1/2*v+1/2*v^3)*Cp[2]
+ (1/4+1/2*v^2+1/4*v^4)
sage: Cp(A[1])
v*Cp[1] - (1/2+1/2*v^2)
sage: Cp(A[1,2])
v^2*Cp[1,2] - (1/2*v+1/2*v^3)*Cp[1]
- (1/2*v+1/2*v^3)*Cp[2] + (1/2+1/2*v^4)
sage: Cp(A[1,2,1])
v^3*Cp[1,2,1] - (1/2*v^2+1/2*v^4)*Cp[1,2]
- (1/2*v^2+1/2*v^4)*Cp[2,1] + (1/2*v+1/2*v^5)*Cp[1]
+ (1/2*v+1/2*v^5)*Cp[2] - (1/2+1/2*v^6)
```

goldman_involution_on_basis (w)

Return the Goldman involution to the basis element indexed by w .

This function is not intended to be called directly. Instead, use `goldman_involution()` .

EXAMPLES:

```
sage: R.<v> = LaurentPolynomialRing(QQ, 'v')
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: B=H.B()
sage: s=H.coxeter_group().simple_reflection(1)
sage: B.goldman_involution_on_basis(s)
-B[1]
sage: B[1,2].goldman_involution()
B[1,2]
```

to_T_basis (w)

Return the B -basis element `self[w]` as a linear combination of T -basis elements.

EXAMPLES:

```
sage: R.<v> = LaurentPolynomialRing(QQ)
sage: H = IwahoriHeckeAlgebra('A3', v**2); B=H.B(); T=H.T()
sage: s=H.coxeter_group().simple_reflection(1)
sage: B.to_T_basis(s)
T[1] + (1/2-1/2*v^2)
sage: T(B[1,2])
T[1,2] + (1/2-1/2*v^2)*T[1] + (1/2-1/2*v^2)*T[2]
sage: B(T[1,2])
B[1,2] - (1/2-1/2*v^2)*B[1] - (1/2-1/2*v^2)*B[2] + (1/2-v^2+1/2*v^4)
```

class IwahoriHeckeAlgebra. **C** (IHAAlgebra, prefix=None)

Bases: `sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra._KLHeckeBasis`

The Kazhdan-Lusztig C -basis of Iwahori-Hecke algebra.

Assuming the standard quadratic relations of $(T_r - q)(T_r + 1) = 0$, for every element w in the Coxeter group, there is a unique element C_w in the Iwahori-Hecke algebra which is uniquely determined by the two properties:

$$\overline{C_w} = C_w$$

$$C_w = (-1)^{\ell(w)} q^{\ell(w)/2} \sum_{v \leq w} (-q)^{-\ell(v)} \overline{P_{v,w}(q)} T_v$$

where \leq is the Bruhat order on the underlying Coxeter group and $P_{v,w}(q) \in \mathbf{Z}[q, q^{-1}]$ are polynomials in $\mathbf{Z}[q]$ such that $P_{w,w}(q) = 1$ and if $v < w$ then $\deg P_{v,w}(q) \leq \frac{1}{2}(\ell(w) - \ell(v) - 1)$. This is related to the C' Kazhdan-Lusztig basis by $C_i = -\alpha(C'_i)$ where α is the \mathbf{Z} -linear Hecke involution defined by $q^{1/2} \mapsto q^{-1/2}$ and $\alpha(T_i) = -(q_1 q_2)^{-1/2} T_i$.

More generally, if the quadratic relations are of the form $(T_{s-q_1})(T_{s-q_2})=0$ and $\sqrt{-q_1 q_2}$ exists then, for a simple reflection s , the corresponding Kazhdan-Lusztig basis element is:

$$C_s = (-q_1 q_2)^{1/2} (1 - (-q_1 q_2)^{-1/2} T_s).$$

See [KL79] for more details.

EXAMPLES:

```
sage: R.<v> = LaurentPolynomialRing(QQ)
sage: H = IwahoriHeckeAlgebra('A5', v**2)
sage: W = H.coxeter_group()
sage: s1,s2,s3,s4,s5 = W.simple_reflections()
sage: T = H.T()
sage: C = H.C()
sage: T(s1)**2
-(1-v^2)*T[1] + v^2
sage: T(C(s1))
(v^-1)*T[1] - v
sage: T(C(s1)*C(s2)*C(s1))
(v^-3)*T[1,2,1] - (v^-1)*T[1,2] - (v^-1)*T[2,1] + (v^-1+v)*T[1] + v*T[2] -
↳ (v+v^3)
```

```
sage: R.<v> = LaurentPolynomialRing(QQ)
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: W = H.coxeter_group()
sage: s1,s2,s3 = W.simple_reflections()
sage: C = H.C()
sage: C(s1*s2*s1)
C[1,2,1]
sage: C(s1)**2
-(v^-1+v)*C[1]
sage: C(s1)*C(s2)*C(s1)
C[1,2,1] + C[1]
```

hash_involution_on_basis (w)

Return the effect of applying the hash involution to the basis element `self[w]`.

This function is not intended to be called directly. Instead, use `hash_involution()`.

EXAMPLES:

```
sage: R.<v> = LaurentPolynomialRing(QQ, 'v')
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: C=H.C()
```

```

sage: s=H.coxeter_group().simple_reflection(1)
sage: C.hash_involution_on_basis(s)
-C[1] - (v^-1+v)
sage: C[s].hash_involution()
-C[1] - (v^-1+v)

```

class IwahoriHeckeAlgebra. **Cp** (*IHAlgebra*, *prefix=None*)
 Bases: sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra._KLHeckeBasis

The C' Kazhdan-Lusztig basis of Iwahori-Hecke algebra.

Assuming the standard quadratic relations of $(T_r - q)(T_r + 1) = 0$, for every element w in the Coxeter group, there is a unique element C'_w in the Iwahori-Hecke algebra which is uniquely determined by the two properties:

$$\overline{C'_w} = C'_w$$

$$C'_w = q^{-\ell(w)/2} \sum_{v \leq w} P_{v,w}(q) T_v$$

where \leq is the Bruhat order on the underlying Coxeter group and $P_{v,w}(q) \in \mathbf{Z}[q, q^{-1}]$ are polynomials in $\mathbf{Z}[q]$ such that $P_{w,w}(q) = 1$ and if $v < w$ then $\deg P_{v,w}(q) \leq \frac{1}{2}(\ell(w) - \ell(v) - 1)$.

More generally, if the quadratic relations are of the form $(T_s - q_1)(T_s - q_2) = 0$ and $\sqrt{-q_1 q_2}$ exists then, for a simple reflection s , the corresponding Kazhdan-Lusztig basis element is:

$$C'_s = (-q_1 q_2)^{-1/2} (T_s + 1).$$

See [KL79] for more details.

EXAMPLES:

```

sage: R = LaurentPolynomialRing(QQ, 'v')
sage: v = R.gen(0)
sage: H = IwahoriHeckeAlgebra('A5', v**2)
sage: W = H.coxeter_group()
sage: s1,s2,s3,s4,s5 = W.simple_reflections()
sage: T = H.T()
sage: Cp = H.Cp()
sage: T(s1)**2
-(1-v^2)*T[1] + v^2
sage: T(Cp(s1))
(v^-1)*T[1] + (v^-1)
sage: T(Cp(s1)*Cp(s2)*Cp(s1))
(v^-3)*T[1,2,1] + (v^-3)*T[1,2] + (v^-3)*T[2,1]
+ (v^-3+v^-1)*T[1] + (v^-3)*T[2] + (v^-3+v^-1)

```

```

sage: R = LaurentPolynomialRing(QQ, 'v')
sage: v = R.gen(0)
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: W = H.coxeter_group()
sage: s1,s2,s3 = W.simple_reflections()
sage: Cp = H.Cp()
sage: Cp(s1*s2*s1)
Cp[1,2,1]
sage: Cp(s1)**2
(v^-1+v)*Cp[1]
sage: Cp(s1)*Cp(s2)*Cp(s1)

```



```
Cp[1,2,1] + Cp[1]
sage: Cp(s1)*Cp(s2)*Cp(s3)*Cp(s1)*Cp(s2) # long time
Cp[1,2,3,1,2] + Cp[1,2,1] + Cp[3,1,2]
```

hash_involution_on_basis (w)

Return the effect of applying the hash involution to the basis element `self[w]`.

This function is not intended to be called directly. Instead, use `hash_involution()`.

EXAMPLES:

```
sage: R.<v> = LaurentPolynomialRing(QQ, 'v')
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: Cp=H.Cp()
sage: s=H.coxeter_group().simple_reflection(1)
sage: Cp.hash_involution_on_basis(s)
-Cp[1] + (v^-1+v)
sage: Cp[s].hash_involution()
-Cp[1] + (v^-1+v)
```

class IwahoriHeckeAlgebra. **T** (algebra, prefix=None)

Bases: `sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra._Basis`

The standard basis of Iwahori-Hecke algebra.

For every simple reflection s_i of the Coxeter group, there is a corresponding generator T_i of Iwahori-Hecke algebra. These are subject to the relations:

$$(T_i - q_1)(T_i - q_2) = 0$$

together with the braid relations:

$$T_i T_j T_i \cdots = T_j T_i T_j \cdots,$$

where the number of terms on each of the two sides is the order of $s_i s_j$ in the Coxeter group.

Weyl group elements form a basis of Iwahori-Hecke algebra H with the property that if w_1 and w_2 are Coxeter group elements such that $\ell(w_1 w_2) = \ell(w_1) + \ell(w_2)$ then $T_{w_1 w_2} = T_{w_1} T_{w_2}$.

With the default value $q_2 = -1$ and with $q_1 = q$ the generating relation may be written $T_i^2 = (q - 1) \cdot T_i + q \cdot 1$ as in [Iwa1964].

EXAMPLES:

```
sage: H = IwahoriHeckeAlgebra("A3", 1)
sage: T = H.T()
sage: T1, T2, T3 = T.algebra_generators()
sage: T1*T2*T3*T1*T2*T1 == T3*T2*T1*T3*T2*T3
True
sage: w0 = T(H.coxeter_group().long_element())
sage: w0
T[1,2,3,1,2,1]
sage: T = H.T(prefix="s")
sage: T.an_element()
2*s[1,2,3,2,1] + 3*s[1,2,3,1] + s[1,2,3] + 1
```

class Element (M, x)

Bases: `sage.combinat.free_module.CombinatorialFreeModuleElement`

A class for elements of an Iwahori-Hecke algebra in the T basis.

inverse ()

Return the inverse if `self` is a basis element.

An element is a basis element if it is T_w where w is in the Weyl group. The base ring must be a field or Laurent polynomial ring. Other elements of the ring have inverses but the inverse method is only implemented for the basis elements.

EXAMPLES:

```
sage: R.<q> = LaurentPolynomialRing(QQ)
sage: H = IwahoriHeckeAlgebra("A2", q).T()
sage: [T1,T2] = H.algebra_generators()
sage: x = (T1*T2).inverse(); x
(q^-2)*T[2,1] + (q^-2-q^-1)*T[1] + (q^-2-q^-1)*T[2] + (q^-2-2*q^-1+1)
sage: x*T1*T2
1
```

IwahoriHeckeAlgebra.T.bar_on_basis (w)

Return the bar involution of T_w , which is $T_{w^{-1}}^{-1}$.

EXAMPLES:

```
sage: R.<v> = LaurentPolynomialRing(QQ)
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: W = H.coxeter_group()
sage: s1,s2,s3 = W.simple_reflections()
sage: T = H.T()
sage: b = T.bar_on_basis(s1*s2*s3); b
(v^-6)*T[1,2,3]
+ (v^-6-v^-4)*T[1,2]
+ (v^-6-v^-4)*T[3,1]
+ (v^-6-v^-4)*T[2,3]
+ (v^-6-2*v^-4+v^-2)*T[1]
+ (v^-6-2*v^-4+v^-2)*T[2]
+ (v^-6-2*v^-4+v^-2)*T[3]
+ (v^-6-3*v^-4+3*v^-2-1)
sage: b.bar()
T[1,2,3]
```

IwahoriHeckeAlgebra.T.goldman_involution_on_basis (w)

Return the Goldman involution to the basis element indexed by w .

The goldman involution is the algebra involution of the Iwahori-Hecke algebra determined by

$$T_w \mapsto (-q_1 q_2)^{\ell(w)} T_{w^{-1}}^{-1},$$

where w is an element of the corresponding Coxeter group.

This map is defined in [Iwa1964] and it is used to define the alternating subalgebra of the Iwahori-Hecke algebra, which is the fixed-point subalgebra of the Goldman involution.

This function is not intended to be called directly. Instead, use `goldman_involution()`.

EXAMPLES:

```
sage: R.<v> = LaurentPolynomialRing(QQ, 'v')
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: T=H.T()
sage: s=H.coxeter_group().simple_reflection(1)
sage: T.goldman_involution_on_basis(s)
```

```

-T[1] - (1-v^2)
sage: T[s].goldman_involution()
-T[1] - (1-v^2)
sage: h = T[1]*T[2] + (v^3 - v^-1 + 2)*T[3,1,2,3]
sage: h.goldman_involution()
-(v^-1-2-v^3)*T[1,2,3,2] - (v^-1-2-v+2*v^2-v^3+v^5)*T[1,2,3]
- (v^-1-2-v+2*v^2-v^3+v^5)*T[3,1,2]
- (v^-1-2-v+2*v^2-v^3+v^5)*T[2,3,2]
- (v^-1-3-2*v+4*v^2-2*v^4+2*v^5-v^7)*T[1,2]
- (v^-1-2-2*v+4*v^2-2*v^4+2*v^5-v^7)*T[3,1]
- (v^-1-2-2*v+4*v^2-2*v^4+2*v^5-v^7)*T[2,3]
- (v^-1-2-2*v+4*v^2-2*v^4+2*v^5-v^7)*T[3,2]
- (v^-1-3-2*v+5*v^2+v^3-4*v^4+v^5+2*v^6-2*v^7+v^9)*T[1]
- (v^-1-3-3*v+7*v^2+2*v^3-6*v^4+2*v^5+2*v^6-3*v^7+v^9)*T[2]
- (v^-1-2-3*v+6*v^2+2*v^3-6*v^4+2*v^5+2*v^6-3*v^7+v^9)*T[3]
- (v^-1-3-3*v+8*v^2+3*v^3-9*v^4+6*v^6-3*v^7-2*v^8+3*v^9-v^11)
sage: h.goldman_involution().goldman_involution() == h
True

```

IwahoriHeckeAlgebra.T. **hash_involution_on_basis** (w)

Return the hash involution on the basis element `self[w]`.

The hash involution α is a \mathbf{Z} -algebra involution of the Iwahori-Hecke algebra determined by $q^{1/2} \mapsto q^{-1/2}$, and $T_w \mapsto (-q_1 q_2)^{-\ell(w)} T_w$, for w an element of the corresponding Coxeter group.

This map is defined in [KL79] and it is used to change between the C and C' bases because $\alpha(C_w) = (-1)^{\ell(w)} C'_w$.

This function is not intended to be called directly. Instead, use `hash_involution()`.

EXAMPLES:

```

sage: R.<v> = LaurentPolynomialRing(QQ, 'v')
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: T=H.T()
sage: s=H.coxeter_group().simple_reflection(1)
sage: T.hash_involution_on_basis(s)
-(v^-2)*T[1]
sage: T[s].hash_involution()
-(v^-2)*T[1]
sage: h = T[1]*T[2] + (v^3 - v^-1 + 2)*T[3,1,2,3]
sage: h.hash_involution()
(v^-11+2*v^-8-v^-7)*T[1,2,3,2] + (v^-4)*T[1,2]
sage: h.hash_involution().hash_involution() == h
True

```

IwahoriHeckeAlgebra.T. **inverse_generator** (i)

Return the inverse of the i -th generator, if it exists.

This method is only available if the Iwahori-Hecke algebra parameters q_1 and q_2 are both invertible. In this case, the algebra generators are also invertible and this method returns the inverse of `self.algebra_generator(i)`.

EXAMPLES:

```

sage: P.<q1, q2>=QQ[]
sage: F = Frac(P)
sage: H = IwahoriHeckeAlgebra("A2", q1, q2=q2, base_ring=F).T()
sage: H.base_ring()
Fraction Field of Multivariate Polynomial Ring in q1, q2 over Rational_
->Field

```

```

sage: H.inverse_generator(1)
-1/(q1*q2)*T[1] + ((q1+q2)/(q1*q2))
sage: H = IwahoriHeckeAlgebra("A2", q1, base_ring=F).T()
sage: H.inverse_generator(2)
-(1/(-q1))*T[2] + ((q1-1)/(-q1))
sage: P1.<r1, r2> = LaurentPolynomialRing(QQ)
sage: H1 = IwahoriHeckeAlgebra("B2", r1, q2=r2, base_ring=P1).T()
sage: H1.base_ring()
Multivariate Laurent Polynomial Ring in r1, r2 over Rational Field
sage: H1.inverse_generator(2)
(-r1^-1*r2^-1)*T[2] + (r2^-1+r1^-1)
sage: H2 = IwahoriHeckeAlgebra("C2", r1, base_ring=P1).T()
sage: H2.inverse_generator(2)
(r1^-1)*T[2] + (-1+r1^-1)

```

`IwahoriHeckeAlgebra.T.inverse_generators()`

Return the inverses of all the generators, if they exist.

This method is only available if q_1 and q_2 are invertible. In that case, the algebra generators are also invertible.

EXAMPLES:

```

sage: P.<q> = PolynomialRing(QQ)
sage: F = Frac(P)
sage: H = IwahoriHeckeAlgebra("A2", q, base_ring=F).T()
sage: T1,T2 = H.algebra_generators()
sage: U1,U2 = H.inverse_generators()
sage: U1*T1,T1*U1
(1, 1)
sage: P1.<q> = LaurentPolynomialRing(QQ)
sage: H1 = IwahoriHeckeAlgebra("A2", q, base_ring=P1).T(prefix="V")
sage: V1,V2 = H1.algebra_generators()
sage: W1,W2 = H1.inverse_generators()
sage: [W1,W2]
[(q^-1)*V[1] + (q^-1-1), (q^-1)*V[2] + (q^-1-1)]
sage: V1*W1, W2*V2
(1, 1)

```

`IwahoriHeckeAlgebra.T.product_by_generator(x, i, side='right')`

Return $T_i \cdot x$, where T_i is the i -th generator. This is coded individually for use in `x._mul_()`.

EXAMPLES:

```

sage: R.<q> = QQ[]; H = IwahoriHeckeAlgebra("A2", q).T()
sage: T1, T2 = H.algebra_generators()
sage: [H.product_by_generator(x, 1) for x in [T1,T2]]
[(q-1)*T[1] + q, T[2,1]]
sage: [H.product_by_generator(x, 1, side = "left") for x in [T1,T2]]
[(q-1)*T[1] + q, T[1,2]]

```

`IwahoriHeckeAlgebra.T.product_by_generator_on_basis(w, i, side='right')`

Return the product $T_w T_i$ (resp. $T_i T_w$) if side is 'right' (resp. 'left').

If the quadratic relation is $(T_i - u)(T_i - v) = 0$, then we have

$$T_w T_i = \begin{cases} T_{ws_i} & \text{if } \ell(ws_i) = \ell(w) + 1, \\ (u+v)T_{ws_i} - uvT_w & \text{if } \ell(ws_i) = \ell(w) - 1. \end{cases}$$

The left action is similar.

INPUT:

- w – an element of the Coxeter group
- i – an element of the index set
- $side$ – 'right' (default) or 'left'

EXAMPLES:

```
sage: R.<q> = QQ[]; H = IwahoriHeckeAlgebra("A2", q)
sage: T = H.T()
sage: s1,s2 = H.coxeter_group().simple_reflections()
sage: [T.product_by_generator_on_basis(w, 1) for w in [s1,s2,s1*s2]]
[(q-1)*T[1] + q, T[2,1], T[1,2,1]]
sage: [T.product_by_generator_on_basis(w, 1, side="left") for w in
↪ [s1,s2,s1*s2]]
[(q-1)*T[1] + q, T[1,2], (q-1)*T[1,2] + q*T[2]]
```

`IwahoriHeckeAlgebra.T.product_on_basis (w1,w2)`

Return $T_{w_1}T_{w_2}$, where w_1 and w_2 are words in the Coxeter group.

EXAMPLES:

```
sage: R.<q> = QQ[]; H = IwahoriHeckeAlgebra("A2", q)
sage: T = H.T()
sage: s1,s2 = H.coxeter_group().simple_reflections()
sage: [T.product_on_basis(s1,x) for x in [s1,s2]]
[(q-1)*T[1] + q, T[1,2]]
```

`IwahoriHeckeAlgebra.T.to_C_basis (w)`

Return T_w as a linear combination of C -basis elements.

EXAMPLES:

```
sage: R = LaurentPolynomialRing(QQ, 'v')
sage: v = R.gen(0)
sage: H = IwahoriHeckeAlgebra('A2', v**2)
sage: s1,s2 = H.coxeter_group().simple_reflections()
sage: T = H.T()
sage: C = H.C()
sage: T.to_C_basis(s1)
v*T[1] + v^2
sage: C(T(s1))
v*C[1] + v^2
sage: C(v^-1*T(s1) - v)
C[1]
sage: C(T(s1*s2)+T(s1)+T(s2)+1)
v^2*C[1,2] + (v+v^3)*C[1] + (v+v^3)*C[2] + (1+2*v^2+v^4)
sage: C(T(s1*s2*s1))
v^3*C[1,2,1] + v^4*C[1,2] + v^4*C[2,1] + v^5*C[1] + v^5*C[2] + v^6
```

`IwahoriHeckeAlgebra.T.to_Cp_basis (w)`

Return T_w as a linear combination of C' -basis elements.

EXAMPLES:

```
sage: R.<v> = LaurentPolynomialRing(QQ)
sage: H = IwahoriHeckeAlgebra('A2', v**2)
sage: s1,s2 = H.coxeter_group().simple_reflections()
sage: T = H.T()
```

```

sage: Cp = H.Cp()
sage: T.to_Cp_basis(s1)
v*Cp[1] - 1
sage: Cp(T(s1))
v*Cp[1] - 1
sage: Cp(T(s1)+1)
v*Cp[1]
sage: Cp(T(s1*s2)+T(s1)+T(s2)+1)
v^2*Cp[1,2]
sage: Cp(T(s1*s2*s1))
v^3*Cp[1,2,1] - v^2*Cp[1,2] - v^2*Cp[2,1] + v*Cp[1] + v*Cp[2] - 1

```

`IwahoriHeckeAlgebra.a_realization()`
Return a particular realization of self (the T -basis).

EXAMPLES:

```

sage: H = IwahoriHeckeAlgebra("B2", 1)
sage: H.a_realization()
Iwahori-Hecke algebra of type B2 in 1,-1 over Integer Ring in the T-basis

```

`IwahoriHeckeAlgebra.cartan_type()`
Return the Cartan type of self.

EXAMPLES:

```

sage: IwahoriHeckeAlgebra("D4", 1).cartan_type()
['D', 4]

```

`IwahoriHeckeAlgebra.coxeter_group()`
Return the Coxeter group of self.

EXAMPLES:

```

sage: IwahoriHeckeAlgebra("B2", 1).coxeter_group()
Weyl Group of type ['B', 2] (as a matrix group acting on the ambient space)

```

`IwahoriHeckeAlgebra.q1()`
Return the parameter q_1 of self.

EXAMPLES:

```

sage: H = IwahoriHeckeAlgebra("B2", 1)
sage: H.q1()
1

```

`IwahoriHeckeAlgebra.q2()`
Return the parameter q_2 of self.

EXAMPLES:

```

sage: H = IwahoriHeckeAlgebra("B2", 1)
sage: H.q2()
-1

```

`class sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_nonstandard (W)`
Bases: `sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra`

This is a class which is used behind the scenes by `IwahoriHeckeAlgebra` to compute the Kazhdan-Lusztig bases. It is not meant to be used directly. It implements the slightly idiosyncratic (but convenient) Iwahori-Hecke algebra with two parameters which is defined over the Laurent polynomial ring $\mathbb{Z}[u, u^{-1}, v, v^{-1}]$ in two variables and has quadratic relations:

$$(T_r - u)(T_r + v^2/u) = 0.$$

The point of these relations is that the product of the two parameters is v^2 which is a square in $\mathbb{Z}[u, u^{-1}, v, v^{-1}]$. Consequently, the Kazhdan-Lusztig bases are defined for this algebra.

More generally, if we have a Iwahori-Hecke algebra with two parameters which has quadratic relations of the form:

$$(T_r - q_1)(T_r - q_2) = 0$$

where $-q_1q_2$ is a square then the Kazhdan-Lusztig bases are well-defined for this algebra. Moreover, these bases be computed by specialization from the generic Iwahori-Hecke algebra using the specialization which sends $u \mapsto q_1$ and $v \mapsto \sqrt{-q_1q_2}$, so that $v^2/u \mapsto -q_2$.

For example, if $q_1 = q = Q^2$ and $q_2 = -1$ then $u \mapsto q$ and $v \mapsto \sqrt{q} = Q$; this is the standard presentation of the Iwahori-Hecke algebra with $(T_r - q)(T_r + 1) = 0$. On the other hand, when $q_1 = q$ and $q_2 = -q^{-1}$ then $u \mapsto q$ and $v \mapsto 1$. This is the normalized presentation with $(T_r - v)(T_r + v^{-1}) = 0$.

Warning: This class uses non-standard parameters for the Iwahori-Hecke algebra and are related to the standard parameters by an outer automorphism that is non-trivial on the T -basis.

class C (*IHAlgebra*, *prefix=None*)

Bases: `sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.C`

The Kazhdan-Lusztig C -basis for the generic Iwahori-Hecke algebra.

to_T_basis (*w*)

Return C_w as a linear combination of T -basis elements.

EXAMPLES:

```
sage: H = sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_
      ↪nonstandard("A3")
sage: s1,s2,s3 = H.coxeter_group().simple_reflections()
sage: T = H.T()
sage: C = H.C()
sage: C.to_T_basis(s1)
(v^-1)*T[1] + (-u*v^-1)
sage: C.to_T_basis(s1*s2)
(v^-2)*T[1,2] + (-u*v^-2)*T[1] + (-u*v^-2)*T[2] + (u^2*v^-2)
sage: C.to_T_basis(s1*s2*s1)
(v^-3)*T[1,2,1] + (-u*v^-3)*T[1,2] + (-u*v^-3)*T[2,1]
+ (u^2*v^-3)*T[1] + (u^2*v^-3)*T[2] + (-u^3*v^-3)
sage: T(C(s1*s2*s1))
(v^-3)*T[1,2,1] + (-u*v^-3)*T[1,2] + (-u*v^-3)*T[2,1]
+ (u^2*v^-3)*T[1] + (u^2*v^-3)*T[2] + (-u^3*v^-3)
sage: T(C(s2*s1*s3*s2))
(v^-4)*T[2,3,1,2] + (-u*v^-4)*T[1,2,1] + (-u*v^-4)*T[3,1,2]
+ (-u*v^-4)*T[2,3,1] + (-u*v^-4)*T[2,3,2] + (u^2*v^-4)*T[1,2]
+ (u^2*v^-4)*T[2,1] + (u^2*v^-4)*T[3,1] + (u^2*v^-4)*T[2,3]
+ (u^2*v^-4)*T[3,2] + (-u^3*v^-4)*T[1]
+ (-u^3*v^-4-u*v^-2)*T[2] + (-u^3*v^-4)*T[3]
+ (u^4*v^-4+u^2*v^-2)
```

class IwahoriHeckeAlgebra_nonstandard. **Cp** (*IHAlgebra*, *prefix=None*)
 Bases: [sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.Cp](#)

The Kazhdan-Lusztig C' -basis for the generic Iwahori-Hecke algebra.

to_T_basis (*w*)

Return C'_w as a linear combination of T -basis elements.

EXAMPLES:

```
sage: H = sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_
      ↪nonstandard("A3")
sage: s1,s2,s3 = H.coxeter_group().simple_reflections()
sage: T = H.T()
sage: Cp = H.Cp()
sage: Cp.to_T_basis(s1)
(v^-1)*T[1] + (u^-1*v)
sage: Cp.to_T_basis(s1*s2)
(v^-2)*T[1,2] + (u^-1)*T[1] + (u^-1)*T[2] + (u^-2*v^2)
sage: Cp.to_T_basis(s1*s2*s1)
(v^-3)*T[1,2,1] + (u^-1*v^-1)*T[1,2] + (u^-1*v^-1)*T[2,1]
+ (u^-2*v)*T[1] + (u^-2*v)*T[2] + (u^-3*v^3)
sage: T(Cp(s1*s2*s1))
(v^-3)*T[1,2,1] + (u^-1*v^-1)*T[1,2] + (u^-1*v^-1)*T[2,1]
+ (u^-2*v)*T[1] + (u^-2*v)*T[2] + (u^-3*v^3)
sage: T(Cp(s2*s1*s3*s2))
(v^-4)*T[2,3,1,2] + (u^-1*v^-2)*T[1,2,1] + (u^-1*v^-2)*T[3,1,2]
+ (u^-1*v^-2)*T[2,3,1] + (u^-1*v^-2)*T[2,3,2] + (u^-2)*T[1,2]
+ (u^-2)*T[2,1] + (u^-2)*T[3,1] + (u^-2)*T[2,3]
+ (u^-2)*T[3,2] + (u^-3*v^2)*T[1] + (u^-1+u^-3*v^2)*T[2]
+ (u^-3*v^2)*T[3] + (u^-2*v^2+u^-4*v^4)
```

class IwahoriHeckeAlgebra_nonstandard. **T** (*algebra*, *prefix=None*)

Bases: [sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T](#)

The T -basis for the generic Iwahori-Hecke algebra.

to_C_basis (*w*)

Return T_w as a linear combination of C -basis elements.

To compute this we piggy back off the C' -basis conversion using the observation that the hash involution sends T_w to $(-q_1q_2)^{\ell(w)}T_w$ and C_w to $(-1)^{\ell(w)}C'_w$. Therefore, if

$$T_w = \sum_v a_{vw} C'_v$$

then

$$T_w = (-q_1q_2)^{\ell(w)} \left(\sum_v a_{vw} C'_v \right)^\# = \sum_v (-1)^{\ell(v)} \overline{a_{vw}} C_v$$

Note that we cannot just apply `hash_involution()` here because this involution always returns the answer with respect to the same basis.

EXAMPLES:

```
sage: H = sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_
      ↪nonstandard("A2")
sage: s1,s2 = H.coxeter_group().simple_reflections()
sage: T = H.T()
sage: C = H.C()
```



```

sage: T.to_C_basis(s1)
v*T[1] + u
sage: C(T(s1))
v*C[1] + u
sage: C(T( C[1] ))
C[1]
sage: C(T(s1*s2)+T(s1)+T(s2)+1)
v^2*C[1,2] + (u*v+v)*C[1] + (u*v+v)*C[2] + (u^2+2*u+1)
sage: C(T(s1*s2*s1))
v^3*C[1,2,1] + u*v^2*C[1,2] + u*v^2*C[2,1] + u^2*v*C[1] + u^2*v*C[2] + u^3

```

to_Cp_basis (w)

Return T_w as a linear combination of C' -basis elements.

EXAMPLES:

```

sage: H = sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_
      ↪nonstandard("A2")
sage: s1,s2 = H.coxeter_group().simple_reflections()
sage: T = H.T()
sage: Cp = H.Cp()
sage: T.to_Cp_basis(s1)
v*Cp[1] + (-u^-1*v^2)
sage: Cp(T(s1))
v*Cp[1] + (-u^-1*v^2)
sage: Cp(T(s1)+1)
v*Cp[1] + (-u^-1*v^2+1)
sage: Cp(T(s1*s2)+T(s1)+T(s2)+1)
v^2*Cp[1,2] + (-u^-1*v^3+v)*Cp[1] + (-u^-1*v^3+v)*Cp[2] + (u^-2*v^4-2*u^-
      ↪1*v^2+1)
sage: Cp(T(s1*s2*s1))
v^3*Cp[1,2,1] + (-u^-1*v^4)*Cp[1,2] + (-u^-1*v^4)*Cp[2,1]
      + (u^-2*v^5)*Cp[1] + (u^-2*v^5)*Cp[2] + (-u^-3*v^6)

```

`sage.algebras.iwahori_hecke_algebra.index_cmp` (x, y)

Compare two term indices x and y by Bruhat order, then by word length, and then by the generic comparison.

EXAMPLES:

```

sage: from sage.algebras.iwahori_hecke_algebra import index_cmp
sage: W = WeylGroup(['A',2,1])
sage: x = W.from_reduced_word([0,1])
sage: y = W.from_reduced_word([0,2,1])
sage: x.bruhat_le(y)
True
sage: index_cmp(x, y)
1

```

`sage.algebras.iwahori_hecke_algebra.normalized_laurent_polynomial` (R, p)

Return a normalized version of the (Laurent polynomial) p in the ring R .

Various ring operations in `sage` return an element of the field of fractions of the parent ring even though the element is “known” to belong to the base ring. This function is a hack to recover from this. This occurs somewhat haphazardly with Laurent polynomial rings:

```

sage: R.<q>=LaurentPolynomialRing(ZZ)
sage: [type(c) for c in (q**-1).coefficients()]
[<type 'sage.rings.integer.Integer'>]

```

It also happens in any ring when dividing by units:

```
sage: type ( 3/1 )
<type 'sage.rings.rational.Rational'>
sage: type ( -1/-1 )
<type 'sage.rings.rational.Rational'>
```

This function is a variation on a suggested workaround of Nils Bruin.

EXAMPLES:

```
sage: from sage.algebras.iwahori_hecke_algebra import normalized_laurent_
      ↪polynomial
sage: type ( normalized_laurent_polynomial(ZZ, 3/1) )
<type 'sage.rings.integer.Integer'>
sage: R.<q>=LaurentPolynomialRing(ZZ)
sage: [type(c) for c in normalized_laurent_polynomial(R, q**-1).coefficients()]
[<type 'sage.rings.integer.Integer'>]
sage: R.<u,v>=LaurentPolynomialRing(ZZ,2)
sage: p=normalized_laurent_polynomial(R, 2*u**-1*v**-1+u*v)
sage: ui=normalized_laurent_polynomial(R, u^-1)
sage: vi=normalized_laurent_polynomial(R, v^-1)
sage: p(ui,vi)
2*u*v + u^-1*v^-1
sage: q= u+v+ui
sage: q(ui,vi)
u + v^-1 + u^-1
```

4.8 Incidence Algebras

class `sage.combinat.posets.incidence_algebras.IncidenceAlgebra (R, P, prefix='I')`
 Bases: `sage.combinat.free_module.CombinatorialFreeModule`

The incidence algebra of a poset.

Let P be a poset and R be a commutative unital associative ring. The *incidence algebra* I_P is the algebra of functions $\alpha: P \times P \rightarrow R$ such that $\alpha(x, y) = 0$ if $x \not\leq y$ where multiplication is given by convolution:

$$(\alpha * \beta)(x, y) = \sum_{x \leq k \leq y} \alpha(x, k) \beta(k, y).$$

This has a natural basis given by indicator functions for the interval $[a, b]$, i.e. $X_{a,b}(x, y) = \delta_{ax} \delta_{by}$. The incidence algebra is a unital algebra with the identity given by the Kronecker delta $\delta(x, y) = \delta_{xy}$. The Möbius function of P is another element of I_P whose inverse is the ζ function of the poset (so $\zeta(x, y) = 1$ for every interval $[x, y]$).

Todo

Implement the incidence coalgebra.

REFERENCES:

• [Wikipedia article Incidence_algebra](#)

class `Element (M, x)`

Bases: `sage.combinat.free_module.CombinatorialFreeModuleElement`

An element of an incidence algebra.

is_unit ()

Return if `self` is a unit.

EXAMPLES:

```
sage: P = posets.BooleanLattice(2)
sage: I = P.incidence_algebra(QQ)
sage: mu = I.moebius()
sage: mu.is_unit()
True
sage: zeta = I.zeta()
sage: zeta.is_unit()
True
sage: x = mu - I.zeta() + I[2,2]
sage: x.is_unit()
False
sage: y = I.moebius() + I.zeta()
sage: y.is_unit()
True
```

This depends on the base ring:

```
sage: I = P.incidence_algebra(ZZ)
sage: y = I.moebius() + I.zeta()
sage: y.is_unit()
False
```

to_matrix ()

Return `self` as a matrix.

We define a matrix $M_{xy} = \alpha(x, y)$ for some element $\alpha \in I_P$ in the incidence algebra I_P and we order the elements $x, y \in P$ by some linear extension of P . This defines an algebra (iso)morphism; in particular, multiplication in the incidence algebra goes to matrix multiplication.

EXAMPLES:

```
sage: P = posets.BooleanLattice(2)
sage: I = P.incidence_algebra(QQ)
sage: I.moebius().to_matrix()
[ 1 -1 -1  1]
[ 0  1  0 -1]
[ 0  0  1 -1]
[ 0  0  0  1]
sage: I.zeta().to_matrix()
[1 1 1 1]
[0 1 0 1]
[0 0 1 1]
[0 0 0 1]
```

`IncidenceAlgebra`. **delta** ()

Return the element 1 in `self` (which is the Kronecker delta $\delta(x, y)$).

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: I = P.incidence_algebra(QQ)
sage: I.one()
I[0, 0] + I[1, 1] + I[2, 2] + I[3, 3] + I[4, 4] + I[5, 5]
```

```
+ I[6, 6] + I[7, 7] + I[8, 8] + I[9, 9] + I[10, 10]
+ I[11, 11] + I[12, 12] + I[13, 13] + I[14, 14] + I[15, 15]
```

`IncidenceAlgebra.mobius (*args, **kws)`

Deprecated: Use `moebius()` instead. See [trac ticket #19855](#) for details.

`IncidenceAlgebra.moebius ()`

Return the Möbius function of `self`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(2)
sage: I = P.incidence_algebra(QQ)
sage: I.moebius()
I[0, 0] - I[0, 1] - I[0, 2] + I[0, 3] + I[1, 1]
- I[1, 3] + I[2, 2] - I[2, 3] + I[3, 3]
```

`IncidenceAlgebra.one ()`

Return the element 1 in `self` (which is the Kronecker delta $\delta(x, y)$).

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: I = P.incidence_algebra(QQ)
sage: I.one()
I[0, 0] + I[1, 1] + I[2, 2] + I[3, 3] + I[4, 4] + I[5, 5]
+ I[6, 6] + I[7, 7] + I[8, 8] + I[9, 9] + I[10, 10]
+ I[11, 11] + I[12, 12] + I[13, 13] + I[14, 14] + I[15, 15]
```

`IncidenceAlgebra.poset ()`

Return the defining poset of `self`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: I = P.incidence_algebra(QQ)
sage: I.poset()
Finite lattice containing 16 elements
sage: I.poset() == P
True
```

`IncidenceAlgebra.product_on_basis (A, B)`

Return the product of basis elements indexed by `A` and `B`.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: I = P.incidence_algebra(QQ)
sage: I.product_on_basis((1, 3), (3, 11))
I[1, 11]
sage: I.product_on_basis((1, 3), (2, 2))
0
```

`IncidenceAlgebra.reduced_subalgebra (prefix='R')`

Return the reduced incidence subalgebra.

EXAMPLES:

```

sage: P = posets.BooleanLattice(4)
sage: I = P.incidence_algebra(QQ)
sage: I.reduced_subalgebra()
Reduced incidence algebra of Finite lattice containing 16 elements
over Rational Field

```

`IncidenceAlgebra.some_elements()`

Return a list of elements of `self`.

EXAMPLES:

```

sage: P = posets.BooleanLattice(1)
sage: I = P.incidence_algebra(QQ)
sage: I.some_elements()
[2*I[0, 0] + 2*I[0, 1] + 3*I[1, 1],
 I[0, 0] - I[0, 1] + I[1, 1],
 I[0, 0] + I[0, 1] + I[1, 1]]

```

`IncidenceAlgebra.zeta()`

Return the ζ function in `self`.

The ζ function on a poset P is given by

$$\zeta(x, y) = \begin{cases} 1 & x \leq y, \\ 0 & x \not\leq y. \end{cases}$$

EXAMPLES:

```

sage: P = posets.BooleanLattice(4)
sage: I = P.incidence_algebra(QQ)
sage: I.zeta() * I.moebius() == I.one()
True

```

class `sage.combinat.posets.incidence_algebras.ReducedIncidenceAlgebra` (I , *pre-fix*='R')

Bases: `sage.combinat.free_module.CombinatorialFreeModule`

The reduced incidence algebra of a poset.

The reduced incidence algebra R_P is a subalgebra of the incidence algebra I_P where $\alpha(x, y) = \alpha(x', y')$ when $[x, y]$ is isomorphic to $[x', y']$ as posets. Thus the delta, Möbius, and zeta functions are all elements of R_P .

class `Element` (M, x)

Bases: `sage.combinat.free_module.CombinatorialFreeModuleElement`

An element of a reduced incidence algebra.

is_unit ()

Return if `self` is a unit.

EXAMPLES:

```

sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: x = R.an_element()
sage: x.is_unit()
True

```

lift ()

Return the lift of `self` to the ambient space.

EXAMPLES:

```

sage: P = posets.BooleanLattice(2)
sage: I = P.incidence_algebra(QQ)
sage: R = I.reduced_subalgebra()
sage: x = R.an_element(); x
2*R[(0, 0)] + 2*R[(0, 1)] + 3*R[(0, 3)]
sage: x.lift()
2*I[0, 0] + 2*I[0, 1] + 2*I[0, 2] + 3*I[0, 3] + 2*I[1, 1]
+ 2*I[1, 3] + 2*I[2, 2] + 2*I[2, 3] + 2*I[3, 3]

```

to_matrix()

Return self as a matrix.

EXAMPLES:

```

sage: P = posets.BooleanLattice(2)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: mu = R.moebius()
sage: mu.to_matrix()
[ 1 -1 -1  1]
[ 0  1  0 -1]
[ 0  0  1 -1]
[ 0  0  0  1]

```

ReducedIncidenceAlgebra.delta()

Return the Kronecker delta function in self.

EXAMPLES:

```

sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.delta()
R[(0, 0)]

```

ReducedIncidenceAlgebra.lift()

Return the lift morphism from self to the ambient space.

EXAMPLES:

```

sage: P = posets.BooleanLattice(2)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.lift
Generic morphism:
  From: Reduced incidence algebra of Finite lattice containing 4 elements_
  ↪over Rational Field
  To:   Incidence algebra of Finite lattice containing 4 elements over_
  ↪Rational Field
sage: R.an_element() - R.one()
R[(0, 0)] + 2*R[(0, 1)] + 3*R[(0, 3)]
sage: R.lift(R.an_element() - R.one())
I[0, 0] + 2*I[0, 1] + 2*I[0, 2] + 3*I[0, 3] + I[1, 1]
+ 2*I[1, 3] + I[2, 2] + 2*I[2, 3] + I[3, 3]

```

ReducedIncidenceAlgebra.mobius(*args, **kws)Deprecated: Use `moebius()` instead. See [trac ticket #19855](#) for details.**ReducedIncidenceAlgebra.moebius()**

Return the Möbius function of self.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.moebius()
R[(0, 0)] - R[(0, 1)] + R[(0, 3)] - R[(0, 7)] + R[(0, 15)]
```

ReducedIncidenceAlgebra.**one_basis** ()

Return the index of the element 1 in self.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.one_basis()
(0, 0)
```

ReducedIncidenceAlgebra.**poset** ()

Return the defining poset of self.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.poset()
Finite lattice containing 16 elements
sage: R.poset() == P
True
```

ReducedIncidenceAlgebra.**some_elements** ()

Return a list of elements of self.

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.some_elements()
[2*R[(0, 0)] + 2*R[(0, 1)] + 3*R[(0, 3)],
 R[(0, 0)] - R[(0, 1)] + R[(0, 3)] - R[(0, 7)] + R[(0, 15)],
 R[(0, 0)] + R[(0, 1)] + R[(0, 3)] + R[(0, 7)] + R[(0, 15)]]
```

ReducedIncidenceAlgebra.**zeta** ()

Return the ζ function in self.

The ζ function on a poset P is given by

$$\zeta(x, y) = \begin{cases} 1 & x \leq y, \\ 0 & x \not\leq y. \end{cases}$$

EXAMPLES:

```
sage: P = posets.BooleanLattice(4)
sage: R = P.incidence_algebra(QQ).reduced_subalgebra()
sage: R.zeta()
R[(0, 0)] + R[(0, 1)] + R[(0, 3)] + R[(0, 7)] + R[(0, 15)]
```

4.9 Group algebras

This module implements group algebras for arbitrary groups over arbitrary commutative rings.

EXAMPLES:

```
sage: D4 = DihedralGroup(4)
sage: kD4 = GroupAlgebra(D4, GF(7))
sage: a = kD4.an_element(); a
() + 4*(1,2,3,4) + 2*(1,4)(2,3)
sage: a * a
5*() + (2,4) + (1,2,3,4) + (1,3) + 2*(1,3)(2,4) + 4*(1,4)(2,3)
```

Given the group and the base ring, the corresponding group algebra is unique:

```
sage: A = GroupAlgebra(GL(3, QQ), ZZ)
sage: B = GroupAlgebra(GL(3, QQ), ZZ)
sage: A is B
True
sage: C = GroupAlgebra(GL(3, QQ), QQ)
sage: A == C
False
```

As long as there is no natural map from the group to the base ring, you can easily convert elements of the group to the group algebra:

```
sage: A = GroupAlgebra(DihedralGroup(2), ZZ)
sage: g = DihedralGroup(2).gen(0); g
(3,4)
sage: A(g)
(3,4)
sage: A(2) * g
2*(3,4)
```

Since there is a natural inclusion from the dihedral group D_2 of order 4 into the symmetric group S_4 of order $4!$, and since there is a natural map from the integers to the rationals, there is a natural map from $\mathbf{Z}[D_2]$ to $\mathbf{Q}[S_4]$:

```
sage: A = GroupAlgebra(DihedralGroup(2), ZZ)
sage: B = GroupAlgebra(SymmetricGroup(4), QQ)
sage: a = A.an_element(); a
() + 3*(3,4) + 3*(1,2)
sage: b = B.an_element(); b
() + 2*(1,2) + 4*(1,2,3,4)
sage: B(a)
() + 3*(3,4) + 3*(1,2)
sage: a * b # a is automatically converted to an element of B
7*() + 3*(3,4) + 5*(1,2) + 6*(1,2)(3,4) + 12*(1,2,3) + 4*(1,2,3,4) + 12*(1,3,4)
sage: parent(a * b)
Group algebra of group "Symmetric group of order 4! as a permutation group" over base_
↪ring Rational Field

sage: G = GL(3, GF(7))
sage: ZG = GroupAlgebra(G)
sage: c, d = G.random_element(), G.random_element()
sage: zc, zd = ZG(c), ZG(d)
sage: zc * d == zc * zd # d is automatically converted to an element of ZG
True
```


There is no obvious map in the other direction, though:

```
sage: A(b)
Traceback (most recent call last):
...
TypeError: Don't know how to create an element of Group algebra of group
"Dihedral group of order 4 as a permutation group" over base ring Integer
Ring from () + 2*(1,2) + 4*(1,2,3,4)
```

Group algebras have the structure of Hopf algebras:

```
sage: a = kD4.an_element(); a
() + 4*(1,2,3,4) + 2*(1,4)(2,3)
sage: a.antipode()
() + 4*(1,4,3,2) + 2*(1,4)(2,3)
sage: a.coproduct()
() # () + 4*(1,2,3,4) # (1,2,3,4) + 2*(1,4)(2,3) # (1,4)(2,3)
```

Note: As alluded to above, it is problematic to make group algebras fit nicely with Sage's coercion model. The problem is that (for example) if G is the additive group $(\mathbb{Z}, +)$, and $R = \mathbb{Z}[G]$ is its group ring, then the integer 2 can be coerced into R in two ways – via G , or via the base ring – and *the answers are different*. In practice we get around this by preventing elements of a group H from coercing automatically into a group ring $k[G]$ if H coerces into both k and G . This is unfortunate, but it seems like the most sensible solution in this ambiguous situation.

AUTHOR:

- David Loeffler (2008-08-24): initial version
- Martin Raum (2009-08): update to use new coercion model – see [trac ticket #6670](#).
- John Palmieri (2011-07): more updates to coercion, categories, etc., group algebras constructed using `CombinatorialFreeModule` – see [trac ticket #6670](#).

class `sage.algebras.group_algebra.GroupAlgebra (group, base_ring=Integer Ring)`

Bases: `sage.combinat.free_module.CombinatorialFreeModule`

Create the given group algebra.

INPUT:

- `group`, a group
- `base_ring` (optional, default \mathbb{Z}), a commutative ring

OUTPUT:

– a `GroupAlgebra` instance.

EXAMPLES:

```
sage: GroupAlgebra(GL(3, GF(7)))
Group algebra of group "General Linear Group of degree 3 over Finite Field of
↳size 7" over base ring Integer Ring
sage: GroupAlgebra(GL(3, GF(7)), QQ)
Group algebra of group "General Linear Group of degree 3 over Finite Field of
↳size 7" over base ring Rational Field
sage: GroupAlgebra(1)
Traceback (most recent call last):
...
TypeError: "1" is not a group
```

```
sage: GroupAlgebra(SU(2, GF(4, 'a')), IntegerModRing(12)).category()
Category of finite dimensional group algebras over Ring of integers modulo 12
sage: GroupAlgebra(KleinFourGroup()) is GroupAlgebra(KleinFourGroup())
True
```

The one of the group indexes the one of this algebra:

```
sage: A = GroupAlgebra(DihedralGroup(6), QQ)
sage: A.one_basis()
()
sage: A.one()
()
```

The product of two basis elements is induced by the product of the corresponding elements of the group:

```
sage: A = GroupAlgebra(DihedralGroup(3), QQ)
sage: (a, b) = A._group.gens()
sage: a*b
(1,2)
sage: A.product_on_basis(a, b)
(1,2)
```

The basis elements are group-like for the coproduct: $\Delta(g) = g \otimes g$:

```
sage: A = GroupAlgebra(DihedralGroup(3), QQ)
sage: (a, b) = A._group.gens()
sage: A.coproduct_on_basis(a)
(1,2,3) # (1,2,3)
```

The counit on the basis elements is 1:

```
sage: A = GroupAlgebra(DihedralGroup(6), QQ)
sage: (a, b) = A._group.gens()
sage: A.counit_on_basis(a)
1
```

The antipode on basis elements is given by $\chi(g) = g^{-1}$:

```
sage: A = GroupAlgebra(DihedralGroup(3), QQ)
sage: (a, b) = A._group.gens(); a
(1,2,3)
sage: A.antipode_on_basis(a)
(1,3,2)
```

algebra_generators ()

The generators of this algebra, as per `Algebras.ParentMethods.algebra_generators ()`.

They correspond to the generators of the group.

EXAMPLES:

```
sage: A = GroupAlgebra(DihedralGroup(3), QQ); A
Group algebra of group "Dihedral group of order 6 as a permutation group"
↳over base ring Rational Field
sage: A.algebra_generators()
Finite family {(1,3): (1,3), (1,2,3): (1,2,3)}
```

construction ()

EXAMPLES:

```
sage: A = GroupAlgebra(KleinFourGroup(), QQ)
sage: A.construction()
(GroupAlgebraFunctor, Rational Field)
```

gen (i=0)

EXAMPLES:

```
sage: A = GroupAlgebra(GL(3, GF(7)))
sage: A.gen(0)
[3 0 0]
[0 1 0]
[0 0 1]
```

gens ()The generators of this algebra, as per `Algebras.ParentMethods.algebra_generators()`.

They correspond to the generators of the group.

EXAMPLES:

```
sage: A = GroupAlgebra(DihedralGroup(3), QQ); A
Group algebra of group "Dihedral group of order 6 as a permutation group"
↳over base ring Rational Field
sage: A.algebra_generators()
Finite family {(1,3): (1,3), (1,2,3): (1,2,3)}
```

group ()

Return the group of this group algebra.

EXAMPLES:

```
sage: GroupAlgebra(GL(3, GF(11))).group()
General Linear Group of degree 3 over Finite Field of size 11
sage: GroupAlgebra(SymmetricGroup(10)).group()
Symmetric group of order 10! as a permutation group
```

is_commutative ()Return True if self is a commutative ring. True if and only if `self.group()` is abelian.

EXAMPLES:

```
sage: GroupAlgebra(SymmetricGroup(2)).is_commutative()
True
sage: GroupAlgebra(SymmetricGroup(3)).is_commutative()
False
```

is_exact ()Return True if elements of self have exact representations, which is true of self if and only if it is true of `self.group()` and `self.base_ring()`.

EXAMPLES:

```
sage: GroupAlgebra(GL(3, GF(7))).is_exact()
True
sage: GroupAlgebra(GL(3, GF(7)), RR).is_exact()
False
```

```
sage: GroupAlgebra(GL(3, pAdicRing(7))).is_exact() # not implemented,
↪correctly (not my fault)!
False
```

is_field (*proof=True*)

Return True if self is a field. This is always false unless self.group() is trivial and self.base_ring() is a field.

EXAMPLES:

```
sage: GroupAlgebra(SymmetricGroup(2)).is_field()
False
sage: GroupAlgebra(SymmetricGroup(1)).is_field()
False
sage: GroupAlgebra(SymmetricGroup(1), QQ).is_field()
True
```

is_finite ()

Return True if self is finite, which is true if and only if self.group() and self.base_ring() are both finite.

EXAMPLES:

```
sage: GroupAlgebra(SymmetricGroup(2), IntegerModRing(10)).is_finite()
True
sage: GroupAlgebra(SymmetricGroup(2)).is_finite()
False
sage: GroupAlgebra(AbelianGroup(1), IntegerModRing(10)).is_finite()
False
```

is_integral_domain (*proof=True*)

Return True if self is an integral domain.

This is false unless self.base_ring() is an integral domain, and even then it is false unless self.group() has no nontrivial elements of finite order. I don't know if this condition suffices, but it obviously does if the group is abelian and finitely generated.

EXAMPLES:

```
sage: GroupAlgebra(SymmetricGroup(2)).is_integral_domain()
False
sage: GroupAlgebra(SymmetricGroup(1)).is_integral_domain()
True
sage: GroupAlgebra(SymmetricGroup(1), IntegerModRing(4)).is_integral_domain()
False
sage: GroupAlgebra(AbelianGroup(1)).is_integral_domain()
True
sage: GroupAlgebra(AbelianGroup(2, [0,2])).is_integral_domain()
False
sage: GroupAlgebra(GL(2, ZZ)).is_integral_domain() # not implemented
False
```

ngens ()

Return the number of generators.

EXAMPLES:

```
sage: GroupAlgebra(SL2Z).ngens()
2
```

```
sage: GroupAlgebra(DihedralGroup(4), RR).ngens()
2
```

random_element (*n=2*)

Return a ‘random’ element of self.

INPUT:

- *n* – integer (optional, default 2), number of summands

Algorithm: return a sum of *n* terms, each of which is formed by multiplying a random element of the base ring by a random element of the group.

EXAMPLES:

```
sage: GroupAlgebra(DihedralGroup(6), QQ).random_element()
-1/95*() - 1/2*(1,4)(2,5)(3,6)
sage: GroupAlgebra(SU(2, 13), QQ).random_element(1)
1/2*[      0 4*a + 11]
[2*a + 12      4]
```

class sage.algebras.group_algebra. **GroupAlgebraFunctor** (*group*)

Bases: sage.categories.pushout.ConstructionFunctor

For a fixed group, a functor sending a commutative ring to the corresponding group algebra.

INPUT :

- *group* – the group associated to each group algebra under consideration.

EXAMPLES:

```
sage: from sage.algebras.group_algebra import GroupAlgebraFunctor
sage: F = GroupAlgebraFunctor(KleinFourGroup())
sage: loads(dumps(F)) == F
True
sage: GroupAlgebra(SU(2, GF(4, 'a')), IntegerModRing(12)).category()
Category of finite dimensional group algebras over Ring of integers modulo 12
```

group ()

Return the group which is associated to this functor.

EXAMPLES:

```
sage: from sage.algebras.group_algebra import GroupAlgebraFunctor
sage: GroupAlgebraFunctor(CyclicPermutationGroup(17)).group() ==
↪CyclicPermutationGroup(17)
True
```

4.10 Möbius Algebras

```
class sage.combinat.posets.moebius_algebra. BasisAbstract ( R, basis_keys, element_class=None, category=None, prefix='B', **kws )
Bases: sage.combinat.free_module.CombinatorialFreeModule, sage.misc.bindable_class.BindableClass
```

Abstract base class for a basis.

class `sage.combinat.posets.moebius_algebra.MoebiusAlgebra (R, L)`
 Bases: `sage.structure.parent.Parent`, `sage.structure.unique_representation.UniqueRepresentation`

The Möbius algebra of a lattice.

Let L be a lattice. The *Möbius algebra* M_L was originally constructed by Solomon [Solomon67] and has a natural basis $\{E_x \mid x \in L\}$ with multiplication given by $E_x \cdot E_y = E_{x \vee y}$. Moreover this has a basis given by orthogonal idempotents $\{I_x \mid x \in L\}$ (so $I_x I_y = \delta_{xy} I_x$ where δ is the Kronecker delta) related to the natural basis by

$$I_x = \sum_{x \leq y} \mu_L(x, y) E_y,$$

where μ_L is the Möbius function of L .

Note: We use the join \vee for our multiplication, whereas [Greene73] and [Etienne98] define the Möbius algebra using the meet \wedge . This is done for compatibility with *QuantumMoebiusAlgebra*.

REFERENCES:

class `E (M, prefix='E')`

Bases: `sage.combinat.posets.moebius_algebra.BasisAbstract`

The natural basis of a Möbius algebra.

Let E_x and E_y be basis elements of M_L for some lattice L . Multiplication is given by $E_x E_y = E_{x \vee y}$.

one ()

Return the element 1 of self.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: E = L.moebius_algebra(QQ).E()
sage: E.one()
E[0]
```

product_on_basis (x, y)

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

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: E = L.moebius_algebra(QQ).E()
sage: E.product_on_basis(5, 14)
E[15]
sage: E.product_on_basis(2, 8)
E[10]
```

class `MoebiusAlgebra.I (M, prefix='I')`

Bases: `sage.combinat.posets.moebius_algebra.BasisAbstract`

The (orthogonal) idempotent basis of a Möbius algebra.

Let I_x and I_y be basis elements of M_L for some lattice L . Multiplication is given by $I_x I_y = \delta_{xy} I_x$ where δ_{xy} is the Kronecker delta.

one ()

Return the element 1 of self.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: I = L.moebius_algebra(QQ).I()
sage: I.one()
I[0] + I[1] + I[2] + I[3] + I[4] + I[5] + I[6] + I[7] + I[8]
+ I[9] + I[10] + I[11] + I[12] + I[13] + I[14] + I[15]
```

product_on_basis (*x*, *y*)

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

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: I = L.moebius_algebra(QQ).I()
sage: I.product_on_basis(5, 14)
0
sage: I.product_on_basis(2, 2)
I[2]
```

MoebiusAlgebra.a_realization ()

Return a particular realization of self (the *B*-basis).

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: M = L.moebius_algebra(QQ)
sage: M.a_realization()
Moebius algebra of Finite lattice containing 16 elements
over Rational Field in the natural basis
```

MoebiusAlgebra.lattice ()

Return the defining lattice of self.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: M = L.moebius_algebra(QQ)
sage: M.lattice()
Finite lattice containing 16 elements
sage: M.lattice() == L
True
```

class sage.combinat.posets.moebius_algebra. **MoebiusAlgebraBases** (*parent_with_realization*)
 Bases: sage.categories.realizations.Category_realization_of_parent

The category of bases of a Möbius algebra.

INPUT:

- *base* – a Möbius algebra

class **ElementMethods**

class **MoebiusAlgebraBases.ParentMethods**

one ()

Return the element 1 of self.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: C = L.quantum_moebius_algebra().C()
sage: all(C.one() * b == b for b in C.basis())
True
```

product_on_basis (x, y)

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

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: C = L.quantum_moebius_algebra().C()
sage: C.product_on_basis(5, 14)
q^3*C[15]
sage: C.product_on_basis(2, 8)
q^4*C[10]
```

MoebiusAlgebraBases.super_categories ()

The super categories of self.

EXAMPLES:

```
sage: from sage.combinat.posets.moebius_algebra import MoebiusAlgebraBases
sage: M = posets.BooleanLattice(4).moebius_algebra(QQ)
sage: bases = MoebiusAlgebraBases(M)
sage: bases.super_categories()
[Category of finite dimensional commutative algebras with basis over Rational_
↪Field,
Category of realizations of Moebius algebra of Finite lattice
containing 16 elements over Rational Field]
```

class sage.combinat.posets.moebius_algebra. **QuantumMoebiusAlgebra** (L, q=None)

Bases: sage.structure.parent.Parent, sage.structure.unique_representation.UniqueRepresentation

The quantum Möbius algebra of a lattice.

Let L be a lattice, and we define the *quantum Möbius algebra* $M_L(q)$ as the algebra with basis $\{E_x \mid x \in L\}$ with multiplication given by

$$E_x E_y = \sum_{z \geq a \geq x \vee y} \mu_L(a, z) q^{\text{crk } a} E_z,$$

where μ_L is the Möbius function of L and crk is the corank function (i.e., $\text{crk } a = \text{rank } L - \text{rank } a$). At $q = 1$, this reduces to the multiplication formula originally given by Solomon.

class C (M, prefix='C')

Bases: sage.combinat.posets.moebius_algebra.BasisAbstract

The characteristic basis of a quantum Möbius algebra.

The characteristic basis $\{C_x \mid x \in L\}$ of M_L for some lattice L is defined by

$$C_x = \sum_{a \geq x} P(F^x; q) E_a,$$

where $F^x = \{y \in L \mid y \geq x\}$ is the principal order filter of x and $P(F^x; q)$ is the characteristic polynomial of the (sub)poset F^x .

class QuantumMöbiusAlgebra . **E** (*M*, *prefix*='E')

Bases: *sage.combinat.posets.moebius_algebra.BasisAbstract*

The natural basis of a quantum Möbius algebra.

Let E_x and E_y be basis elements of M_L for some lattice L . Multiplication is given by

$$E_x E_y = \sum_{z \geq a \geq x \vee y} \mu_L(a, z) q^{\text{crk } a} E_z,$$

where μ_L is the Möbius function of L and crk is the corank function (i.e., $\text{crk } a = \text{rank } L - \text{rank } a$).

one ()

Return the element 1 of self .

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: E = L.quantum_moebius_algebra().E()
sage: all(E.one() * b == b for b in E.basis())
True
```

product_on_basis (*x*, *y*)

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

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: E = L.quantum_moebius_algebra().E()
sage: E.product_on_basis(5, 14)
E[15]
sage: E.product_on_basis(2, 8)
q^2*E[10] + (q-q^2)*E[11] + (q-q^2)*E[14] + (1-2*q+q^2)*E[15]
```

class QuantumMöbiusAlgebra . **KL** (*M*, *prefix*='KL')

Bases: *sage.combinat.posets.moebius_algebra.BasisAbstract*

The Kazhdan-Lusztig basis of a quantum Möbius algebra.

The Kazhdan-Lusztig basis $\{B_x \mid x \in L\}$ of M_L for some lattice L is defined by

$$B_x = \sum_{y \geq x} P_{x,y}(q) E_a,$$

where $P_{x,y}(q)$ is the Kazhdan-Lusztig polynomial of L , following the definition given in [EPW14].

EXAMPLES:

We construct some examples of Proposition 4.5 of [EPW14]:

```
sage: M = posets.BooleanLattice(4).quantum_moebius_algebra()
sage: KL = M.KL()
sage: KL[4] * KL[5]
(q^2+q^3)*KL[5] + (q+2*q^2+q^3)*KL[7] + (q+2*q^2+q^3)*KL[13]
+ (1+3*q+3*q^2+q^3)*KL[15]
sage: KL[4] * KL[15]
(1+3*q+3*q^2+q^3)*KL[15]
sage: KL[4] * KL[10]
(q+3*q^2+3*q^3+q^4)*KL[14] + (1+4*q+6*q^2+4*q^3+q^4)*KL[15]
```

QuantumMoebiusAlgebra.**a_realization** ()
 Return a particular realization of self (the B -basis).

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: M = L.quantum_moebius_algebra()
sage: M.a_realization()
Quantum Moebius algebra of Finite lattice containing 16 elements
with q=q over Univariate Laurent Polynomial Ring in q
over Integer Ring in the natural basis
```

QuantumMoebiusAlgebra.**lattice** ()
 Return the defining lattice of self.

EXAMPLES:

```
sage: L = posets.BooleanLattice(4)
sage: M = L.quantum_moebius_algebra()
sage: M.lattice()
Finite lattice containing 16 elements
sage: M.lattice() == L
True
```

4.11 Nil-Coxeter Algebra

class sage.algebras.nil_coxeter_algebra.**NilCoxeterAlgebra** (W , *base_ring=Rational Field, prefix='u'*)
 Bases: *sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T*

Construct the Nil-Coxeter algebra of given type. This is the algebra with generators u_i for every node i of the corresponding Dynkin diagram. It has the usual braid relations (from the Weyl group) as well as the quadratic relation $u_i^2 = 0$.

INPUT:

- W – a Weyl group

OPTIONAL ARGUMENTS:

- *base_ring* – a ring (default is the rational numbers)
- *prefix* – a label for the generators (default “u”)

EXAMPLES:

```
sage: U = NilCoxeterAlgebra(WeylGroup(['A', 3, 1]))
sage: u0, u1, u2, u3 = U.algebra_generators()
sage: u1*u1
0
sage: u2*u1*u2 == u1*u2*u1
True
sage: U.an_element()
u[0,1,2,3] + 2*u[0] + 3*u[1] + 1
```

homogeneous_generator_noncommutative_variables (r)

Give the r^{th} homogeneous function inside the Nil-Coxeter algebra. In finite type A this is the sum of all decreasing elements of length r . In affine type A this is the sum of all cyclically decreasing elements of length r . This is only defined in finite type A , B and affine types $A^{(1)}$, $B^{(1)}$, $C^{(1)}$, $D^{(1)}$.

INPUT:

- r – a positive integer at most the rank of the Weyl group

EXAMPLES:

```
sage: U = NilCoxeterAlgebra(WeylGroup(['A', 3, 1]))
sage: U.homogeneous_generator_noncommutative_variables(2)
u[1, 0] + u[2, 0] + u[0, 3] + u[3, 2] + u[3, 1] + u[2, 1]

sage: U = NilCoxeterAlgebra(WeylGroup(['B', 4]))
sage: U.homogeneous_generator_noncommutative_variables(2)
u[1, 2] + u[2, 1] + u[3, 1] + u[4, 1] + u[2, 3] + u[3, 2] + u[4, 2] + u[3, 4] + u[4, 3]

sage: U = NilCoxeterAlgebra(WeylGroup(['C', 3]))
sage: U.homogeneous_generator_noncommutative_variables(2)
Traceback (most recent call last):
...
AssertionError: Analogue of symmetric functions in noncommutative variables_
↪ is not defined in type ['C', 3]
```

homogeneous_noncommutative_variables (la)

Give the homogeneous function indexed by la , viewed inside the Nil-Coxeter algebra. This is only defined in finite type A , B and affine types $A^{(1)}$, $B^{(1)}$, $C^{(1)}$, $D^{(1)}$.

INPUT:

- la – a partition with first part bounded by the rank of the Weyl group

EXAMPLES:

```
sage: U = NilCoxeterAlgebra(WeylGroup(['B', 2, 1]))
sage: U.homogeneous_noncommutative_variables([2, 1])
u[1, 2, 0] + 2*u[2, 1, 0] + u[0, 2, 0] + u[0, 2, 1] + u[1, 2, 1] + u[2, 1, 2] + u[2, 0, 2]
↪ + u[1, 0, 2]
```

k_schur_noncommutative_variables (la)

In type $A^{(1)}$ this is the k -Schur function in noncommutative variables defined by Thomas Lam [Lam2005].

This function is currently only defined in type $A^{(1)}$.

INPUT:

- la – a partition with first part bounded by the rank of the Weyl group

EXAMPLES:

```
sage: A = NilCoxeterAlgebra(WeylGroup(['A', 3, 1]))
sage: A.k_schur_noncommutative_variables([2, 2])
u[0, 3, 1, 0] + u[3, 1, 2, 0] + u[1, 2, 0, 1] + u[3, 2, 0, 3] + u[2, 0, 3, 1] + u[2, 3, 1, 2]
```

4.12 Orlik-Solomon Algebras

class sage.algebras.orlik_solomon. **OrlikSolomonAlgebra** ($R, M, ordering=None$)

Bases: sage.combinat.free_module.CombinatorialFreeModule

An Orlik-Solomon algebra.

Let R be a commutative ring. Let M be a matroid with ground set X . Let $C(M)$ denote the set of circuits of M . Let E denote the exterior algebra over R generated by $\{e_x \mid x \in X\}$. The *Orlik-Solomon ideal* $J(M)$ is

the ideal of E generated by

$$\partial e_S := \sum_{i=1}^t (-1)^{i-1} e_{j_1} \wedge e_{j_2} \wedge \cdots \wedge \widehat{e_{j_i}} \wedge \cdots \wedge e_{j_t}$$

for all $S = \{j_1 < j_2 < \cdots < j_t\} \in C(M)$, where $\widehat{e_{j_i}}$ means that the term e_{j_i} is being omitted. The notation ∂e_S is not a coincidence, as ∂e_S is actually the image of $e_S := e_{j_1} \wedge e_{j_2} \wedge \cdots \wedge e_{j_t}$ under the unique derivation ∂ of E which sends all e_x to 1.

It is easy to see that $\partial e_S \in J(M)$ not only for circuits S , but also for any dependent set S of M . Moreover, every dependent set S of M satisfies $e_S \in J(M)$.

The *Orlik-Solomon algebra* $A(M)$ is the quotient $E/J(M)$. This is a graded finite-dimensional skew-commutative R -algebra. Fix some ordering on X ; then, the NBC sets of M (that is, the subsets of X containing no broken circuit of M) form a basis of $A(M)$. (Here, a *broken circuit* of M is defined to be the result of removing the smallest element from a circuit of M .)

In the current implementation, the basis of $A(M)$ is indexed by the NBC sets, which are implemented as frozensets.

INPUT:

- **R** – the base ring
- **M** – the defining matroid
- **ordering** – (optional) an ordering of the ground set

EXAMPLES:

We create the Orlik-Solomon algebra of the uniform matroid $U(3, 4)$ and do some basic computations:

```
sage: M = matroids.Uniform(3, 4)
sage: OS = M.orlik_solomon_algebra(QQ)
sage: OS.dimension()
14
sage: G = OS.algebra_generators()
sage: M.broken_circuits()
frozenset({frozenset({1, 2, 3})})
sage: G[1] * G[2] * G[3]
OS{0, 1, 2} - OS{0, 1, 3} + OS{0, 2, 3}
```

REFERENCES:

- [Wikipedia article Arrangement_of_hyperplanes#The_Orlik-Solomon_algebra](#)
- [CE2001]

algebra_generators ()

Return the algebra generators of `self`.

These form a family indexed by the ground set X of M . For each $x \in X$, the x -th element is e_x .

EXAMPLES:

```
sage: M = matroids.Uniform(2, 2)
sage: OS = M.orlik_solomon_algebra(QQ)
sage: OS.algebra_generators()
Finite family {0: OS{0}, 1: OS{1}}

sage: M = matroids.Uniform(1, 2)
sage: OS = M.orlik_solomon_algebra(QQ)
```

```

sage: OS.algebra_generators()
Finite family {0: OS{0}, 1: OS{0}}

sage: M = matroids.Uniform(1, 3)
sage: OS = M.orlik_solomon_algebra(QQ)
sage: OS.algebra_generators()
Finite family {0: OS{0}, 1: OS{0}, 2: OS{0}}

```

degree_on_basis (m)

Return the degree of the basis element indexed by m .

EXAMPLES:

```

sage: M = matroids.Wheel(3)
sage: OS = M.orlik_solomon_algebra(QQ)
sage: OS.degree_on_basis(frozenset([1]))
1
sage: OS.degree_on_basis(frozenset([0, 2, 3]))
3

```

one_basis ()

Return the index of the basis element corresponding to 1 in `self`.

EXAMPLES:

```

sage: M = matroids.Wheel(3)
sage: OS = M.orlik_solomon_algebra(QQ)
sage: OS.one_basis() == frozenset([])
True

```

product_on_basis (a, b)

Return the product in `self` of the basis elements indexed by a and b .

EXAMPLES:

```

sage: M = matroids.Wheel(3)
sage: OS = M.orlik_solomon_algebra(QQ)
sage: OS.product_on_basis(frozenset([2]), frozenset([3,4]))
OS{0, 1, 2} - OS{0, 1, 4} + OS{0, 2, 3} + OS{0, 3, 4}

```

```

sage: G = OS.algebra_generators()
sage: prod(G)
0
sage: G[2] * G[4]
-OS{1, 2} + OS{1, 4}
sage: G[3] * G[4] * G[2]
OS{0, 1, 2} - OS{0, 1, 4} + OS{0, 2, 3} + OS{0, 3, 4}
sage: G[2] * G[3] * G[4]
OS{0, 1, 2} - OS{0, 1, 4} + OS{0, 2, 3} + OS{0, 3, 4}
sage: G[3] * G[2] * G[4]
-OS{0, 1, 2} + OS{0, 1, 4} - OS{0, 2, 3} - OS{0, 3, 4}

```

subset_image (S)

Return the element e_S of $A(M)$ (`== self`) corresponding to a subset S of the ground set of M .

INPUT:

- S – a frozenset which is a subset of the ground set of M

EXAMPLES:

```
sage: M = matroids.Wheel(3)
sage: OS = M.orlik_solomon_algebra(QQ)
sage: BC = sorted(M.broken_circuits(), key=sorted)
sage: for bc in BC: (sorted(bc), OS.subset_image(bc))
([1, 3], -OS{0, 1} + OS{0, 3})
([1, 4, 5], OS{0, 1, 4} - OS{0, 1, 5} - OS{0, 3, 4} + OS{0, 3, 5})
([2, 3, 4], OS{0, 1, 2} - OS{0, 1, 4} + OS{0, 2, 3} + OS{0, 3, 4})
([2, 3, 5], OS{0, 2, 3} + OS{0, 3, 5})
([2, 4], -OS{1, 2} + OS{1, 4})
([2, 5], -OS{0, 2} + OS{0, 5})
([4, 5], -OS{3, 4} + OS{3, 5})

sage: M4 = matroids.CompleteGraphic(4)
sage: OS = M4.orlik_solomon_algebra(QQ)
sage: OS.subset_image(frozenset({2,3,4}))
OS{0, 2, 3} + OS{0, 3, 4}
```

An example of a custom ordering:

```
sage: G = Graph([[3, 4], [4, 1], [1, 2], [2, 3], [3, 5], [5, 6], [6, 3]])
sage: M = Matroid(G)
sage: s = [(5, 6), (1, 2), (3, 5), (2, 3), (1, 4), (3, 6), (3, 4)]
sage: sorted([sorted(c) for c in M.circuits()])
[[1, 2), (1, 4), (2, 3), (3, 4)],
 [(3, 5), (3, 6), (5, 6)]]
sage: OS = M.orlik_solomon_algebra(QQ, ordering=s)
sage: OS.subset_image(frozenset([]))
OS{}
sage: OS.subset_image(frozenset([(1,2), (3,4), (1,4), (2,3)]))
0
sage: OS.subset_image(frozenset([(2,3), (1,2), (3,4)]))
OS{(1, 2), (3, 4), (2, 3)}
sage: OS.subset_image(frozenset([(1,4), (3,4), (2,3), (3,6), (5,6)]))
-OS{(1, 2), (5, 6), (2, 3), (1, 4), (3, 6)}
+ OS{(1, 2), (5, 6), (3, 4), (1, 4), (3, 6)}
- OS{(1, 2), (5, 6), (3, 4), (2, 3), (3, 6)}
sage: OS.subset_image(frozenset([(1,4), (3,4), (2,3), (3,6), (3,5)]))
OS{(1, 2), (5, 6), (2, 3), (1, 4), (3, 5)}
- OS{(1, 2), (5, 6), (2, 3), (1, 4), (3, 6)}
+ OS{(1, 2), (5, 6), (3, 4), (1, 4), (3, 5)}
+ OS{(1, 2), (5, 6), (3, 4), (1, 4), (3, 6)}
- OS{(1, 2), (5, 6), (3, 4), (2, 3), (3, 5)}
- OS{(1, 2), (5, 6), (3, 4), (2, 3), (3, 6)}
```

4.13 Partition/Diagram Algebras

class sage.combinat.partition_algebra. **PartitionAlgebraElement_ak** (*M, x*)
 Bases: *sage.combinat.partition_algebra.PartitionAlgebraElement_generic*

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

class sage.combinat.partition_algebra. **PartitionAlgebraElement_bk** (*M, x*)
 Bases: *sage.combinat.partition_algebra.PartitionAlgebraElement_generic*

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

```
class sage.combinat.partition_algebra. PartitionAlgebraElement_generic ( M, x)
    Bases: sage.combinat.free_module.CombinatorialFreeModuleElement
```

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

```
class sage.combinat.partition_algebra. PartitionAlgebraElement_pk ( M, x)
    Bases: sage.combinat.partition_algebra.PartitionAlgebraElement_generic
```

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

```
class sage.combinat.partition_algebra. PartitionAlgebraElement_prk ( M, x)
    Bases: sage.combinat.partition_algebra.PartitionAlgebraElement_generic
```

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

```
class sage.combinat.partition_algebra. PartitionAlgebraElement_rk ( M, x)
    Bases: sage.combinat.partition_algebra.PartitionAlgebraElement_generic
```

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

```
class sage.combinat.partition_algebra. PartitionAlgebraElement_sk ( M, x)
    Bases: sage.combinat.partition_algebra.PartitionAlgebraElement_generic
```

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

```
class sage.combinat.partition_algebra. PartitionAlgebraElement_tk ( M, x)
    Bases: sage.combinat.partition_algebra.PartitionAlgebraElement_generic
```

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

```
class sage.combinat.partition_algebra. PartitionAlgebra_ak ( R, k, n, name=None)
    Bases: sage.combinat.partition_algebra.PartitionAlgebra_generic
```

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_ak(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

```
class sage.combinat.partition_algebra. PartitionAlgebra_bk ( R, k, n, name=None)
    Bases: sage.combinat.partition_algebra.PartitionAlgebra_generic
```

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_bk(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

```
class sage.combinat.partition_algebra. PartitionAlgebra_generic ( R, cclass, n, k,
                                                                name=None, pre-
                                                                fix=None)
    Bases: sage.combinat.combinatorial_algebra.CombinatorialAlgebra
```

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: s = PartitionAlgebra_sk(QQ, 3, 1)
sage: s == loads(dumps(s))
True
```

class `sage.combinat.partition_algebra.PartitionAlgebra_pk (R, k, n, name=None)`
 Bases: `sage.combinat.partition_algebra.PartitionAlgebra_generic`

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_pk(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

class `sage.combinat.partition_algebra.PartitionAlgebra_prk (R, k, n, name=None)`
 Bases: `sage.combinat.partition_algebra.PartitionAlgebra_generic`

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_prk(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

class `sage.combinat.partition_algebra.PartitionAlgebra_rk (R, k, n, name=None)`
 Bases: `sage.combinat.partition_algebra.PartitionAlgebra_generic`

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_rk(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

class `sage.combinat.partition_algebra.PartitionAlgebra_sk (R, k, n, name=None)`
 Bases: `sage.combinat.partition_algebra.PartitionAlgebra_generic`

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_sk(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

class `sage.combinat.partition_algebra.PartitionAlgebra_tk (R, k, n, name=None)`
 Bases: `sage.combinat.partition_algebra.PartitionAlgebra_generic`

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import *
sage: p = PartitionAlgebra_tk(QQ, 3, 1)
sage: p == loads(dumps(p))
True
```

`sage.combinat.partition_algebra.SetPartitionsAk (k)`
 Returns the combinatorial class of set partitions of type A_k .

EXAMPLES:

```

sage: A3 = SetPartitionsAk(3); A3
Set partitions of {1, ..., 3, -1, ..., -3}

sage: A3.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: A3.last() #random
{{-1}, {-2}, {3}, {1}, {-3}, {2}}
sage: A3.random_element() #random
{{1, 3, -3, -1}, {2, -2}}

sage: A3.cardinality()
203

sage: A2p5 = SetPartitionsAk(2.5); A2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block
sage: A2p5.cardinality()
52

sage: A2p5.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: A2p5.last() #random
{{-1}, {-2}, {2}, {3, -3}, {1}}
sage: A2p5.random_element() #random
{{-1}, {-2}, {3, -3}, {1, 2}}

```

class sage.combinat.partition_algebra. **SetPartitionsAk_k** (*k*)
 Bases: sage.combinat.set_partition.SetPartitions_set

Element

alias of *SetPartitionsXkElement*

class sage.combinat.partition_algebra. **SetPartitionsAkhalf_k** (*k*)
 Bases: sage.combinat.set_partition.SetPartitions_set

Element

alias of *SetPartitionsXkElement*

sage.combinat.partition_algebra. **SetPartitionsBk** (*k*)
 Returns the combinatorial class of set partitions of type B_k. These are the set partitions where every block has size 2.

EXAMPLES:

```

sage: B3 = SetPartitionsBk(3); B3
Set partitions of {1, ..., 3, -1, ..., -3} with block size 2

sage: B3.first() #random
{{2, -2}, {1, -3}, {3, -1}}
sage: B3.last() #random
{{1, 2}, {3, -2}, {-3, -1}}
sage: B3.random_element() #random
{{2, -1}, {1, -3}, {3, -2}}

sage: B3.cardinality()
15

sage: B2p5 = SetPartitionsBk(2.5); B2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and
↳with block size 2

```

```

sage: B2p5.first() #random
{{2, -1}, {3, -3}, {1, -2}}
sage: B2p5.last() #random
{{1, 2}, {3, -3}, {-1, -2}}
sage: B2p5.random_element() #random
{{2, -2}, {3, -3}, {1, -1}}

sage: B2p5.cardinality()
3

```

class sage.combinat.partition_algebra. **SetPartitionsBk_k** (*k*)
 Bases: *sage.combinat.partition_algebra.SetPartitionsAk_k*

cardinality ()

Returns the number of set partitions in B_k where k is an integer. This is given by $(2k)!! = (2k-1)*(2k-3)*\dots*5*3*1$.

EXAMPLES:

```

sage: SetPartitionsBk(3).cardinality()
15
sage: SetPartitionsBk(2).cardinality()
3
sage: SetPartitionsBk(1).cardinality()
1
sage: SetPartitionsBk(4).cardinality()
105
sage: SetPartitionsBk(5).cardinality()
945

```

class sage.combinat.partition_algebra. **SetPartitionsBkhalf_k** (*k*)
 Bases: *sage.combinat.partition_algebra.SetPartitionsAkhalf_k*

cardinality ()

sage.combinat.partition_algebra. SetPartitionsIk (*k*)

Returns the combinatorial class of set partitions of type I_k . These are set partitions with a propagating number of less than k . Note that the identity set partition $\{\{1, -1\}, \dots, \{k, -k\}\}$ is not in I_k .

EXAMPLES:

```

sage: I3 = SetPartitionsIk(3); I3
Set partitions of {1, ..., 3, -1, ..., -3} with propagating number < 3
sage: I3.cardinality()
197

sage: I3.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: I3.last() #random
{{-1}, {-2}, {3}, {1}, {-3}, {2}}
sage: I3.random_element() #random
{{-1}, {-3, -2}, {2, 3}, {1}}

sage: I2p5 = SetPartitionsIk(2.5); I2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and
↳ propagating number < 3
sage: I2p5.cardinality()
50

```

```

sage: I2p5.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: I2p5.last() #random
{{-1}, {-2}, {2}, {3, -3}, {1}}
sage: I2p5.random_element() #random
{{-1}, {-2}, {1, 3, -3}, {2}}

```

class sage.combinat.partition_algebra. **SetPartitionsIk_k** (*k*)
 Bases: *sage.combinat.partition_algebra.SetPartitionsAk_k*

cardinality ()

class sage.combinat.partition_algebra. **SetPartitionsIkhalf_k** (*k*)
 Bases: *sage.combinat.partition_algebra.SetPartitionsAkhalf_k*

cardinality ()

sage.combinat.partition_algebra. **SetPartitionsPRk** (*k*)

class sage.combinat.partition_algebra. **SetPartitionsPRk_k** (*k*)
 Bases: *sage.combinat.partition_algebra.SetPartitionsRk_k*

cardinality ()

class sage.combinat.partition_algebra. **SetPartitionsPRkhalf_k** (*k*)
 Bases: *sage.combinat.partition_algebra.SetPartitionsRkhalf_k*

cardinality ()

sage.combinat.partition_algebra. **SetPartitionsPk** (*k*)
 Returns the combinatorial class of set partitions of type *P_k*. These are the planar set partitions.

EXAMPLES:

```

sage: P3 = SetPartitionsPk(3); P3
Set partitions of {1, ..., 3, -1, ..., -3} that are planar
sage: P3.cardinality()
132

sage: P3.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: P3.last() #random
{{-1}, {-2}, {3}, {1}, {-3}, {2}}
sage: P3.random_element() #random
{{1, 2, -1}, {-3}, {3, -2}}

sage: P2p5 = SetPartitionsPk(2.5); P2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and
↳ that are planar
sage: P2p5.cardinality()
42

sage: P2p5.first() #random
{{1, 2, 3, -1, -3, -2}}
sage: P2p5.last() #random
{{-1}, {-2}, {2}, {3, -3}, {1}}
sage: P2p5.random_element() #random
{{1, 2, 3, -3}, {-1, -2}}

```

```

class sage.combinat.partition_algebra. SetPartitionsPk_k (k)
    Bases: sage.combinat.partition_algebra.SetPartitionsAk_k

    cardinality ( )

class sage.combinat.partition_algebra. SetPartitionsPkhalf_k (k)
    Bases: sage.combinat.partition_algebra.SetPartitionsAkhalf_k

    cardinality ( )

sage.combinat.partition_algebra. SetPartitionsRk (k)

class sage.combinat.partition_algebra. SetPartitionsRk_k (k)
    Bases: sage.combinat.partition_algebra.SetPartitionsAk_k

    cardinality ( )

class sage.combinat.partition_algebra. SetPartitionsRkhalf_k (k)
    Bases: sage.combinat.partition_algebra.SetPartitionsAkhalf_k

    cardinality ( )

sage.combinat.partition_algebra. SetPartitionsSk (k)
    Returns the combinatorial class of set partitions of type  $S_k$ . There is a bijection between these set partitions
    and the permutations of  $1, \dots, k$ .

```

EXAMPLES:

```

sage: S3 = SetPartitionsSk(3); S3
Set partitions of {1, ..., 3, -1, ..., -3} with propagating number 3
sage: S3.cardinality()
6

sage: S3.list() #random
[{{2, -2}, {3, -3}, {1, -1}},
 {{1, -1}, {2, -3}, {3, -2}},
 {{2, -1}, {3, -3}, {1, -2}},
 {{1, -2}, {2, -3}, {3, -1}},
 {{1, -3}, {2, -1}, {3, -2}},
 {{1, -3}, {2, -2}, {3, -1}}]
sage: S3.first() #random
{{2, -2}, {3, -3}, {1, -1}}
sage: S3.last() #random
{{1, -3}, {2, -2}, {3, -1}}
sage: S3.random_element() #random
{{1, -3}, {2, -1}, {3, -2}}

sage: S3p5 = SetPartitionsSk(3.5); S3p5
Set partitions of {1, ..., 4, -1, ..., -4} with 4 and -4 in the same block and
↳ propagating number 4
sage: S3p5.cardinality()
6

sage: S3p5.list() #random
[{{2, -2}, {3, -3}, {1, -1}, {4, -4}},
 {{2, -3}, {1, -1}, {4, -4}, {3, -2}},
 {{2, -1}, {3, -3}, {1, -2}, {4, -4}},
 {{2, -3}, {1, -2}, {4, -4}, {3, -1}},
 {{1, -3}, {2, -1}, {4, -4}, {3, -2}},
 {{1, -3}, {2, -2}, {4, -4}, {3, -1}}]
sage: S3p5.first() #random
{{2, -2}, {3, -3}, {1, -1}, {4, -4}}

```

```

sage: S3p5.last() #random
{{1, -3}, {2, -2}, {4, -4}, {3, -1}}
sage: S3p5.random_element() #random
{{1, -3}, {2, -2}, {4, -4}, {3, -1}}

```

class sage.combinat.partition_algebra. **SetPartitionsSk_k** (*k*)
 Bases: *sage.combinat.partition_algebra.SetPartitionsAk_k*

cardinality ()
 Returns $k!$.

class sage.combinat.partition_algebra. **SetPartitionsSkhalf_k** (*k*)
 Bases: *sage.combinat.partition_algebra.SetPartitionsAkhalf_k*

cardinality ()

```

sage: ks = [2.5, 3.5, 4.5, 5.5]
sage: sks = [SetPartitionsSk(k) for k in ks]
sage: all(sk.cardinality() == len(sk.list()) for sk in sks)
True

```

sage.combinat.partition_algebra. **SetPartitionsTk** (*k*)
 Returns the combinatorial class of set partitions of type T_k . These are planar set partitions where every block is of size 2.

EXAMPLES:

```

sage: T3 = SetPartitionsTk(3); T3
Set partitions of {1, ..., 3, -1, ..., -3} with block size 2 and that are planar
sage: T3.cardinality()
5

sage: T3.first() #random
{{1, -3}, {2, 3}, {-1, -2}}
sage: T3.last() #random
{{1, 2}, {3, -1}, {-3, -2}}
sage: T3.random_element() #random
{{1, -3}, {2, 3}, {-1, -2}}

sage: T2p5 = SetPartitionsTk(2.5); T2p5
Set partitions of {1, ..., 3, -1, ..., -3} with 3 and -3 in the same block and
↳ with block size 2 and that are planar
sage: T2p5.cardinality()
2

sage: T2p5.first() #random
{{2, -2}, {3, -3}, {1, -1}}
sage: T2p5.last() #random
{{1, 2}, {3, -3}, {-1, -2}}

```

class sage.combinat.partition_algebra. **SetPartitionsTk_k** (*k*)
 Bases: *sage.combinat.partition_algebra.SetPartitionsBk_k*

cardinality ()

class sage.combinat.partition_algebra. **SetPartitionsTkhalf_k** (*k*)
 Bases: *sage.combinat.partition_algebra.SetPartitionsBkhalf_k*

cardinality ()

class sage.combinat.partition_algebra. **SetPartitionsXkElement** (*parent, s*)
 Bases: sage.combinat.set_partition.SetPartition

An element for the classes of SetPartitionXk where X is some letter.

check ()

Check to make sure this is a set partition.

EXAMPLES:

```
sage: A2p5 = SetPartitionsAk(2.5)
sage: x = A2p5.first(); x # random
{{1, 2, 3, -1, -3, -2}}
sage: x.check()
sage: y = A2p5.next(x); y
{{-3, -2, -1, 2, 3}, {1}}
sage: y.check()
```

sage.combinat.partition_algebra. **create_set_partition_function** (*letter, k*)

EXAMPLES:

```
sage: from sage.combinat.partition_algebra import create_set_partition_function
sage: create_set_partition_function('A', 3)
Set partitions of {1, ..., 3, -1, ..., -3}
```

sage.combinat.partition_algebra. **identity** (*k*)

Returns the identity set partition 1, -1, ..., k, -k

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: pa.identity(2)
{{2, -2}, {1, -1}}
```

sage.combinat.partition_algebra. **is_planar** (*sp*)

Returns True if the diagram corresponding to the set partition is planar; otherwise, it returns False.

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: pa.is_planar( pa.to_set_partition([[1,-2],[2,-1]]))
False
sage: pa.is_planar( pa.to_set_partition([[1,-1],[2,-2]]))
True
```

sage.combinat.partition_algebra. **pair_to_graph** (*sp1, sp2*)

Return a graph consisting of the disjoint union of the graphs of set partitions *sp1* and *sp2* along with edges joining the bottom row (negative numbers) of *sp1* to the top row (positive numbers) of *sp2*.

The vertices of the graph *sp1* appear in the result as pairs $(k, 1)$, whereas the vertices of the graph *sp2* appear as pairs $(k, 2)$.

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: sp1 = pa.to_set_partition([[1,-2],[2,-1]])
sage: sp2 = pa.to_set_partition([[1,-2],[2,-1]])
sage: g = pa.pair_to_graph( sp1, sp2 ); g
Graph on 8 vertices
```

```

sage: g.vertices() #random
[(1, 2), (-1, 1), (-2, 2), (-1, 2), (-2, 1), (2, 1), (2, 2), (1, 1)]
sage: g.edges() #random
[((1, 2), (-1, 1), None),
 ((1, 2), (-2, 2), None),
 ((-1, 1), (2, 1), None),
 ((-1, 2), (2, 2), None),
 ((-2, 1), (1, 1), None),
 ((-2, 1), (2, 2), None)]

```

Another example which used to be wrong until [trac ticket #15958](#):

```

sage: sp3 = pa.to_set_partition([[1, -1], [2], [-2]])
sage: sp4 = pa.to_set_partition([[1], [-1], [2], [-2]])
sage: g = pa.pair_to_graph( sp3, sp4 ); g
Graph on 8 vertices

sage: g.vertices()
[(-2, 1), (-2, 2), (-1, 1), (-1, 2), (1, 1), (1, 2), (2, 1), (2, 2)]
sage: g.edges()
[((-2, 1), (2, 2), None), ((-1, 1), (1, 1), None),
 ((-1, 1), (1, 2), None)]

```

`sage.combinat.partition_algebra.propagating_number (sp)`

Returns the propagating number of the set partition *sp*. The propagating number is the number of blocks with both a positive and negative number.

EXAMPLES:

```

sage: import sage.combinat.partition_algebra as pa
sage: sp1 = pa.to_set_partition([[1,-2],[2,-1]])
sage: sp2 = pa.to_set_partition([[1,2],[-2,-1]])
sage: pa.propagating_number(sp1)
2
sage: pa.propagating_number(sp2)
0

```

`sage.combinat.partition_algebra.set_partition_composition (sp1, sp2)`

Returns a tuple consisting of the composition of the set partitions *sp1* and *sp2* and the number of components removed from the middle rows of the graph.

EXAMPLES:

```

sage: import sage.combinat.partition_algebra as pa
sage: sp1 = pa.to_set_partition([[1,-2],[2,-1]])
sage: sp2 = pa.to_set_partition([[1,-2],[2,-1]])
sage: pa.set_partition_composition(sp1, sp2) == (pa.identity(2), 0)
True

```

`sage.combinat.partition_algebra.to_graph (sp)`

Returns a graph representing the set partition *sp*.

EXAMPLES:

```

sage: import sage.combinat.partition_algebra as pa
sage: g = pa.to_graph( pa.to_set_partition([[1,-2],[2,-1]]) ); g
Graph on 4 vertices

```

```
sage: g.vertices() #random
[1, 2, -2, -1]
sage: g.edges() #random
[(1, -2, None), (2, -1, None)]
```

`sage.combinat.partition_algebra.to_set_partition (l, k=None)`

Coverts a list of a list of numbers to a set partitions. Each list of numbers in the outer list specifies the numbers contained in one of the blocks in the set partition.

If k is specified, then the set partition will be a set partition of $1, \dots, k, -1, \dots, -k$. Otherwise, k will default to the minimum number needed to contain all of the specified numbers.

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: pa.to_set_partition([[1,-1],[2,-2]]) == pa.identity(2)
True
```

4.14 Quaternion Algebras

AUTHORS:

- Jon Bobber (2009): rewrite
- William Stein (2009): rewrite
- Julian Rueth (2014-03-02): use UniqueFactory for caching

This code is partly based on Sage code by David Kohel from 2005.

class `sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebraFactory`
 Bases: `sage.structure.factory.UniqueFactory`

There are three input formats:

- `QuaternionAlgebra(a,b)` : quaternion algebra generated by i, j subject to $i^2 = a, j^2 = b, j \cdot i = -i \cdot j$.
- `QuaternionAlgebra(K,a,b)` : same as above but over a field K . Here, a and b are nonzero elements of a field (K) of characteristic not 2, and we set $k = i \cdot j$.
- `QuaternionAlgebra(D)` : a rational quaternion algebra with discriminant D , where $D > 1$ is a squarefree integer.

EXAMPLES:

`QuaternionAlgebra(a,b)` - return quaternion algebra over the *smallest* field containing the nonzero elements a and b with generators i, j, k with $i^2 = a, j^2 = b$ and $j \cdot i = -i \cdot j$:

```
sage: QuaternionAlgebra(-2,-3)
Quaternion Algebra (-2, -3) with base ring Rational Field
sage: QuaternionAlgebra(GF(5)(2), GF(5)(3))
Quaternion Algebra (2, 3) with base ring Finite Field of size 5
sage: QuaternionAlgebra(2, GF(5)(3))
Quaternion Algebra (2, 3) with base ring Finite Field of size 5
sage: QuaternionAlgebra(QQ[sqrt(2)](-1), -5)
Quaternion Algebra (-1, -5) with base ring Number Field in sqrt2 with defining_
  ↪ polynomial x^2 - 2
sage: QuaternionAlgebra(sqrt(-1), sqrt(-3))
```



```

Quaternion Algebra (I, sqrt(-3)) with base ring Symbolic Ring
sage: QuaternionAlgebra(1r,1)
Quaternion Algebra (1, 1) with base ring Rational Field

```

Python ints, longs and floats may be passed to the `QuaternionAlgebra(a,b)` constructor, as may all pairs of nonzero elements of a ring not of characteristic 2. The following tests address the issues raised in [trac ticket #10601](#):

```

sage: QuaternionAlgebra(1r,1)
Quaternion Algebra (1, 1) with base ring Rational Field
sage: QuaternionAlgebra(1,1.0r)
Quaternion Algebra (1.0000000000000000, 1.0000000000000000) with base ring Real Field_
→with 53 bits of precision
sage: QuaternionAlgebra(0,0)
Traceback (most recent call last):
...
ValueError: a and b must be nonzero
sage: QuaternionAlgebra(GF(2)(1),1)
Traceback (most recent call last):
...
ValueError: a and b must be elements of a ring with characteristic not 2
sage: a = PermutationGroupElement([1,2,3])
sage: QuaternionAlgebra(a, a)
Traceback (most recent call last):
...
ValueError: a and b must be elements of a ring with characteristic not 2

```

`QuaternionAlgebra(K, a, b)` - return quaternion algebra over the field K with generators i, j, k with $i^2 = a, j^2 = b$ and $i \cdot j = -j \cdot i$:

```

sage: QuaternionAlgebra(QQ, -7, -21)
Quaternion Algebra (-7, -21) with base ring Rational Field
sage: QuaternionAlgebra(QQ[sqrt(2)], -2, -3)
Quaternion Algebra (-2, -3) with base ring Number Field in sqrt2 with defining_
→polynomial x^2 - 2

```

`QuaternionAlgebra(D)` - D is a squarefree integer; returns a rational quaternion algebra of discriminant D :

```

sage: QuaternionAlgebra(1)
Quaternion Algebra (-1, 1) with base ring Rational Field
sage: QuaternionAlgebra(2)
Quaternion Algebra (-1, -1) with base ring Rational Field
sage: QuaternionAlgebra(7)
Quaternion Algebra (-1, -7) with base ring Rational Field
sage: QuaternionAlgebra(2*3*5*7)
Quaternion Algebra (-22, 210) with base ring Rational Field

```

If the coefficients a and b in the definition of the quaternion algebra are not integral, then a slower generic type is used for arithmetic:

```

sage: type(QuaternionAlgebra(-1,-3).0)
<type 'sage.algebras.quatalg.quaternion_algebra_element.QuaternionAlgebraElement_
→rational_field'>
sage: type(QuaternionAlgebra(-1,-3/2).0)
<type 'sage.algebras.quatalg.quaternion_algebra_element.QuaternionAlgebraElement_
→generic'>

```

Make sure caching is sane:

```
sage: A = QuaternionAlgebra(2,3); A
Quaternion Algebra (2, 3) with base ring Rational Field
sage: B = QuaternionAlgebra(GF(5)(2),GF(5)(3)); B
Quaternion Algebra (2, 3) with base ring Finite Field of size 5
sage: A is QuaternionAlgebra(2,3)
True
sage: B is QuaternionAlgebra(GF(5)(2),GF(5)(3))
True
sage: Q = QuaternionAlgebra(2); Q
Quaternion Algebra (-1, -1) with base ring Rational Field
sage: Q is QuaternionAlgebra(QQ,-1,-1)
True
sage: Q is QuaternionAlgebra(-1,-1)
True
sage: Q.<ii,jj,kk> = QuaternionAlgebra(15); Q.variable_names()
('ii', 'jj', 'kk')
sage: QuaternionAlgebra(15).variable_names()
('i', 'j', 'k')
```

create_key (*arg0*, *arg1=None*, *arg2=None*, *names='i,j,k'*)

Create a key that uniquely determines a quaternion algebra.

create_object (*version*, *key*, ***extra_args*)

Create the object from the key (extra arguments are ignored). This is only called if the object was not found in the cache.

```
class sage.algebras.quatalg.quaternion_algebra. QuaternionAlgebra_ab ( base_ring,
                                                                    a,      b,
                                                                    names='i,
                                                                    j,k')
```

Bases: *sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract*

The quaternion algebra of the form $(a,b/K)$, where $i^2 = a$, $j^2 = b$, and $j * i = -i * j$. K is a field not of characteristic 2 and a, b are nonzero elements of K .

See `QuaternionAlgebra` for many more examples.

INPUT:

- *base_ring* – commutative ring
- *a, b* – elements of *base_ring*
- *names* – string (optional, default 'i,j,k') names of the generators

EXAMPLES:

```
sage: QuaternionAlgebra(QQ, -7, -21) # indirect doctest
Quaternion Algebra (-7, -21) with base ring Rational Field
```

discriminant ()

Given a quaternion algebra A defined over a number field, return the discriminant of A , i.e. the product of the ramified primes of A .

EXAMPLES:

```
sage: QuaternionAlgebra(210,-22).discriminant()
210
sage: QuaternionAlgebra(19).discriminant()
```

```

19

sage: F.<a> = NumberField(x^2-x-1)
sage: B.<i,j,k> = QuaternionAlgebra(F, 2*a,F(-1))
sage: B.discriminant()
Fractional ideal (2)

sage: QuaternionAlgebra(QQ[sqrt(2)],3,19).discriminant()
Fractional ideal (1)

```

gen (*i=0*)

Return the i^{th} generator of self.

INPUT:

- *i* - integer (optional, default 0)

EXAMPLES:

```

sage: Q.<ii,jj,kk> = QuaternionAlgebra(QQ,-1,-2); Q
Quaternion Algebra (-1, -2) with base ring Rational Field
sage: Q.gen(0)
ii
sage: Q.gen(1)
jj
sage: Q.gen(2)
kk
sage: Q.gens()
[ii, jj, kk]

```

ideal (*gens*, *left_order=None*, *right_order=None*, *check=True*, ***kws*)

Return the quaternion ideal with given gens over \mathbb{Z} . Neither a left or right order structure need be specified.

INPUT:

- *gens* – a list of elements of this quaternion order
- *check* – bool (default: True); if False, then *gens* must 4-tuple that forms a Hermite basis for an ideal
- *left_order* – a quaternion order or None
- *right_order* – a quaternion order or None

EXAMPLES:

```

sage: R = QuaternionAlgebra(-11,-1)
sage: R.ideal([2*a for a in R.basis()])
Fractional ideal (2, 2*i, 2*j, 2*k)

```

inner_product_matrix ()

Return the inner product matrix associated to self, i.e. the Gram matrix of the reduced norm as a quadratic form on self. The standard basis $1, i, j, k$ is orthogonal, so this matrix is just the diagonal matrix with diagonal entries $1, a, b, ab$.

EXAMPLES:

```

sage: Q.<i,j,k> = QuaternionAlgebra(-5,-19)
sage: Q.inner_product_matrix()
[ 2  0  0  0]
[ 0 10  0  0]

```

```

[ 0  0 38  0]
[ 0  0  0 190]

sage: R.<a,b> = QQ[]; Q.<i,j,k> = QuaternionAlgebra(Frac(R),a,b)
sage: Q.inner_product_matrix()
[ 2  0  0  0]
[ 0 -2*a 0  0]
[ 0  0 -2*b 0]
[ 0  0  0 2*a*b]

```

invariants ()

Return the structural invariants a, b of this quaternion algebra: `self` is generated by i, j subject to $i^2 = a$, $j^2 = b$ and $j * i = -i * j$.

EXAMPLES:

```

sage: Q.<i,j,k> = QuaternionAlgebra(15)
sage: Q.invariants()
(-3, 5)
sage: i^2
-3
sage: j^2
5

```

maximal_order (take_shortcuts=True)

Return a maximal order in this quaternion algebra.

The algorithm used is from [Voi2012].

INPUT:

- `take_shortcuts` – (default: `True`) if the discriminant is prime and the invariants of the algebra are of a nice form, use Proposition 5.2 of [Piz1980].

OUTPUT:

A maximal order in this quaternion algebra.

EXAMPLES:

```

sage: QuaternionAlgebra(-1,-7).maximal_order()
Order of Quaternion Algebra (-1, -7) with base ring Rational Field with basis_
↪ (1/2 + 1/2*j, 1/2*i + 1/2*k, j, k)

sage: QuaternionAlgebra(-1,-1).maximal_order().basis()
(1/2 + 1/2*i + 1/2*j + 1/2*k, i, j, k)

sage: QuaternionAlgebra(-1,-11).maximal_order().basis()
(1/2 + 1/2*j, 1/2*i + 1/2*k, j, k)

sage: QuaternionAlgebra(-1,-3).maximal_order().basis()
(1/2 + 1/2*j, 1/2*i + 1/2*k, j, k)

sage: QuaternionAlgebra(-3,-1).maximal_order().basis()
(1/2 + 1/2*i, 1/2*j - 1/2*k, i, -k)

sage: QuaternionAlgebra(-2,-5).maximal_order().basis()
(1/2 + 1/2*j + 1/2*k, 1/4*i + 1/2*j + 1/4*k, j, k)

sage: QuaternionAlgebra(-5,-2).maximal_order().basis()

```

```
(1/2 + 1/2*i - 1/2*k, 1/2*i + 1/4*j - 1/4*k, i, -k)

sage: QuaternionAlgebra(-17,-3).maximal_order().basis()
(1/2 + 1/2*j, 1/2*i + 1/2*k, -1/3*j - 1/3*k, k)

sage: QuaternionAlgebra(-3,-17).maximal_order().basis()
(1/2 + 1/2*i, 1/2*j - 1/2*k, -1/3*i + 1/3*k, -k)

sage: QuaternionAlgebra(-17*9,-3).maximal_order().basis()
(1, 1/3*i, 1/6*i + 1/2*j, 1/2 + 1/3*j + 1/18*k)

sage: QuaternionAlgebra(-2, -389).maximal_order().basis()
(1/2 + 1/2*j + 1/2*k, 1/4*i + 1/2*j + 1/4*k, j, k)
```

If you want bases containing 1, switch off `take_shortcuts` :

```
sage: QuaternionAlgebra(-3,-89).maximal_order(take_shortcuts=False)
Order of Quaternion Algebra (-3, -89) with base ring Rational Field with
↳basis (1, 1/2 + 1/2*i, j, 1/2 + 1/6*i + 1/2*j + 1/6*k)

sage: QuaternionAlgebra(1,1).maximal_order(take_shortcuts=False)      # Matrix_
↳ring
Order of Quaternion Algebra (1, 1) with base ring Rational Field with basis_
↳(1, 1/2 + 1/2*i, j, 1/2*j + 1/2*k)

sage: QuaternionAlgebra(-22,210).maximal_order(take_shortcuts=False)
Order of Quaternion Algebra (-22, 210) with base ring Rational Field with
↳basis (1, i, 1/2*i + 1/2*j, 1/2 + 17/22*i + 1/44*k)

sage: for d in ( m for m in range(1, 750) if is_squarefree(m) ):      #
↳long time (3s)
....:     A = QuaternionAlgebra(d)
....:     R = A.maximal_order(take_shortcuts=False)
....:     assert A.discriminant() == R.discriminant()
```

We don't support number fields other than the rationals yet:

```
sage: K = QuadraticField(5)
sage: QuaternionAlgebra(K,-1,-1).maximal_order()
Traceback (most recent call last):
...
NotImplementedError: maximal order only implemented for rational quaternion_
↳algebras
```

`modp_splitting_data (p)`

Return mod p splitting data for this quaternion algebra at the unramified prime p . This is 2×2 matrices I, J, K over the finite field \mathbb{F}_p such that if the quaternion algebra has generators i, j, k , then $I^2 = i^2$, $J^2 = j^2$, $IJ = K$ and $IJ = -JI$.

Note: Currently only implemented when p is odd and the base ring is \mathbb{Q} .

INPUT:

- p – unramified odd prime

OUTPUT:

•2-tuple of matrices over finite field

EXAMPLES:

```
sage: Q = QuaternionAlgebra(-15, -19)
sage: Q.modp_splitting_data(7)
(
 [0 6]  [6 1]  [6 6]
 [1 0], [1 1], [6 1]
)
sage: Q.modp_splitting_data(next_prime(10^5))
(
 [ 0 99988]  [97311 4]  [99999 59623]
 [ 1 0], [13334 2692], [97311 4]
)
sage: I, J, K = Q.modp_splitting_data(23)
sage: I
[0 8]
[1 0]
sage: I^2
[8 0]
[0 8]
sage: J
[19 2]
[17 4]
sage: J^2
[4 0]
[0 4]
sage: I*J == -J*I
True
sage: I*J == K
True
```

The following is a good test because of the asserts in the code:

```
sage: v = [Q.modp_splitting_data(p) for p in primes(20, 1000)]
```

Proper error handling:

```
sage: Q.modp_splitting_data(5)
Traceback (most recent call last):
...
NotImplementedError: algorithm for computing local splittings not implemented,
↳ in general (currently require the first invariant to be coprime to p)

sage: Q.modp_splitting_data(2)
Traceback (most recent call last):
...
NotImplementedError: p must be odd
```

modp_splitting_map (*p*)

Return Python map from the (*p*-integral) quaternion algebra to the set of 2×2 matrices over \mathbb{F}_p .

INPUT:

•*p* – prime number

EXAMPLES:

```

sage: Q.<i,j,k> = QuaternionAlgebra(-1, -7)
sage: f = Q.modp_splitting_map(13)
sage: a = 2+i-j+3*k; b = 7+2*i-4*j+k
sage: f(a*b)
[12  3]
[10  5]
sage: f(a)*f(b)
[12  3]
[10  5]

```

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

Return the order of this quaternion order with given basis.

INPUT:

- *basis* - list of 4 elements of self
- *check* - bool (default: True)

EXAMPLES:

```

sage: Q.<i,j,k> = QuaternionAlgebra(-11,-1)
sage: Q.quaternion_order([1,i,j,k])
Order of Quaternion Algebra (-11, -1) with base ring Rational Field with
↳basis (1, i, j, k)

```

We test out *check=False* :

```

sage: Q.quaternion_order([1,i,j,k], check=False)
Order of Quaternion Algebra (-11, -1) with base ring Rational Field with
↳basis [1, i, j, k]
sage: Q.quaternion_order([i,j,k], check=False)
Order of Quaternion Algebra (-11, -1) with base ring Rational Field with
↳basis [i, j, k]

```

ramified_primes ()

Return the primes that ramify in this quaternion algebra. Currently only implemented over the rational numbers.

EXAMPLES:

```

sage: QuaternionAlgebra(QQ, -1, -1).ramified_primes()
[2]

```

class sage.algebras.quatalg.quaternion_algebra. **QuaternionAlgebra_abstract**

Bases: sage.rings.ring.Algebra

basis ()

Return the fixed basis of *self*, which is $1, i, j, k$, where i, j, k are the generators of *self*.

EXAMPLES:

```

sage: Q.<i,j,k> = QuaternionAlgebra(QQ,-5,-2)
sage: Q.basis()
(1, i, j, k)

sage: Q.<xyz,abc,theta> = QuaternionAlgebra(GF(9,'a'),-5,-2)
sage: Q.basis()
(1, xyz, abc, theta)

```

The basis is cached:

```
sage: Q.basis() is Q.basis()
True
```

`inner_product_matrix ()`

Return the inner product matrix associated to `self`, i.e. the Gram matrix of the reduced norm as a quadratic form on `self`. The standard basis $1, i, j, k$ is orthogonal, so this matrix is just the diagonal matrix with diagonal entries $2, 2a, 2b, 2ab$.

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(-5,-19)
sage: Q.inner_product_matrix()
[ 2  0  0  0]
[ 0 10  0  0]
[ 0  0 38  0]
[ 0  0  0 190]
```

`is_commutative ()`

Return `False` always, since all quaternion algebras are noncommutative.

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -3,-7)
sage: Q.is_commutative()
False
```

`is_division_algebra ()`

Return `True` if the quaternion algebra is a division algebra (i.e. every nonzero element in `self` is invertible), and `False` if the quaternion algebra is isomorphic to the 2×2 matrix algebra.

EXAMPLES:

```
sage: QuaternionAlgebra(QQ,-5,-2).is_division_algebra()
True
sage: QuaternionAlgebra(1).is_division_algebra()
False
sage: QuaternionAlgebra(2,9).is_division_algebra()
False
sage: QuaternionAlgebra(RR(2.),1).is_division_algebra()
Traceback (most recent call last):
...
NotImplementedError: base field must be rational numbers
```

`is_exact ()`

Return `True` if elements of this quaternion algebra are represented exactly, i.e. there is no precision loss when doing arithmetic. A quaternion algebra is exact if and only if its base field is exact.

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -3, -7)
sage: Q.is_exact()
True
sage: Q.<i,j,k> = QuaternionAlgebra(Qp(7), -3, -7)
sage: Q.is_exact()
False
```


is_field (*proof=True*)

Return False always, since all quaternion algebras are noncommutative and all fields are commutative.

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -3, -7)
sage: Q.is_field()
False
```

is_finite ()

Return True if the quaternion algebra is finite as a set.

Algorithm: A quaternion algebra is finite if and only if the base field is finite.

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -3, -7)
sage: Q.is_finite()
False
sage: Q.<i,j,k> = QuaternionAlgebra(GF(5), -3, -7)
sage: Q.is_finite()
True
```

is_integral_domain (*proof=True*)

Return False always, since all quaternion algebras are noncommutative and integral domains are commutative (in Sage).

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -3, -7)
sage: Q.is_integral_domain()
False
```

is_matrix_ring ()

Return True if the quaternion algebra is isomorphic to the 2x2 matrix ring, and False if self is a division algebra (i.e. every nonzero element in self is invertible).

EXAMPLES:

```
sage: QuaternionAlgebra(QQ, -5, -2).is_matrix_ring()
False
sage: QuaternionAlgebra(1).is_matrix_ring()
True
sage: QuaternionAlgebra(2, 9).is_matrix_ring()
True
sage: QuaternionAlgebra(RR(2.), 1).is_matrix_ring()
Traceback (most recent call last):
...
NotImplementedError: base field must be rational numbers
```

is_noetherian ()

Return True always, since any quaternion algebra is a noetherian ring (because it is a finitely generated module over a field).

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -3, -7)
sage: Q.is_noetherian()
True
```

ngens ()

Return the number of generators of the quaternion algebra as a K -vector space, not including 1. This value is always 3: the algebra is spanned by the standard basis $1, i, j, k$.

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -5, -2)
sage: Q.ngens()
3
sage: Q.gens()
[i, j, k]
```

order ()

Return the number of elements of the quaternion algebra, or `+Infinity` if the algebra is not finite.

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -3, -7)
sage: Q.order()
+Infinity
sage: Q.<i,j,k> = QuaternionAlgebra(GF(5), -3, -7)
sage: Q.order()
625
```

random_element (*args, **kwds)

Return a random element of this quaternion algebra.

The `args` and `kwds` are passed to the `random_element` method of the base ring.

EXAMPLES:

```
sage: QuaternionAlgebra(QQ[sqrt(2)], -3, 7).random_element()
(sqrt(2) + 2)*i + (-12*sqrt(2) - 2)*j + (-sqrt(2) + 1)*k
sage: QuaternionAlgebra(-3, 19).random_element()
-1 + 2*i - j - 6/5*k
sage: QuaternionAlgebra(GF(17)(2), 3).random_element()
14 + 10*i + 4*j + 7*k
```

Specify the numerator and denominator bounds:

```
sage: QuaternionAlgebra(-3, 19).random_element(10^6, 10^6)
-979933/553629 + 255525/657688*i - 3511/6929*j - 700105/258683*k
```

vector_space ()

Return the vector space associated to `self` with inner product given by the reduced norm.

EXAMPLES:

```
sage: QuaternionAlgebra(-3, 19).vector_space()
Ambient quadratic space of dimension 4 over Rational Field
Inner product matrix:
[  2   0   0   0]
[  0   6   0   0]
[  0   0 -38   0]
[  0   0   0 -114]
```

```
class sage.algebras.quatalg.quaternion_algebra. QuaternionFractionalIdeal ( ring,
                                                                           gens,
                                                                           co-
                                                                           erce=True)
```

Bases: `sage.rings.ideal.Ideal_fractional`

Initialize this ideal.

INPUT:

- `ring` – A ring
- `gens` – The generators for this ideal
- `coerce` – (default: `True`) If `gens` needs to be coerced into `ring`.

EXAMPLES:

```
sage: R.<x> = ZZ[]
sage: R.ideal([4 + 3*x + x^2, 1 + x^2])
Ideal (x^2 + 3*x + 4, x^2 + 1) of Univariate Polynomial Ring in x over Integer_
↪Ring
```

class `sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational` (*basis*, *left_order=None*, *right_order=None*, *check=True*)

Bases: `sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal`

A fractional ideal in a rational quaternion algebra.

INPUT:

- `left_order` – a quaternion order or `None`
- `right_order` – a quaternion order or `None`
- `basis` – tuple of length 4 of elements in of ambient quaternion algebra whose \mathbf{Z} -span is an ideal
- `check` – bool (default: `True`); if `False`, do no type checking, and the input basis *must* be in Hermite form.

basis ()

Return basis for this fractional ideal. The basis is in Hermite form.

OUTPUT: tuple

EXAMPLES:

```
sage: QuaternionAlgebra(-11,-1).maximal_order().unit_ideal().basis()
(1/2 + 1/2*i, 1/2*j - 1/2*k, i, -k)
```

basis_matrix ()

Return basis matrix M in Hermite normal form for self as a matrix with rational entries.

If Q is the ambient quaternion algebra, then the \mathbf{Z} -span of the rows of M viewed as linear combinations of $Q.basis() = [1, i, j, k]$ is the fractional ideal self. Also, $M * M.denominator()$ is an integer matrix in Hermite normal form.

OUTPUT: matrix over Q

EXAMPLES:

```
sage: QuaternionAlgebra(-11,-1).maximal_order().unit_ideal().basis_matrix()
[ 1/2  1/2   0   0]
[  0    0  1/2 -1/2]
[  0    1   0   0]
[  0    0   0  -1]
```

conjugate ()

Return the ideal with generators the conjugates of the generators for self.

OUTPUT: a quaternionic fractional ideal

EXAMPLES:

```
sage: I = BrandtModule(3,5).right_ideals()[1]; I
Fractional ideal (2 + 6*j + 4*k, 2*i + 4*j + 34*k, 8*j + 32*k, 40*k)
sage: I.conjugate()
Fractional ideal (2 + 2*j + 28*k, 2*i + 4*j + 34*k, 8*j + 32*k, 40*k)
```

cyclic_right_subideals (p, alpha=None)

Let $I = \text{self}$. This function returns the right subideals J of I such that I/J is an \mathbf{F}_p -vector space of dimension 2.

INPUT:

- p – prime number (see below)
- α – (default: None) element of quaternion algebra, which can be used to parameterize the order of the ideals J . More precisely the J 's are the right annihilators of $(1,0)\alpha^i$ for $i = 0, 1, 2, \dots, p$

OUTPUT:

- list of right ideals

Note: Currently, p must satisfy a bunch of conditions, or a `NotImplementedError` is raised. In particular, p must be odd and unramified in the quaternion algebra, must be coprime to the index of the right order in the maximal order, and also coprime to the normal of self. (The Brandt modules code has a more general algorithm in some cases.)

EXAMPLES:

```
sage: B = BrandtModule(2,37); I = B.right_ideals()[0]
sage: I.cyclic_right_subideals(3)
[Fractional ideal (2 + 2*i + 10*j + 90*k, 4*i + 4*j + 152*k, 12*j + 132*k, 444*k),
Fractional ideal (2 + 2*i + 2*j + 150*k, 4*i + 8*j + 196*k, 12*j + 132*k, 444*k),
Fractional ideal (2 + 2*i + 6*j + 194*k, 4*i + 8*j + 344*k, 12*j + 132*k, 444*k),
Fractional ideal (2 + 2*i + 6*j + 46*k, 4*i + 4*j + 4*k, 12*j + 132*k, 444*k)]

sage: B = BrandtModule(5,389); I = B.right_ideals()[0]
sage: C = I.cyclic_right_subideals(3); C
[Fractional ideal (2 + 10*j + 546*k, i + 6*j + 133*k, 12*j + 3456*k, 4668*k),
Fractional ideal (2 + 2*j + 2910*k, i + 6*j + 3245*k, 12*j + 3456*k, 4668*k),
Fractional ideal (2 + i + 2295*k, 3*i + 2*j + 3571*k, 4*j + 2708*k, 4668*k),
Fractional ideal (2 + 2*i + 2*j + 4388*k, 3*i + 2*j + 2015*k, 4*j + 4264*k, 4668*k)]
sage: [(I.free_module()/J.free_module()).invariants() for J in C]
[(3, 3), (3, 3), (3, 3), (3, 3)]
sage: I.scale(3).cyclic_right_subideals(3)
[Fractional ideal (6 + 30*j + 1638*k, 3*i + 18*j + 399*k, 36*j + 10368*k, 14004*k),
Fractional ideal (6 + 6*j + 8730*k, 3*i + 18*j + 9735*k, 36*j + 10368*k, 14004*k),
Fractional ideal (6 + 3*i + 6885*k, 9*i + 6*j + 10713*k, 12*j + 8124*k, 14004*k),
Fractional ideal (6 + 6*i + 6*j + 13164*k, 9*i + 6*j + 6045*k, 12*j + 12792*k, 14004*k)]
sage: C = I.scale(1/9).cyclic_right_subideals(3); C
[Fractional ideal (2/9 + 10/9*j + 182/3*k, 1/9*i + 2/3*j + 133/9*k, 4/3*j + 384*k, 1556/3*k),
Fractional ideal (2/9 + 2/9*j + 970/3*k, 1/9*i + 2/3*j + 3245/9*k, 4/3*j + 384*k, 1556/3*k),
Fractional ideal (2/9 + 1/9*i + 255*k, 1/3*i + 2/9*j + 3571/9*k, 4/9*j + 2708/9*k, 1556/3*k),
Fractional ideal (2/9 + 2/9*i + 2/9*j + 4388/9*k, 1/3*i + 2/9*j + 2015/9*k, 4/9*j + 4264/9*k, 1556/3*k)]
```

```

sage: [(I.scale(1/9).free_module()/J.free_module()).invariants() for J in C]
[(3, 3), (3, 3), (3, 3), (3, 3)]

sage: Q.<i,j,k> = QuaternionAlgebra(-2,-5)
sage: I = Q.ideal([Q(1),i,j,k])
sage: I.cyclic_right_subideals(3)
[Fractional ideal (1 + 2*j, i + k, 3*j, 3*k), Fractional ideal (1 + j, i +
↪ 2*k, 3*j, 3*k), Fractional ideal (1 + 2*i, 3*i, j + 2*k, 3*k), Fractional
↪ ideal (1 + i, 3*i, j + k, 3*k)]

```

The general algorithm is not yet implemented here:

```

sage: I.cyclic_right_subideals(3)[0].cyclic_right_subideals(3)
Traceback (most recent call last):
...
NotImplementedError: general algorithm not implemented (The given basis
↪ vectors must be linearly independent.)

```

free_module ()

Return the underlying free \mathbf{Z} -module corresponding to this ideal.

EXAMPLES:

```

sage: X = BrandtModule(3,5).right_ideals()
sage: X[0]
Fractional ideal (2 + 2*j + 8*k, 2*i + 18*k, 4*j + 16*k, 20*k)
sage: X[0].free_module()
Free module of degree 4 and rank 4 over Integer Ring
Echelon basis matrix:
[ 2  0  2  8]
[ 0  2  0 18]
[ 0  0  4 16]
[ 0  0  0 20]
sage: X[0].scale(1/7).free_module()
Free module of degree 4 and rank 4 over Integer Ring
Echelon basis matrix:
[ 2/7  0  2/7  8/7]
[  0  2/7  0 18/7]
[  0  0  4/7 16/7]
[  0  0  0 20/7]

```

The free module method is also useful since it allows for checking if one ideal is contained in another, computing quotients I/J , etc.:

```

sage: X = BrandtModule(3,17).right_ideals()
sage: I = X[0].intersection(X[2]); I
Fractional ideal (2 + 2*j + 164*k, 2*i + 4*j + 46*k, 16*j + 224*k, 272*k)
sage: I.free_module().is_submodule(X[3].free_module())
False
sage: I.free_module().is_submodule(X[1].free_module())
True
sage: X[0].free_module() / I.free_module()
Finitely generated module V/W over Integer Ring with invariants (4, 4)

```

gens ()

Return the generators for this ideal, which are the same as the \mathbf{Z} -basis for this ideal.

EXAMPLES:

```
sage: QuaternionAlgebra(-11,-1).maximal_order().unit_ideal().gens()
(1/2 + 1/2*i, 1/2*j - 1/2*k, i, -k)
```

gram_matrix ()

Return the Gram matrix of this fractional ideal.

OUTPUT: 4×4 matrix over \mathbb{Q} .

EXAMPLES:

```
sage: I = BrandtModule(3,5).right_ideals()[1]; I
Fractional ideal (2 + 6*j + 4*k, 2*i + 4*j + 34*k, 8*j + 32*k, 40*k)
sage: I.gram_matrix()
[ 640  1920  2112  1920]
[ 1920 14080 13440 16320]
[ 2112 13440 13056 15360]
[ 1920 16320 15360 19200]
```

intersection (J)

Return the intersection of the ideals self and J .

EXAMPLES:

```
sage: X = BrandtModule(3,5).right_ideals()
sage: I = X[0].intersection(X[1]); I
Fractional ideal (2 + 6*j + 4*k, 2*i + 4*j + 34*k, 8*j + 32*k, 40*k)
```

is_equivalent (I, J, B=10)

Return True if I and J are equivalent as right ideals.

INPUT:

- I – a fractional quaternion ideal (self)
- J – a fractional quaternion ideal with same order as I
- B – a bound to compute and compare theta series before doing the full equivalence test

OUTPUT: bool

EXAMPLES:

```
sage: R = BrandtModule(3,5).right_ideals(); len(R)
2
sage: R[0].is_equivalent(R[1])
False
sage: R[0].is_equivalent(R[0])
True
sage: OO = R[0].quaternion_order()
sage: S = OO.right_ideal([3*a for a in R[0].basis()])
sage: R[0].is_equivalent(S)
True
```

left_order ()

Return the left order associated to this fractional ideal.

OUTPUT: an order in a quaternion algebra

EXAMPLES:

```

sage: B = BrandtModule(11)
sage: R = B.maximal_order()
sage: I = R.unit_ideal()
sage: I.left_order()
Order of Quaternion Algebra (-1, -11) with base ring Rational Field with
↳basis (1/2 + 1/2*j, 1/2*i + 1/2*k, j, k)

```

We do a consistency check:

```

sage: B = BrandtModule(11,19); R = B.right_ideals()
sage: [r.left_order().discriminant() for r in R]
[209, 209, 209, 209, 209, 209, 209, 209, 209, 209, 209, 209, 209, 209, 209,
↳209, 209, 209]

```

multiply_by_conjugate (J)

Return product of self and the conjugate Jbar of J.

INPUT:

- J – a quaternion ideal.

OUTPUT: a quaternionic fractional ideal.

EXAMPLES:

```

sage: R = BrandtModule(3,5).right_ideals()
sage: R[0].multiply_by_conjugate(R[1])
Fractional ideal (8 + 8*j + 112*k, 8*i + 16*j + 136*k, 32*j + 128*k, 160*k)
sage: R[0]*R[1].conjugate()
Fractional ideal (8 + 8*j + 112*k, 8*i + 16*j + 136*k, 32*j + 128*k, 160*k)

```

norm ()

Return the reduced norm of this fractional ideal.

OUTPUT: rational number

EXAMPLES:

```

sage: M = BrandtModule(37)
sage: C = M.right_ideals()
sage: [I.norm() for I in C]
[16, 32, 32]

sage: (a,b) = M.quaternion_algebra().invariants()
↳ # optional - magma
sage: magma.eval('A<i,j,k> := QuaternionAlgebra<Rationals() | %s, %s>' %
↳(a,b)) # optional - magma
''
sage: magma.eval('O := QuaternionOrder(%s)' % str(list(C[0].right_order().
↳basis())) # optional - magma
''
sage: [ magma('ideal<O | %s>' % str(list(I.basis()))).Norm() for I in C]
↳ # optional - magma
[16, 32, 32]

sage: A.<i,j,k> = QuaternionAlgebra(-1,-1)
sage: R = A.ideal([i,j,k,1/2 + 1/2*i + 1/2*j + 1/2*k]) # this is
↳actually an order, so has reduced norm 1
sage: R.norm()
1

```

```
sage: [ J.norm() for J in R.cyclic_right_subideals(3) ]      # enumerate_
↪maximal right R-ideals of reduced norm 3, verify their norms
[3, 3, 3, 3]
```

quadratic_form ()

Return the normalized quadratic form associated to this quaternion ideal.

OUTPUT: quadratic form

EXAMPLES:

```
sage: I = BrandtModule(11).right_ideals()[1]
sage: Q = I.quadratic_form(); Q
Quadratic form in 4 variables over Rational Field with coefficients:
[ 18 22 33 22 ]
[ * 7 22 11 ]
[ * * 22 0 ]
[ * * * 22 ]
sage: Q.theta_series(10)
1 + 12*q^2 + 12*q^3 + 12*q^4 + 12*q^5 + 24*q^6 + 24*q^7 + 36*q^8 + 36*q^9 +
↪O(q^10)
sage: I.theta_series(10)
1 + 12*q^2 + 12*q^3 + 12*q^4 + 12*q^5 + 24*q^6 + 24*q^7 + 36*q^8 + 36*q^9 +
↪O(q^10)
```

quaternion_algebra ()

Return the ambient quaternion algebra that contains this fractional ideal.

OUTPUT: a quaternion algebra

EXAMPLES:

```
sage: I = BrandtModule(3,5).right_ideals()[1]; I
Fractional ideal (2 + 6*j + 4*k, 2*i + 4*j + 34*k, 8*j + 32*k, 40*k)
sage: I.quaternion_algebra()
Quaternion Algebra (-1, -3) with base ring Rational Field
```

quaternion_order ()

Return the order for which this ideal is a left or right fractional ideal. If this ideal has both a left and right ideal structure, then the left order is returned. If it has neither structure, then an error is raised.

OUTPUT: QuaternionOrder

EXAMPLES:

```
sage: R = QuaternionAlgebra(-11,-1).maximal_order()
sage: R.unit_ideal().quaternion_order() is R
True
```

right_order ()

Return the right order associated to this fractional ideal.

OUTPUT: an order in a quaternion algebra

EXAMPLES:

```
sage: I = BrandtModule(389).right_ideals()[1]; I
Fractional ideal (2 + 6*j + 2*k, i + 2*j + k, 8*j, 8*k)
sage: I.right_order()
Order of Quaternion Algebra (-2, -389) with base ring Rational Field with
↪basis (1/2 + 1/2*j + 1/2*k, 1/4*i + 1/2*j + 1/4*k, j, k)
```



```
sage: I.left_order()
Order of Quaternion Algebra (-2, -389) with base ring Rational Field with
↪basis (1/2 + 1/2*j + 3/2*k, 1/8*i + 1/4*j + 9/8*k, j + k, 2*k)
```

The following is a big consistency check. We take reps for all the right ideal classes of a certain order, take the corresponding left orders, then take ideals in the left orders and from those compute the right order again:

```
sage: B = BrandtModule(11,19); R = B.right_ideals()
sage: O = [r.left_order() for r in R]
sage: J = [O[i].left_ideal(R[i].basis()) for i in range(len(R))]
sage: len(set(J))
18
sage: len(set([I.right_order() for I in J]))
1
sage: J[0].right_order() == B.order_of_level_N()
True
```

ring ()

Return ring that this is a fractional ideal for.

EXAMPLES:

```
sage: R = QuaternionAlgebra(-11,-1).maximal_order()
sage: R.unit_ideal().ring() is R
True
```

scale (alpha, left=False)

Scale the fractional ideal self by multiplying the basis by alpha .

INPUT:

- α – element of quaternion algebra
- left – bool (default: False); if true multiply α on the left, otherwise multiply α on the right

OUTPUT:

- a new fractional ideal

EXAMPLES:

```
sage: B = BrandtModule(5,37); I = B.right_ideals()[0]; i,j,k = B.quaternion_
↪algebra().gens(); I
Fractional ideal (2 + 2*j + 106*k, i + 2*j + 105*k, 4*j + 64*k, 148*k)
sage: I.scale(i)
Fractional ideal [2*i + 212*j - 2*k, -2 + 210*j - 2*k, 128*j - 4*k, 296*j]
sage: I.scale(i, left=True)
Fractional ideal [2*i - 212*j + 2*k, -2 - 210*j + 2*k, -128*j + 4*k, -296*j]
sage: I.scale(i, left=False)
Fractional ideal [2*i + 212*j - 2*k, -2 + 210*j - 2*k, 128*j - 4*k, 296*j]
sage: i * I.gens()[0]
2*i - 212*j + 2*k
sage: I.gens()[0] * i
2*i + 212*j - 2*k
```

theta_series (B, var='q')

Return normalized theta series of self, as a power series over \mathbb{Z} in the variable var , which is 'q' by default.

The normalized theta series is by definition

$$\theta_I(q) = \sum_{x \in I} q^{\frac{N(x)}{N(I)}}.$$

INPUT:

- **B** – positive integer
- **var** – string (default: 'q')

OUTPUT: power series

EXAMPLES:

```
sage: I = BrandtModule(11).right_ideals()[1]; I
Fractional ideal (2 + 6*j + 4*k, 2*i + 4*j + 2*k, 8*j, 8*k)
sage: I.norm()
32
sage: I.theta_series(5)
1 + 12*q^2 + 12*q^3 + 12*q^4 + O(q^5)
sage: I.theta_series(5, 'T')
1 + 12*T^2 + 12*T^3 + 12*T^4 + O(T^5)
sage: I.theta_series(3)
1 + 12*q^2 + O(q^3)
```

theta_series_vector (*B*)

Return theta series coefficients of *self*, as a vector of *B* integers.

INPUT:

- **B** – positive integer

OUTPUT:

Vector over \mathbf{Z} with *B* entries.

EXAMPLES:

```
sage: I = BrandtModule(37).right_ideals()[1]; I
Fractional ideal (2 + 6*j + 2*k, i + 2*j + k, 8*j, 8*k)
sage: I.theta_series_vector(5)
(1, 0, 2, 2, 6)
sage: I.theta_series_vector(10)
(1, 0, 2, 2, 6, 4, 8, 6, 10, 10)
sage: I.theta_series_vector(5)
(1, 0, 2, 2, 6)
```

class sage.algebras.quatalg.quaternion_algebra. **QuaternionOrder** (*A*, *basis*, *check=True*)

Bases: sage.rings.ring.Algebra

An order in a quaternion algebra.

EXAMPLES:

```
sage: QuaternionAlgebra(-1,-7).maximal_order()
Order of Quaternion Algebra (-1, -7) with base ring Rational Field with basis (1/
→ 2 + 1/2*j, 1/2*i + 1/2*k, j, k)
sage: type(QuaternionAlgebra(-1,-7).maximal_order())
<class 'sage.algebras.quatalg.quaternion_algebra.QuaternionOrder'>
```

basis ()

Return fix choice of basis for this quaternion order.

EXAMPLES:

```
sage: QuaternionAlgebra(-11,-1).maximal_order().basis()
(1/2 + 1/2*i, 1/2*j - 1/2*k, i, -k)
```

discriminant ()

Return the discriminant of this order, which we define as $\sqrt{\det(\text{Tr}(e_i \bar{e}_j))}$, where $\{e_i\}$ is the basis of the order.

OUTPUT: rational number

EXAMPLES:

```
sage: QuaternionAlgebra(-11,-1).maximal_order().discriminant()
11
sage: S = BrandtModule(11,5).order_of_level_N()
sage: S.discriminant()
55
sage: type(S.discriminant())
<type 'sage.rings.rational.Rational'>
```

free_module ()

Return the free \mathbf{Z} -module that corresponds to this order inside the vector space corresponding to the ambient quaternion algebra.

OUTPUT:

A free \mathbf{Z} -module of rank 4.

EXAMPLES:

```
sage: R = QuaternionAlgebra(-11,-1).maximal_order()
sage: R.basis()
(1/2 + 1/2*i, 1/2*j - 1/2*k, i, -k)
sage: R.free_module()
Free module of degree 4 and rank 4 over Integer Ring
Echelon basis matrix:
[1/2 1/2  0  0]
[  0  1  0  0]
[  0  0 1/2 1/2]
[  0  0  0  1]
```

gen (n)

Return the n-th generator.

INPUT:

- n - an integer between 0 and 3, inclusive.

EXAMPLES:

```
sage: R = QuaternionAlgebra(-11,-1).maximal_order(); R
Order of Quaternion Algebra (-11, -1) with base ring Rational Field with
↳basis (1/2 + 1/2*i, 1/2*j - 1/2*k, i, -k)
sage: R.gen(0)
1/2 + 1/2*i
sage: R.gen(1)
1/2*j - 1/2*k
```

```
sage: R.gen(2)
i
sage: R.gen(3)
-k
```

gens ()

Return generators for self.

EXAMPLES:

```
sage: QuaternionAlgebra(-1,-7).maximal_order().gens()
(1/2 + 1/2*j, 1/2*i + 1/2*k, j, k)
```

intersection (*other*)

Return the intersection of this order with other.

INPUT:

- *other* - a quaternion order in the same ambient quaternion algebra

OUTPUT: a quaternion order

EXAMPLES:

```
sage: R = QuaternionAlgebra(-11,-1).maximal_order()
sage: R.intersection(R)
Order of Quaternion Algebra (-11, -1) with base ring Rational Field with
↳basis (1/2 + 1/2*i, i, 1/2*j + 1/2*k, k)
```

We intersect various orders in the quaternion algebra ramified at 11:

```
sage: B = BrandtModule(11,3)
sage: R = B.maximal_order(); S = B.order_of_level_N()
sage: R.intersection(S)
Order of Quaternion Algebra (-1, -11) with base ring Rational Field with
↳basis (1/2 + 1/2*j, 1/2*i + 5/2*k, j, 3*k)
sage: R.intersection(S) == S
True
sage: B = BrandtModule(11,5)
sage: T = B.order_of_level_N()
sage: S.intersection(T)
Order of Quaternion Algebra (-1, -11) with base ring Rational Field with
↳basis (1/2 + 1/2*j, 1/2*i + 23/2*k, j, 15*k)
```

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

Return the ideal with given gens over \mathbb{Z} .

INPUT:

- *gens* - a list of elements of this quaternion order
- *check* - bool (default: True); if False, then *gens* must 4-tuple that forms a Hermite basis for an ideal

EXAMPLES:

```
sage: R = QuaternionAlgebra(-11,-1).maximal_order()
sage: R.left_ideal([2*a for a in R.basis()])
Fractional ideal (1 + i, 2*i, j + k, 2*k)
```

ngens ()

Return the number of generators (which is 4).

EXAMPLES:

```
sage: QuaternionAlgebra(-1,-7).maximal_order().ngens()
4
```

quadratic_form ()

Return the normalized quadratic form associated to this quaternion order.

OUTPUT: quadratic form

EXAMPLES:

```
sage: R = BrandtModule(11,13).order_of_level_N()
sage: Q = R.quadratic_form(); Q
Quadratic form in 4 variables over Rational Field with coefficients:
[ 14 253 55 286 ]
[ * 1455 506 3289 ]
[ * * 55 572 ]
[ * * * 1859 ]
sage: Q.theta_series(10)
1 + 2*q + 2*q^4 + 4*q^6 + 4*q^8 + 2*q^9 + O(q^10)
```

quaternion_algebra ()

Return ambient quaternion algebra that contains this quaternion order.

EXAMPLES:

```
sage: QuaternionAlgebra(-11,-1).maximal_order().quaternion_algebra()
Quaternion Algebra (-11, -1) with base ring Rational Field
```

random_element (*args, **kws)

Return a random element of this order.

The args and kws are passed to the random_element method of the integer ring, and we return an element of the form

$$ae_1 + be_2 + ce_3 + de_4$$

where e_1, \dots, e_4 are the basis of this order and a, b, c, d are random integers.

EXAMPLES:

```
sage: QuaternionAlgebra(-11,-1).maximal_order().random_element()
-4 - 4*i + j - k
sage: QuaternionAlgebra(-11,-1).maximal_order().random_element(-10,10)
-9/2 - 7/2*i - 7/2*j - 3/2*k
```

right_ideal (gens, check=True)Return the ideal with given gens over \mathbb{Z} .

INPUT:

- gens – a list of elements of this quaternion order
- check – bool (default: True); if False, then gens must 4-tuple that forms a Hermite basis for an ideal

EXAMPLES:

```
sage: R = QuaternionAlgebra(-11,-1).maximal_order()
sage: R.right_ideal([2*a for a in R.basis()])
Fractional ideal (1 + i, 2*i, j + k, 2*k)
```

ternary_quadratic_form (*include_basis=False*)

Return the ternary quadratic form associated to this order.

INPUT:

- *include_basis* – bool (default: False), if True also return a basis for the dimension 3 subspace G

OUTPUT:

- QuadraticForm
- optional basis for dimension 3 subspace

This function computes the positive definition quadratic form obtained by letting G be the trace zero subspace of $\mathbf{Z} + 2*\text{self}$, which has rank 3, and restricting the pairing:

```
(x,y) = (x.conjugate()*y).reduced_trace()
```

to G .

APPLICATIONS: Ternary quadratic forms associated to an order in a rational quaternion algebra are useful in computing with Gross points, in deciding whether quaternion orders have embeddings from orders in quadratic imaginary fields, and in computing elements of the Kohnen plus subspace of modular forms of weight $3/2$.

EXAMPLES:

```
sage: R = BrandtModule(11,13).order_of_level_N()
sage: Q = R.ternary_quadratic_form(); Q
Quadratic form in 3 variables over Rational Field with coefficients:
[ 5820 1012 13156 ]
[ * 55 1144 ]
[ * * 7436 ]
sage: factor(Q.disc())
2^4 * 11^2 * 13^2
```

The following theta series is a modular form of weight $3/2$ and level $4*11*13$:

```
sage: Q.theta_series(100)
1 + 2*q^23 + 2*q^55 + 2*q^56 + 2*q^75 + 4*q^92 + O(q^100)
```

unit_ideal ()

Return the unit ideal in this quaternion order.

EXAMPLES:

```
sage: R = QuaternionAlgebra(-11,-1).maximal_order()
sage: I = R.unit_ideal(); I
Fractional ideal (1/2 + 1/2*i, 1/2*j - 1/2*k, i, -k)
```

`sage.algebras.quatalg.quaternion_algebra.basis_for_quaternion_lattice` (*gens,*
re-
verse=False)

Return a basis for the \mathbf{Z} -lattice in a quaternion algebra spanned by the given gens.

INPUT:

- gens – list of elements of a single quaternion algebra
- reverse – when computing the HNF do it on the basis $(k, j, i, 1)$ instead of $(1, i, j, k)$; this ensures that if gens are the generators for an order, the first returned basis vector is 1

EXAMPLES:

```
sage: from sage.algebras.quatalg.quaternion_algebra import basis_for_quaternion_
      ↪lattice
sage: A.<i,j,k> = QuaternionAlgebra(-1,-7)
sage: basis_for_quaternion_lattice([i+j, i-j, 2*k, A(1/3)])
[1/3, i + j, 2*j, 2*k]

sage: basis_for_quaternion_lattice([A(1), i, j, k])
[1, i, j, k]
```

sage.algebras.quatalg.quaternion_algebra. **intersection_of_row_modules_over_ZZ** (v)
Intersects the \mathbf{Z} -modules with basis matrices the full rank 4×4 \mathbf{Q} -matrices in the list v. The returned intersection is represented by a 4×4 matrix over \mathbf{Q} . This can also be done using modules and intersection, but that would take over twice as long because of overhead, hence this function.

EXAMPLES:

```
sage: a = matrix(QQ,4,[-2, 0, 0, 0, 0, -1, -1, 1, 2, -1/2, 0, 0, 1, 1, -1, 0])
sage: b = matrix(QQ,4,[0, -1/2, 0, -1/2, 2, 1/2, -1, -1/2, 1, 2, 1, -2, 0, -1/2, -
      ↪2, 0])
sage: c = matrix(QQ,4,[0, 1, 0, -1/2, 0, 0, 2, 2, 0, -1/2, 1/2, -1, 1, -1, -1/2,
      ↪0])
sage: v = [a,b,c]
sage: from sage.algebras.quatalg.quaternion_algebra import intersection_of_row_
      ↪modules_over_ZZ
sage: M = intersection_of_row_modules_over_ZZ(v); M
[  2   0  -1  -1]
[ -4   1   1  -3]
[  3 -19/2  1   4]
[  2  -3  -8   4]
sage: M2 = a.row_module(ZZ).intersection(b.row_module(ZZ)).intersection(c.row_
      ↪module(ZZ))
sage: M.row_module(ZZ) == M2
True
```

sage.algebras.quatalg.quaternion_algebra. **is_QuaternionAlgebra** (A)
Return True if A is of the QuaternionAlgebra data type.

EXAMPLES:

```
sage: sage.algebras.quatalg.quaternion_algebra.is_
      ↪QuaternionAlgebra(QuaternionAlgebra(QQ,-1,-1))
True
sage: sage.algebras.quatalg.quaternion_algebra.is_QuaternionAlgebra(ZZ)
False
```

sage.algebras.quatalg.quaternion_algebra. **maxord_solve_aux_eq** (a, b, p)
Given a and b and an even prime ideal p find (y,z,w) with y a unit mod p^{2e} such that

$$1 - ay^2 - bz^2 + abw^2 \equiv 0 \pmod{p^{2e}},$$

where e is the ramification index of p.

Currently only $p = 2$ is implemented by hardcoding solutions.

INPUT:

- a – integer with $v_p(a) = 0$
- b – integer with $v_p(b) \in \{0, 1\}$
- p – even prime ideal (actually only $p=2$ is implemented)

OUTPUT:

- A tuple (y, z, w)

EXAMPLES:

```
sage: from sage.algebras.quatalg.quaternion_algebra import maxord_solve_aux_eq
sage: for a in [1,3]:
.....:     for b in [1,2,3]:
.....:         (y,z,w) = maxord_solve_aux_eq(a, b, 2)
.....:         assert mod(y, 4) == 1 or mod(y, 4) == 3
.....:         assert mod(1 - a*y^2 - b*z^2 + a*b*w^2, 4) == 0
```

`sage.algebras.quatalg.quaternion_algebra.normalize_basis_at_p(e, p, B=<function <lambda>>)`

Computes a (at p) normalized basis from the given basis e of a \mathbf{Z} -module.

The returned basis is (at p) a \mathbf{Z}_p basis for the same module, and has the property that with respect to it the quadratic form induced by the bilinear form B is represented as a orthogonal sum of atomic forms multiplied by p -powers.

If $p \neq 2$ this means that the form is diagonal with respect to this basis.

If $p = 2$ there may be additional 2-dimensional subspaces on which the form is represented as $2^e(ax^2 + bxy + cx^2)$ with $0 = v_2(b) = v_2(a) \leq v_2(c)$.

INPUT:

- e – list; basis of a \mathbf{Z} module. WARNING: will be modified!
- p – prime for at which the basis should be normalized
- B – (default: `lambda x,y: ((x*y).conjugate()).reduced_trace()`) a bilinear form with respect to which to normalize

OUTPUT:

- A list containing two-element tuples: The first element of each tuple is a basis element, the second the valuation of the orthogonal summand to which it belongs. The list is sorted by ascending valuation.

EXAMPLES:

```
sage: from sage.algebras.quatalg.quaternion_algebra import normalize_basis_at_p
sage: A.<i,j,k> = QuaternionAlgebra(-1, -1)
sage: e = [A(1), i, j, k]
sage: normalize_basis_at_p(e, 2)
[(1, 0), (i, 0), (j, 0), (k, 0)]

sage: A.<i,j,k> = QuaternionAlgebra(210)
sage: e = [A(1), i, j, k]
sage: normalize_basis_at_p(e, 2)
[(1, 0), (i, 1), (j, 1), (k, 2)]

sage: A.<i,j,k> = QuaternionAlgebra(286)
sage: e = [A(1), k, 1/2*j + 1/2*k, 1/2 + 1/2*i + 1/2*k]
```



```

sage: normalize_basis_at_p(e, 5)
[(1, 0), (1/2*j + 1/2*k, 0), (-5/6*j + 1/6*k, 1), (1/2*i, 1)]

sage: A.<i,j,k> = QuaternionAlgebra(-1,-7)
sage: e = [A(1), k, j, 1/2 + 1/2*i + 1/2*j + 1/2*k]
sage: normalize_basis_at_p(e, 2)
[(1, 0), (1/2 + 1/2*i + 1/2*j + 1/2*k, 0), (-34/105*i - 463/735*j + 71/105*k, 1),
→ (-34/105*i - 463/735*j + 71/105*k, 1)]

```

sage.algebras.quatalg.quaternion_algebra. **unpickle_QuaternionAlgebra_v0** (*key)
The 0th version of pickling for quaternion algebras.

EXAMPLES:

```

sage: Q = QuaternionAlgebra(-5,-19)
sage: t = (QQ, -5, -19, ('i', 'j', 'k'))
sage: sage.algebras.quatalg.quaternion_algebra.unpickle_QuaternionAlgebra_v0(*t)
Quaternion Algebra (-5, -19) with base ring Rational Field
sage: loads(dumps(Q)) == Q
True
sage: loads(dumps(Q)) is Q
True

```

4.15 Rational Cherednik Algebras

class sage.algebras.rational_cherednik_algebra. **RationalCherednikAlgebra** (*ct*,
c, *t*,
base_ring,
pre-
fix)

Bases: sage.combinat.free_module.CombinatorialFreeModule

A rational Cherednik algebra.

Let k be a field. Let W be a complex reflection group acting on a vector space \mathfrak{h} (over k). Let \mathfrak{h}^* denote the corresponding dual vector space. Let \cdot denote the natural action of w on \mathfrak{h} and \mathfrak{h}^* . Let S denote the set of reflections of W and α_s and α_s^\vee are the associated root and coroot of s . Let $c = (c_s)_{s \in W}$ such that $c_s = c_{tst^{-1}}$ for all $t \in W$.

The *rational Cherednik algebra* is the k -algebra $H_{c,t}(W) = T(\mathfrak{h} \oplus \mathfrak{h}^*) \otimes kW$ with parameters $c, t \in k$ that is subject to the relations:

$$\begin{aligned}
 w\alpha &= (w \cdot \alpha)w, \\
 \alpha^\vee w &= w(w^{-1} \cdot \alpha^\vee), \\
 \alpha\alpha^\vee &= \alpha^\vee\alpha + t\langle \alpha^\vee, \alpha \rangle + \sum_{s \in S} c_s \frac{\langle \alpha^\vee, \alpha_s \rangle \langle \alpha_s^\vee, \alpha \rangle}{\langle \alpha^\vee, \alpha \rangle} s,
 \end{aligned}$$

where $w \in W$ and $\alpha \in \mathfrak{h}$ and $\alpha^\vee \in \mathfrak{h}^*$.

INPUT:

- *ct* – a finite Cartan type
- *c* – the parameters c_s given as an element or a tuple, where the first entry is the one for the long roots and (for non-simply-laced types) the second is for the short roots
- *t* – the parameter t

- `base_ring` – (optional) the base ring
- `prefix` – (default: `('a', 's', 'ac')`) the prefixes

Todo

Implement a version for complex reflection groups.

REFERENCES:

- [GGOR2003]
- [EM2001]

algebra_generators ()

Return the algebra generators of `self`.

EXAMPLES:

```
sage: R = algebras.RationalCherednik(['A',2], 1, 1, QQ)
sage: list(R.algebra_generators())
[a1, a2, s1, s2, ac1, ac2]
```

an_element ()

Return an element of `self`.

EXAMPLES:

```
sage: R = algebras.RationalCherednik(['A',2], 1, 1, QQ)
sage: R.an_element()
3*ac1 + 2*s1 + a1
```

deformed_euler ()

Return the element eu_k .

EXAMPLES:

```
sage: R = algebras.RationalCherednik(['A',2], 1, 1, QQ)
sage: R.deformed_euler()
2*I + 2/3*a1*ac1 + 1/3*a1*ac2 + 1/3*a2*ac1 + 2/3*a2*ac2
+ s1 + s2 + s1*s2*s1
```

degree_on_basis (m)

Return the degree on the monomial indexed by `m`.

EXAMPLES:

```
sage: R = algebras.RationalCherednik(['A',2], 1, 1, QQ)
sage: [R.degree_on_basis(g.leading_support())
....:  for g in R.algebra_generators()]
[1, 1, 0, 0, -1, -1]
```

one_basis ()

Return the index of the element 1.

EXAMPLES:

```
sage: R = algebras.RationalCherednik(['A',2], 1, 1, QQ)
sage: R.one_basis()
(1, 1, 1)
```

product_on_basis (*left, right*)

Return *left* multiplied by *right* in self.

EXAMPLES:

```
sage: R = algebras.RationalCherednik(['A',2], 1, 1, QQ)
sage: a2 = R.algebra_generators()['a2']
sage: ac1 = R.algebra_generators()['ac1']
sage: a2 * ac1 # indirect doctest
a2*ac1
sage: ac1 * a2
-I + a2*ac1 - s1 - s2 + 1/2*s1*s2*s1
sage: x = R.an_element()
sage: [y * x for y in R.some_elements()]
[0,
 3*ac1 + 2*s1 + a1,
 9*ac1^2 + 10*I + 6*a1*ac1 + 6*s1 + 3/2*s2 + 3/2*s1*s2*s1 + a1^2,
 3*a1*ac1 + 2*a1*s1 + a1^2,
 3*a2*ac1 + 2*a2*s1 + a1*a2,
 3*s1*ac1 + 2*I - a1*s1,
 3*s2*ac1 + 2*s2*s1 + a1*s2 + a2*s2,
 3*ac1^2 - 2*s1*ac1 + 2*I + a1*ac1 + 2*s1 + 1/2*s2 + 1/2*s1*s2*s1,
 3*ac1*ac2 + 2*s1*ac1 + 2*s1*ac2 - I + a1*ac2 - s1 - s2 + 1/2*s1*s2*s1]
sage: [x * y for y in R.some_elements()]
[0,
 3*ac1 + 2*s1 + a1,
 9*ac1^2 + 10*I + 6*a1*ac1 + 6*s1 + 3/2*s2 + 3/2*s1*s2*s1 + a1^2,
 6*I + 3*a1*ac1 + 6*s1 + 3/2*s2 + 3/2*s1*s2*s1 - 2*a1*s1 + a1^2,
 -3*I + 3*a2*ac1 - 3*s1 - 3*s2 + 3/2*s1*s2*s1 + 2*a1*s1 + 2*a2*s1 + a1*a2,
 -3*s1*ac1 + 2*I + a1*s1,
 3*s2*ac1 + 3*s2*ac2 + 2*s1*s2 + a1*s2,
 3*ac1^2 + 2*s1*ac1 + a1*ac1,
 3*ac1*ac2 + 2*s1*ac2 + a1*ac2]
```

some_elements ()

Return some elements of self.

EXAMPLES:

```
sage: R = algebras.RationalCherednik(['A',2], 1, 1, QQ)
sage: R.some_elements()
[0, I, 3*ac1 + 2*s1 + a1, a1, a2, s1, s2, ac1, ac2]
```

trivial_idempotent ()

Return the trivial idempotent of self.

Let $e = |W|^{-1} \sum_{w \in W} w$ is the trivial idempotent. Thus $e^2 = e$ and $eW = We$. The trivial idempotent is used in the construction of the spherical Cherednik algebra from the rational Cherednik algebra by $U_{c,t}(W) = eH_{c,t}(W)e$.

EXAMPLES:

```
sage: R = algebras.RationalCherednik(['A',2], 1, 1, QQ)
sage: R.trivial_idempotent()
1/6*I + 1/6*s1 + 1/6*s2 + 1/6*s2*s1 + 1/6*s1*s2 + 1/6*s1*s2*s1
```

4.16 Schur algebras for GL_n

This file implements:

- Schur algebras for GL_n over an arbitrary field.
- The canonical action of the Schur algebra on a tensor power of the standard representation.
- Using the above to calculate the characters of irreducible GL_n modules.

AUTHORS:

- Eric Webster (2010-07-01): implement Schur algebra
- Hugh Thomas (2011-05-08): implement action of Schur algebra and characters of irreducible modules

`sage.algebras.schur_algebra.GL_irreducible_character (n, mu, KK)`

Return the character of the irreducible module indexed by μ of $GL(n)$ over the field KK .

INPUT:

- n – a positive integer
- μ – a partition of at most n parts
- KK – a field

OUTPUT:

a symmetric function which should be interpreted in n variables to be meaningful as a character

EXAMPLES:

Over \mathbb{Q} , the irreducible character for μ is the Schur function associated to μ , plus garbage terms (Schur functions associated to partitions with more than n parts):

```
sage: from sage.algebras.schur_algebra import GL_irreducible_character
sage: sbasis = SymmetricFunctions(QQ).s()
sage: z = GL_irreducible_character(2, [2], QQ)
sage: sbasis(z)
s[2]

sage: z = GL_irreducible_character(4, [3, 2], QQ)
sage: sbasis(z)
-5*s[1, 1, 1, 1, 1] + s[3, 2]
```

Over a Galois field, the irreducible character for μ will in general be smaller.

In characteristic p , for a one-part partition (r) , where $r = a_0 + pa_1 + p^2a_2 + \dots$, the result is (see [Gr2007], after 5.5d) the product of $h[a_0], h[a_1](pbasis[p]), h[a_2](pbasis[p^2]), \dots$, which is consistent with the following

```
sage: from sage.algebras.schur_algebra import GL_irreducible_character
sage: GL_irreducible_character(2, [7], GF(3))
m[4, 3] + m[6, 1] + m[7]
```

class `sage.algebras.schur_algebra.SchurAlgebra (R, n, r)`

Bases: `sage.combinat.free_module.CombinatorialFreeModule`

A Schur algebra.

Let R be a commutative ring, n be a positive integer, and r be a non-negative integer. Define $A_R(n, r)$ to be the set of homogeneous polynomials of degree r in n^2 variables x_{ij} . Therefore we can write $R[x_{ij}] = \bigoplus_{r \geq 0} A_R(n, r)$, and $R[x_{ij}]$ is known to be a bialgebra with coproduct given by $\Delta(x_{ij}) = \sum_l x_{il} \otimes x_{lj}$ and counit $\varepsilon(x_{ij}) = \delta_{ij}$. Therefore $A_R(n, r)$ is a subcoalgebra of $R[x_{ij}]$. The *Schur algebra* $S_R(n, r)$ is the linear

dual to $A_R(n, r)$, that is $S_R(n, r) := \text{hom}(A_R(n, r), R)$, and $S_R(n, r)$ obtains its algebra structure naturally by dualizing the comultiplication of $A_R(n, r)$.

Let $V = R^n$. One of the most important properties of the Schur algebra $S_R(n, r)$ is that it is isomorphic to the endomorphisms of $V^{\otimes r}$ which commute with the natural action of S_r .

EXAMPLES:

```
sage: S = SchurAlgebra(ZZ, 2, 2); S
Schur algebra (2, 2) over Integer Ring
```

REFERENCES:

- [Gr2007]
- [Wikipedia article Schur_algebra](#)

dimension ()

Return the dimension of `self`.

The dimension of the Schur algebra $S_R(n, r)$ is

$$\dim S_R(n, r) = \binom{n^2 + r - 1}{r}.$$

EXAMPLES:

```
sage: S = SchurAlgebra(QQ, 4, 2)
sage: S.dimension()
136
sage: S = SchurAlgebra(QQ, 2, 4)
sage: S.dimension()
35
```

one ()

Return the element 1 of `self`.

EXAMPLES:

```
sage: S = SchurAlgebra(ZZ, 2, 2)
sage: e = S.one(); e
S((1, 1), (1, 1)) + S((1, 2), (1, 2)) + S((2, 2), (2, 2))

sage: x = S.an_element()
sage: x * e == x
True
sage: all(e * x == x for x in S.basis())
True

sage: S = SchurAlgebra(ZZ, 4, 4)
sage: e = S.one()
sage: x = S.an_element()
sage: x * e == x
True
```

product_on_basis (e_{ij}, e_{kl})

Return the product of basis elements.

EXAMPLES:

```
sage: S = SchurAlgebra(QQ, 2, 3)
sage: B = S.basis()
```

If we multiply two basis elements x and y , such that $x[1]$ and $y[0]$ are not permutations of each other, the result is zero:

```
sage: S.product_on_basis((1, 1, 1), (1, 1, 2)), ((1, 2, 2), (1, 1, 2)))
0
```

If we multiply a basis element x by a basis element which consists of the same tuple repeated twice (on either side), the result is either zero (if the previous case applies) or x :

```
sage: ww = B[((1, 2, 2), (1, 2, 2))]
sage: x = B[((1, 2, 2), (1, 1, 2))]
sage: ww * x
S((1, 2, 2), (1, 1, 2))
```

An arbitrary product, on the other hand, may have multiplicities:

```
sage: x = B[((1, 1, 1), (1, 1, 2))]
sage: y = B[((1, 1, 2), (1, 2, 2))]
sage: x * y
2*S((1, 1, 1), (1, 2, 2))
```

```
class sage.algebras.schur_algebra. SchurTensorModule (  $R, n, r$  )
Bases: sage.combinat.free_module.CombinatorialFreeModule_Tensor
```

The space $V^{\otimes r}$ where $V = R^n$ equipped with a left action of the Schur algebra $S_R(n, r)$ and a right action of the symmetric group S_r .

Let R be a commutative ring and $V = R^n$. We consider the module $V^{\otimes r}$ equipped with a natural right action of the symmetric group S_r given by

$$(v_1 \otimes v_2 \otimes \cdots \otimes v_n)\sigma = v_{\sigma(1)} \otimes v_{\sigma(2)} \otimes \cdots \otimes v_{\sigma(n)}.$$

The Schur algebra $S_R(n, r)$ is naturally isomorphic to the endomorphisms of $V^{\otimes r}$ which commutes with the S_r action. We get the natural left action of $S_R(n, r)$ by this isomorphism.

EXAMPLES:

```
sage: T = SchurTensorModule(QQ, 2, 3); T
The 3-fold tensor product of a free module of dimension 2
over Rational Field
sage: A = SchurAlgebra(QQ, 2, 3)
sage: P = Permutations(3)
sage: t = T.an_element(); t
2*B[1] # B[1] # B[1] + 2*B[1] # B[1] # B[2] + 3*B[1] # B[2] # B[1]
sage: a = A.an_element(); a
2*S((1, 1, 1), (1, 1, 1)) + 2*S((1, 1, 1), (1, 1, 2))
+ 3*S((1, 1, 1), (1, 2, 2))
sage: p = P.an_element(); p
[3, 1, 2]
sage: y = a * t; y
14*B[1] # B[1] # B[1]
sage: y * p
14*B[1] # B[1] # B[1]
sage: z = t * p; z
2*B[1] # B[1] # B[1] + 3*B[1] # B[1] # B[2] + 2*B[2] # B[1] # B[1]
```

```
sage: a * z
14*B[1] # B[1] # B[1]
```

We check the commuting action property:

```
sage: all( (bA * bT) * p == bA * (bT * p)
.....:      for bT in T.basis() for bA in A.basis() for p in P)
True
```

class Element (M, x)

Bases: `sage.combinat.free_module.CombinatorialFreeModuleElement`

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

`sage.algebras.schur_algebra.schur_representative_from_index` ($i0, i1$)

Simultaneously reorder a pair of tuples to obtain the equivalent element of the distinguished basis of the Schur algebra.

See also:

`schur_representative_indices()`

INPUT:

- A pair of tuples of length r with elements in $\{1, \dots, n\}$

OUTPUT:

- The corresponding pair of tuples ordered correctly.

EXAMPLES:

```
sage: from sage.algebras.schur_algebra import schur_representative_from_index
sage: schur_representative_from_index([2, 1, 2, 2], [1, 3, 0, 0])
((1, 2, 2, 2), (3, 0, 0, 1))
```

`sage.algebras.schur_algebra.schur_representative_indices` (n, r)

Return a set which functions as a basis of $S_K(n, r)$.

More specifically, the basis for $S_K(n, r)$ consists of equivalence classes of pairs of tuples of length r on the alphabet $\{1, \dots, n\}$, where the equivalence relation is simultaneous permutation of the two tuples. We can therefore fix a representative for each equivalence class in which the entries of the first tuple weakly increase, and the entries of the second tuple whose corresponding values in the first tuple are equal, also weakly increase.

EXAMPLES:

```
sage: from sage.algebras.schur_algebra import schur_representative_indices
sage: schur_representative_indices(2, 2)
[((1, 1), (1, 1)), ((1, 1), (1, 2)),
 ((1, 1), (2, 2)), ((1, 2), (1, 1)),
 ((1, 2), (1, 2)), ((1, 2), (2, 1)),
 ((1, 2), (2, 2)), ((2, 2), (1, 1)),
 ((2, 2), (1, 2)), ((2, 2), (2, 2))]
```

4.17 The Steenrod algebra

AUTHORS:

- John H. Palmieri (2008-07-30): version 0.9: Initial implementation.
- John H. Palmieri (2010-06-30): version 1.0: Implemented sub-Hopf algebras and profile functions; direct multi-plication of admissible sequences (rather than conversion to the Milnor basis); implemented the Steenrod algebra using CombinatorialFreeModule; improved the test suite.

This module defines the mod p Steenrod algebra \mathcal{A}_p , some of its properties, and ways to define elements of it.

From a topological point of view, \mathcal{A}_p is the algebra of stable cohomology operations on mod p cohomology; thus for any topological space X , its mod p cohomology algebra $H^*(X, \mathbf{F}_p)$ is a module over \mathcal{A}_p .

From an algebraic point of view, \mathcal{A}_p is an \mathbf{F}_p -algebra; when $p = 2$, it is generated by elements Sq^i for $i \geq 0$ (the *Steenrod squares*), and when p is odd, it is generated by elements \mathcal{P}^i for $i \geq 0$ (the *Steenrod reduced p th powers*) along with an element β (the *mod p Bockstein*). The Steenrod algebra is graded: Sq^i is in degree i for each i , β is in degree 1, and \mathcal{P}^i is in degree $2(p-1)i$.

The unit element is Sq^0 when $p = 2$ and \mathcal{P}^0 when p is odd. The generating elements also satisfy the *Adem relations*. At the prime 2, these have the form

$$\text{Sq}^a \text{Sq}^b = \sum_{c=0}^{\lfloor a/2 \rfloor} \binom{b-c-1}{a-2c} \text{Sq}^{a+b-c} \text{Sq}^c.$$

At odd primes, they are a bit more complicated; see Steenrod and Epstein [SE1962] or [sage.algebras.steenrod.steenrod_algebra_bases](#) for full details. These relations lead to the existence of the *Serre-Cartan* basis for \mathcal{A}_p .

The mod p Steenrod algebra has the structure of a Hopf algebra, and Milnor [Mil1958] has a beautiful description of the dual, leading to a construction of the *Milnor basis* for \mathcal{A}_p . In this module, elements in the Steenrod algebra are represented, by default, using the Milnor basis.

Bases for the Steenrod algebra

There are a handful of other bases studied in the literature; the paper by Monks [Mon1998] is a good reference. Here is a quick summary:

- The *Milnor basis*. When $p = 2$, the Milnor basis consists of symbols of the form $\text{Sq}(m_1, m_2, \dots, m_t)$, where each m_i is a non-negative integer and if $t > 1$, then the last entry $m_t > 0$. When p is odd, the Milnor basis consists of symbols of the form $Q_{e_1} Q_{e_2} \dots \mathcal{P}(m_1, m_2, \dots, m_t)$, where $0 \leq e_1 < e_2 < \dots$, each m_i is a non-negative integer, and if $t > 1$, then the last entry $m_t > 0$.

When $p = 2$, it can be convenient to use the notation $\mathcal{P}(-)$ to mean $\text{Sq}(-)$, so that there is consistent notation for all primes.

- The *Serre-Cartan basis*. This basis consists of ‘admissible monomials’ in the Steenrod operations. Thus at the prime 2, it consists of monomials $\text{Sq}^{m_1} \text{Sq}^{m_2} \dots \text{Sq}^{m_t}$ with $m_i \geq 2m_{i+1}$ for each i . At odd primes, this basis consists of monomials $\beta^{\epsilon_0} \mathcal{P}^{s_1} \beta^{\epsilon_1} \mathcal{P}^{s_2} \dots \mathcal{P}^{s_k} \beta^{\epsilon_k}$ with each ϵ_i either 0 or 1, $s_i \geq ps_{i+1} + \epsilon_i$, and $s_k \geq 1$.

Most of the rest of the bases are only defined when $p = 2$. The only exceptions are the P_t^s -bases and the commutator bases, which are defined at all primes.

- *Wood’s Y basis*. For pairs of non-negative integers (m, k) , let $w(m, k) = \text{Sq}^{2^m(2^{k+1}-1)}$. Wood’s Y basis consists of monomials $w(m_0, k_0) \dots w(m_t, k_t)$ with $(m_i, k_i) > (m_{i+1}, k_{i+1})$, in left lex order.
- *Wood’s Z basis*. For pairs of non-negative integers (m, k) , let $w(m, k) = \text{Sq}^{2^m(2^{k+1}-1)}$. Wood’s Z basis consists of monomials $w(m_0, k_0) \dots w(m_t, k_t)$ with $(m_i + k_i, m_i) > (m_{i+1} + k_{i+1}, m_{i+1})$, in left lex order.
- *Wall’s basis*. For any pair of integers (m, k) with $m \geq k \geq 0$, let $Q_k^m = \text{Sq}^{2^k} \text{Sq}^{2^{k+1}} \dots \text{Sq}^{2^m}$. The elements of Wall’s basis are monomials $Q_{k_0}^{m_0} \dots Q_{k_t}^{m_t}$ with $(m_i, k_i) > (m_{i+1}, k_{i+1})$, ordered left lexicographically.
(Note that Q_k^m is the reverse of the element X_k^m used in defining Arnon’s A basis.)

- *Arnon's A basis.* For any pair of integers (m, k) with $m \geq k \geq 0$, let $X_k^m = \text{Sq}^{2^m} \text{Sq}^{2^{m-1}} \dots \text{Sq}^{2^k}$. The elements of Arnon's A basis are monomials $X_{k_0}^{m_0} \dots X_{k_t}^{m_t}$ with $(m_i, k_i) < (m_{i+1}, k_{i+1})$, ordered left lexicographically. (Note that X_k^m is the reverse of the element Q_k^m used in defining Wall's basis.)
- *Arnon's C basis.* The elements of Arnon's C basis are monomials of the form $\text{Sq}^{t_1} \dots \text{Sq}^{t_m}$ where for each i , we have $t_i \leq 2t_{i+1}$ and $2^i | t_{m-i}$.
- *P_t^s bases.* Let $p = 2$. For integers $s \geq 0$ and $t > 0$, the element P_t^s is the Milnor basis element $\mathcal{P}(0, \dots, 0, p^s, 0, \dots)$, with the nonzero entry in position t . To obtain a P_t^s -basis, for each set $\{P_{t_1}^{s_1}, \dots, P_{t_k}^{s_k}\}$ of (distinct) P_t^s 's, one chooses an ordering and forms the monomials

$$(P_{t_1}^{s_1})^{i_1} \dots (P_{t_k}^{s_k})^{i_k}$$

for all exponents i_j with $0 < i_j < p$. When $p = 2$, the set of all such monomials then forms a basis, and when p is odd, if one multiplies each such monomial on the left by products of the form $Q_{e_1} Q_{e_2} \dots$ with $0 \leq e_1 < e_2 < \dots$, one obtains a basis.

Thus one gets a basis by choosing an ordering on each set of P_t^s 's. There are infinitely many orderings possible, and we have implemented four of them:

- ‘rlex’: right lexicographic ordering
 - ‘llex’: left lexicographic ordering
 - ‘deg’: ordered by degree, which is the same as left lexicographic ordering on the pair $(s + t, t)$
 - ‘revz’: left lexicographic ordering on the pair $(s + t, s)$, which is the reverse of the ordering used (on elements in the same degrees as the P_t^s 's) in Wood's Z basis: ‘revz’ stands for ‘reversed Z’. This is the default: ‘pst’ is the same as ‘pst_revz’.
- *Commutator bases.* Let $c_{i,1} = \mathcal{P}(p^i)$, let $c_{i,2} = [c_{i+1,1}, c_{i,1}]$, and inductively define $c_{i,k} = [c_{i+k-1,1}, c_{i,k-1}]$. Thus $c_{i,k}$ is a k -fold iterated commutator of the elements $\mathcal{P}(p^i), \dots, \mathcal{P}(p^{i+k-1})$. Note that $\dim c_{i,k} = \dim P_k^i$. Commutator bases are obtained in much the same way as P_t^s -bases: for each set $\{c_{s_1,t_1}, \dots, c_{s_k,t_k}\}$ of (distinct) $c_{s,t}$'s, one chooses an ordering and forms the resulting monomials

$$c_{s_1,t_1}^{i_1} \dots c_{s_k,t_k}^{i_k}$$

for all exponents i_j with $0 < i_j < p$. When p is odd, one also needs to left-multiply by products of the Q_i 's. As for P_t^s -bases, every ordering on each set of iterated commutators determines a basis, and the same four orderings have been defined for these bases as for the P_t^s bases: ‘rlex’, ‘llex’, ‘deg’, ‘revz’.

Sub-Hopf algebras of the Steenrod algebra

The sub-Hopf algebras of the Steenrod algebra have been classified. Milnor proved that at the prime 2, the dual of the Steenrod algebra A_* is isomorphic to a polynomial algebra

$$A_* \cong \mathbb{F}_2[\xi_1, \xi_2, \xi_3, \dots].$$

The Milnor basis is dual to the monomial basis. Furthermore, any sub-Hopf algebra corresponds to a quotient of this of the form

$$A_*/(\xi_1^{2^{e_1}}, \xi_2^{2^{e_2}}, \xi_3^{2^{e_3}}, \dots).$$

The list of exponents (e_1, e_2, \dots) may be considered a function e from the positive integers to the extended non-negative integers (the non-negative integers and ∞); this is called the *profile function* for the sub-Hopf algebra. The profile function must satisfy the condition

- $e(r) \geq \min(e(r-i) - i, e(i))$ for all $0 < i < r$.

At odd primes, the situation is similar: the dual is isomorphic to the tensor product of a polynomial algebra and an exterior algebra,

$$A_* = \mathbf{F}_p[\xi_1, \xi_2, \xi_3, \dots] \otimes \Lambda(\tau_0, \tau_1, \dots),$$

and any sub-Hopf algebra corresponds to a quotient of this of the form

$$A_*/(\xi_1^{p^{e_1}}, \xi_2^{p^{e_2}}, \dots; \tau_0^{k_0}, \tau_1^{k_1}, \dots).$$

Here the profile function has two pieces, e as at the prime 2, and k , which maps the non-negative integers to the set $\{1, 2\}$. These must satisfy the following conditions:

- $e(r) \geq \min(e(r-i) - i, e(i))$ for all $0 < i < r$.
- if $k(i+j) = 1$, then either $e(i) \leq j$ or $k(j) = 1$ for all $i \geq 1, j \geq 0$.

(See Adams-Margolis [AM1974], for example, for these results on profile functions.)

This module allows one to construct the Steenrod algebra or any of its sub-Hopf algebras, at any prime. When defining a sub-Hopf algebra, you must work with the Milnor basis or a P_t^s -basis.

Elements of the Steenrod algebra

Basic arithmetic, $p = 2$. To construct an element of the mod 2 Steenrod algebra, use the function `Sq` :

```
sage: a = Sq(1, 2)
sage: b = Sq(4, 1)
sage: z = a + b
sage: z
Sq(1, 2) + Sq(4, 1)
sage: Sq(4) * Sq(1, 2)
Sq(1, 1, 1) + Sq(2, 3) + Sq(5, 2)
sage: z**2 # non-negative exponents work as they should
Sq(1, 2, 1) + Sq(4, 1, 1)
sage: z**0
1
```

Basic arithmetic, $p > 2$. To construct an element of the mod p Steenrod algebra when p is odd, you should first define a Steenrod algebra, using the `SteenrodAlgebra` command:

```
sage: A3 = SteenrodAlgebra(3)
```

Having done this, the newly created algebra `A3` has methods `Q` and `P` which construct elements of `A3` :

```
sage: c = A3.Q(1, 3, 6); c
Q_1 Q_3 Q_6
sage: d = A3.P(2, 0, 1); d
P(2, 0, 1)
sage: c * d
Q_1 Q_3 Q_6 P(2, 0, 1)
sage: e = A3.P(3)
sage: d * e
P(5, 0, 1)
sage: e * d
P(1, 1, 1) + P(5, 0, 1)
sage: c * c
```

```
0
sage: e ** 3
2 P(1,2)
```

Note that one can construct an element like `c` above in one step, without first constructing the algebra:

```
sage: c = SteenrodAlgebra(3).Q(1,3,6)
sage: c
Q_1 Q_3 Q_6
```

And of course, you can do similar constructions with the mod 2 Steenrod algebra:

```
sage: A = SteenrodAlgebra(2); A
mod 2 Steenrod algebra, milnor basis
sage: A.Sq(2,3,5)
Sq(2,3,5)
sage: A.P(2,3,5)    # when p=2, P = Sq
Sq(2,3,5)
sage: A.Q(1,4)      # when p=2, this gives a product of Milnor primitives
Sq(0,1,0,0,1)
```

Associated to each element is its prime (the characteristic of the underlying base field) and its basis (the basis for the Steenrod algebra in which it lies):

```
sage: a = SteenrodAlgebra(basis='milnor').Sq(1,2,1)
sage: a.prime()
2
sage: a.basis_name()
'milnor'
sage: a.degree()
14
```

It can be viewed in other bases:

```
sage: a.milnor() # same as a
Sq(1,2,1)
sage: a.change_basis('adem')
Sq^9 Sq^4 Sq^1 + Sq^11 Sq^2 Sq^1 + Sq^13 Sq^1
sage: a.change_basis('adem').change_basis('milnor')
Sq(1,2,1)
```

Regardless of the prime, each element has an *excess*, and if the element is homogeneous, a *degree*. The excess of $Sq(i_1, i_2, i_3, \dots)$ is $i_1 + i_2 + i_3 + \dots$; when p is odd, the excess of $Q_0^{e_0} Q_1^{e_1} \dots \mathcal{P}(r_1, r_2, \dots)$ is $\sum e_i + 2 \sum r_i$. The excess of a linear combination of Milnor basis elements is the minimum of the excesses of those basis elements.

The degree of $Sq(i_1, i_2, i_3, \dots)$ is $\sum (2^n - 1) i_n$, and when p is odd, the degree of $Q_0^{e_0} Q_1^{e_1} \dots \mathcal{P}(r_1, r_2, \dots)$ is $\sum \epsilon_i (2p^i - 1) + \sum r_j (2p^j - 2)$. The degree of a linear combination of such terms is only defined if the terms all have the same degree.

Here are some simple examples:

```
sage: z = Sq(1,2) + Sq(4,1)
sage: z.degree()
7
sage: (Sq(0,0,1) + Sq(5,3)).degree()
Traceback (most recent call last):
...
ValueError: Element is not homogeneous.
```

```

sage: Sq(7,2,1).excess()
10
sage: z.excess()
3
sage: B = SteenrodAlgebra(3)
sage: x = B.Q(1,4)
sage: y = B.P(1,2,3)
sage: x.degree()
166
sage: x.excess()
2
sage: y.excess()
12

```

Elements have a `weight` in the May filtration, which (when $p = 2$) is related to the `height` function defined by Wall:

```

sage: Sq(2,1,5).may_weight()
9
sage: Sq(2,1,5).wall_height()
[2, 3, 2, 1, 1]
sage: b = Sq(4)*Sq(8) + Sq(8)*Sq(4)
sage: b.may_weight()
2
sage: b.wall_height()
[0, 0, 1, 1]

```

Odd primary May weights:

```

sage: A5 = SteenrodAlgebra(5)
sage: a = A5.Q(1,2,4)
sage: b = A5.P(1,2,1)
sage: a.may_weight()
10
sage: b.may_weight()
8
sage: (a * b).may_weight()
18
sage: A5.P(0,0,1).may_weight()
3

```

Since the Steenrod algebra is a Hopf algebra, every element has a coproduct and an antipode:

```

sage: Sq(5).coproduct()
1 # Sq(5) + Sq(1) # Sq(4) + Sq(2) # Sq(3) + Sq(3) # Sq(2) + Sq(4) # Sq(1) + Sq(5) # 1
sage: Sq(5).antipode()
Sq(2,1) + Sq(5)
sage: d = Sq(0,0,1); d
Sq(0,0,1)
sage: d.antipode()
Sq(0,0,1)
sage: Sq(4).antipode()
Sq(1,1) + Sq(4)
sage: (Sq(4) * Sq(2)).antipode()
Sq(6)
sage: SteenrodAlgebra(7).P(3,1).antipode()
P(3,1)

```

Applying the antipode twice returns the original element:

```
sage: y = Sq(8)*Sq(4)
sage: y == (y.antipode()).antipode()
True
```

Internal representation: you can use any element as an iterator (for x in a: ...), and the method `monomial_coefficients()` returns a dictionary with keys tuples representing basis elements and with corresponding value representing the coefficient of that term:

```
sage: c = Sq(5).antipode(); c
Sq(2,1) + Sq(5)
sage: for mono, coeff in c: print((coeff, mono))
(1, (5,))
(1, (2, 1))
sage: c.monomial_coefficients()
{(2, 1): 1, (5,): 1}
sage: sorted(c.monomials(), key=lambda x: x.support())
[Sq(2,1), Sq(5)]
sage: sorted(c.support())
[(2, 1), (5,)]
sage: Adem = SteenrodAlgebra(basis='adem')
sage: elt = Adem.Sq(10) + Adem.Sq(9) * Adem.Sq(1)
sage: sorted(elt.monomials(), key=lambda x: x.support())
[Sq^9 Sq^1, Sq^10]

sage: A7 = SteenrodAlgebra(p=7)
sage: a = A7.P(1) * A7.P(1); a
2 P(2)
sage: a.leading_coefficient()
2
sage: a.leading_monomial()
P(2)
sage: a.leading_term()
2 P(2)
sage: a.change_basis('adem').monomial_coefficients()
{(0, 2, 0): 2}
```

The tuple in the previous output stands for the element $\beta^0 P^2 \beta^0$, i.e., P^2 . Going in the other direction, if you want to specify a basis element by giving the corresponding tuple, you can use the `monomial()` method on the algebra:

```
sage: SteenrodAlgebra(p=7, basis='adem').monomial((0, 2, 0))
P^2
sage: 10 * SteenrodAlgebra(p=7, basis='adem').monomial((0, 2, 0))
3 P^2
```

In the following example, elements in Wood's Z basis are certain products of the elements $w(m, k) = \text{Sq}^{2^m(2^{k+1}-1)}$. Internally, each $w(m, k)$ is represented by the pair (m, k) , and products of them are represented by tuples of such pairs.

```
sage: A = SteenrodAlgebra(basis='wood_z')
sage: t = ((2, 0), (0, 0))
sage: A.monomial(t)
Sq^4 Sq^1
```

See the documentation for `SteenrodAlgebra()` for more details and examples.

```
sage.algebras.steenrod.steenrod_algebra.AA (n=None, p=2)
This returns the Steenrod algebra A or its sub-Hopf algebra A(n).
```

INPUT:

- n - non-negative integer, optional (default None)
- p - prime number, optional (default 2)

OUTPUT: If n is None, then return the full Steenrod algebra. Otherwise, return $A(n)$.

When $p = 2$, $A(n)$ is the sub-Hopf algebra generated by the elements Sq^i for $i \leq 2^n$. Its profile function is $(n+1, n, n-1, \dots)$. When p is odd, $A(n)$ is the sub-Hopf algebra generated by the elements Q_0 and \mathcal{P}^i for $i \leq p^{n-1}$. Its profile function is $e = (n, n-1, n-2, \dots)$ and $k = (2, 2, \dots, 2)$ (length $n+1$).

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra import AA as A
sage: A()
mod 2 Steenrod algebra, milnor basis
sage: A(2)
sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function [3, 2, 1]
sage: A(2, p=5)
sub-Hopf algebra of mod 5 Steenrod algebra, milnor basis, profile function ([2, 1], [2, 2, 2])
```

`sage.algebras.steenrod.steenrod_algebra.Sq (*nums)`
Milnor element $Sq(a, b, c, \dots)$.

INPUT:

- a, b, c, \dots - non-negative integers

OUTPUT: element of the Steenrod algebra

This returns the Milnor basis element $Sq(a, b, c, \dots)$.

EXAMPLES:

```
sage: Sq(5)
Sq(5)
sage: Sq(5) + Sq(2,1) + Sq(5) # addition is mod 2:
Sq(2,1)
sage: (Sq(4,3) + Sq(7,2)).degree()
13
```

Entries must be non-negative integers; otherwise, an error results.

This function is a good way to define elements of the Steenrod algebra.

```
sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra ( p=2, basis='milnor',
generic='auto',
**kwds)
```

The mod p Steenrod algebra

INPUT:

- p - positive prime integer (optional, default = 2)
- $basis$ - string (optional, default = 'milnor')
- $profile$ - a profile function in form specified below (optional, default None)
- $truncation_type$ - 0 or ∞ or 'auto' (optional, default 'auto')
- $precision$ - integer or None (optional, default None)

•generic - (optional, default 'auto')

OUTPUT: mod p Steenrod algebra or one of its sub-Hopf algebras, elements of which are printed using `basis`

See below for information about `basis`, `profile`, etc.

EXAMPLES:

Some properties of the Steenrod algebra are available:

```
sage: A = SteenrodAlgebra(2)
sage: A.order()
+Infinity
sage: A.is_finite()
False
sage: A.is_commutative()
False
sage: A.is_noetherian()
False
sage: A.is_integral_domain()
False
sage: A.is_field()
False
sage: A.is_division_algebra()
False
sage: A.category()
Category of graded hopf algebras with basis over Finite Field of size 2
```

There are methods for constructing elements of the Steenrod algebra:

```
sage: A2 = SteenrodAlgebra(2); A2
mod 2 Steenrod algebra, milnor basis
sage: A2.Sq(1,2,6)
Sq(1,2,6)
sage: A2.Q(3,4) # product of Milnor primitives Q_3 and Q_4
Sq(0,0,0,1,1)
sage: A2.pst(2,3) # Margolis pst element
Sq(0,0,4)
sage: A5 = SteenrodAlgebra(5); A5
mod 5 Steenrod algebra, milnor basis
sage: A5.P(1,2,6)
P(1,2,6)
sage: A5.Q(3,4)
Q_3 Q_4
sage: A5.Q(3,4) * A5.P(1,2,6)
Q_3 Q_4 P(1,2,6)
sage: A5.pst(2,3)
P(0,0,25)
```

You can test whether elements are contained in the Steenrod algebra:

```
sage: w = Sq(2) * Sq(4)
sage: w in SteenrodAlgebra(2)
True
sage: w in SteenrodAlgebra(17)
False
```

Different bases for the Steenrod algebra:

There are two standard vector space bases for the mod p Steenrod algebra: the Milnor basis and the Serre-Cartan basis. When $p = 2$, there are also several other, less well-known, bases. See the documentation for this module (type `sage.algebras.steenrod.steenrod_algebra?`) and the function `steenrod_algebra_basis` for full descriptions of each of the implemented bases.

This module implements the following bases at all primes:

- ‘milnor’: Milnor basis.
- ‘serre-cartan’ or ‘adem’ or ‘admissible’: Serre-Cartan basis.
- ‘pst’, ‘pst_rlex’, ‘pst_llex’, ‘pst_deg’, ‘pst_revz’: various P_t^s -bases.
- ‘comm’, ‘comm_rlex’, ‘comm_llex’, ‘comm_deg’, ‘comm_revz’, or these with ‘_long’ appended: various commutator bases.

It implements the following bases when $p = 2$:

- ‘wood_y’: Wood’s Y basis.
- ‘wood_z’: Wood’s Z basis.
- ‘wall’, ‘wall_long’: Wall’s basis.
- ‘arnon_a’, ‘arnon_a_long’: Arnon’s A basis.
- ‘arnon_c’: Arnon’s C basis.

When defining a Steenrod algebra, you can specify a basis. Then elements of that Steenrod algebra are printed in that basis:

```
sage: adem = SteenrodAlgebra(2, 'adem')
sage: x = adem.Sq(2,1) # Sq(-) always means a Milnor basis element
sage: x
Sq^4 Sq^1 + Sq^5
sage: y = Sq(0,1) # unadorned Sq defines elements w.r.t. Milnor basis
sage: y
Sq(0,1)
sage: adem(y)
Sq^2 Sq^1 + Sq^3
sage: adem5 = SteenrodAlgebra(5, 'serre-cartan')
sage: adem5.P(0,2)
P^10 P^2 + 4 P^11 P^1 + P^12
```

If you add or multiply elements defined using different bases, the left-hand factor determines the form of the output:

```
sage: SteenrodAlgebra(basis='adem').Sq(3) + SteenrodAlgebra(basis='pst').Sq(0,1)
Sq^2 Sq^1
sage: SteenrodAlgebra(basis='pst').Sq(3) + SteenrodAlgebra(basis='milnor').Sq(0,1)
P^0_1 P^1_1 + P^0_2
sage: SteenrodAlgebra(basis='milnor').Sq(2) * SteenrodAlgebra(basis='arnonc').
→ Sq(2)
Sq(1,1)
```

You can get a list of basis elements in a given dimension:

```
sage: A3 = SteenrodAlgebra(3, 'milnor')
sage: A3.basis(13)
Family (Q_1 P(2), Q_0 P(3))
```


Algebras defined over different bases are not equal:

```
sage: SteenrodAlgebra(basis='milnor') == SteenrodAlgebra(basis='pst')
False
```

Bases have various synonyms, and in general Sage tries to figure out what basis you meant:

```
sage: SteenrodAlgebra(basis='MiLNOor')
mod 2 Steenrod algebra, milnor basis
sage: SteenrodAlgebra(basis='MiLNOor') == SteenrodAlgebra(basis='milnor')
True
sage: SteenrodAlgebra(basis='adem')
mod 2 Steenrod algebra, serre-cartan basis
sage: SteenrodAlgebra(basis='adem').basis_name()
'serre-cartan'
sage: SteenrodAlgebra(basis='wood---z---').basis_name()
'woodz'
```

As noted above, several of the bases ('arnon_a', 'wall', 'comm') have alternate, sometimes longer, representations. These provide ways of expressing elements of the Steenrod algebra in terms of the Sq^{2^n} .

```
sage: A_long = SteenrodAlgebra(2, 'arnon_a_long')
sage: A_long(Sq(6))
Sq^1 Sq^2 Sq^1 Sq^2 + Sq^2 Sq^4
sage: SteenrodAlgebra(2, 'wall_long')(Sq(6))
Sq^2 Sq^1 Sq^2 Sq^1 + Sq^2 Sq^4
sage: SteenrodAlgebra(2, 'comm_deg_long')(Sq(6))
s_1 s_2 s_12 + s_2 s_4
```

Sub-Hopf algebras of the Steenrod algebra:

These are specified using the argument `profile`, along with, optionally, `truncation_type` and `precision`. The `profile` argument specifies the profile function for this algebra. Any sub-Hopf algebra of the Steenrod algebra is determined by its *profile function*. When $p = 2$, this is a map e from the positive integers to the set of non-negative integers, plus ∞ , corresponding to the sub-Hopf algebra dual to this quotient of the dual Steenrod algebra:

$$\mathbf{F}_2[\xi_1, \xi_2, \xi_3, \dots] / (\xi_1^{2^{e(1)}}, \xi_2^{2^{e(2)}}, \xi_3^{2^{e(3)}}, \dots).$$

The profile function e must satisfy the condition

$$\bullet e(r) \geq \min(e(r-i) - i, e(i)) \text{ for all } 0 < i < r.$$

This is specified via `profile`, and optionally `precision` and `truncation_type`. First, `profile` must have one of the following forms:

- a list or tuple, e.g., `[3, 2, 1]`, corresponding to the function sending 1 to 3, 2 to 2, 3 to 1, and all other integers to the value of `truncation_type`.
- a function from positive integers to non-negative integers (and ∞), e.g., `lambda n: n+2`.
- `None` or `Infinity` - use this for the profile function for the whole Steenrod algebra.

In the first and third cases, `precision` is ignored. In the second case, this function is converted to a tuple of length one less than `precision`, which has default value 100. The function is truncated at this point, and all remaining values are set to the value of `truncation_type`.

`truncation_type` may be 0, ∞ , or 'auto'. If it's 'auto', then it gets converted to 0 in the first case above (when `profile` is a list), and otherwise (when `profile` is a function, `None`, or `Infinity`) it gets converted to ∞ .

For example, the sub-Hopf algebra $A(2)$ has profile function $[3, 2, 1, 0, 0, 0, \dots]$, so it can be defined by any of the following:

```
sage: A2 = SteenrodAlgebra(profile=[3,2,1])
sage: B2 = SteenrodAlgebra(profile=[3,2,1,0,0]) # trailing 0's ignored
sage: A2 == B2
True
sage: C2 = SteenrodAlgebra(profile=lambda n: max(4-n, 0), truncation_type=0)
sage: A2 == C2
True
```

In the following case, the profile function is specified by a function and `truncation_type` isn't specified, so it defaults to ∞ ; therefore this gives a different sub-Hopf algebra:

```
sage: D2 = SteenrodAlgebra(profile=lambda n: max(4-n, 0))
sage: A2 == D2
False
sage: D2.is_finite()
False
sage: E2 = SteenrodAlgebra(profile=lambda n: max(4-n, 0), truncation_
    ↪type=Infinity)
sage: D2 == E2
True
```

The argument `precision` only needs to be specified if the profile function is defined by a function and you want to control when the profile switches from the given function to the truncation type. For example:

```
sage: D3 = SteenrodAlgebra(profile=lambda n: n, precision=3)
sage: D3
sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function [1, 2, ↪
    ↪+Infinity, +Infinity, +Infinity, ...]
sage: D4 = SteenrodAlgebra(profile=lambda n: n, precision=4); D4
sub-Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function [1, 2, ↪
    ↪3, +Infinity, +Infinity, +Infinity, ...]
sage: D3 == D4
False
```

When p is odd, `profile` is a pair of functions e and k , corresponding to the quotient

$$\mathbf{F}_p[\xi_1, \xi_2, \xi_3, \dots] \otimes \Lambda(\tau_0, \tau_1, \dots) / (\xi_1^{p^{e_1}}, \xi_2^{p^{e_2}}, \dots; \tau_0^{k_0}, \tau_1^{k_1}, \dots).$$

Together, the functions e and k must satisfy the conditions

- $e(r) \geq \min(e(r-i) - i, e(i))$ for all $0 < i < r$,
- if $k(i+j) = 1$, then either $e(i) \leq j$ or $k(j) = 1$ for all $i \geq 1, j \geq 0$.

Therefore `profile` must have one of the following forms:

- a pair of lists or tuples, the second of which takes values in the set $\{1, 2\}$, e.g., $([3, 2, 1, 1], [1, 1, 2, 2, 1])$.
- a pair of functions, one from the positive integers to non-negative integers (and ∞), one from the non-negative integers to the set $\{1, 2\}$, e.g., $(\text{lambda } n: n+2, \text{lambda } n: 1 \text{ if } n < 3 \text{ else } 2)$.
- `None` or `Infinity` - use this for the profile function for the whole Steenrod algebra.

You can also mix and match the first two, passing a pair with first entry a list and second entry a function, for instance. The values of `precision` and `truncation_type` are determined by the first entry.

More examples:

```
sage: E = SteenrodAlgebra(profile=lambda n: 0 if n<3 else 3, truncation_type=0)
sage: E.is_commutative()
True

sage: A2 = SteenrodAlgebra(profile=[3,2,1]) # the algebra A(2)
sage: Sq(7,3,1) in A2
True
sage: Sq(8) in A2
False
sage: Sq(8) in SteenrodAlgebra().basis(8)
True
sage: Sq(8) in A2.basis(8)
False
sage: A2.basis(8)
Family (Sq(1,0,1), Sq(2,2), Sq(5,1))

sage: A5 = SteenrodAlgebra(p=5)
sage: A51 = SteenrodAlgebra(p=5, profile=([1], [2,2]))
sage: A5.Q(0,1) * A5.P(4) in A51
True
sage: A5.Q(2) in A51
False
sage: A5.P(5) in A51
False
```

For sub-Hopf algebras of the Steenrod algebra, only the Milnor basis or the various P_t^s -bases may be used.

```
sage: SteenrodAlgebra(profile=[1,2,1,1], basis='adem')
Traceback (most recent call last):
...
NotImplementedError: For sub-Hopf algebras of the Steenrod algebra, only the
↳Milnor basis and the pst bases are implemented.
```

The generic Steenrod algebra at the prime 2:

The structure formulas for the Steenrod algebra at odd primes p also make sense when p is set to 2. We refer to the resulting algebra as the “generic Steenrod algebra” for the prime 2. The dual Hopf algebra is given by

$$A_* = \mathbf{F}_2[\xi_1, \xi_2, \xi_3, \dots] \otimes \Lambda(\tau_0, \tau_1, \dots)$$

The degree of ξ_k is $2^{k+1} - 2$ and the degree of τ_k is $2^{k+1} - 1$.

The generic Steenrod algebra is an associated graded algebra of the usual Steenrod algebra that is occasionally useful. Its cohomology, for example, is the E_2 -term of a spectral sequence that computes the E_2 -term of the Novikov spectral sequence. It can also be obtained as a specialisation of Voevodsky’s “motivic Steenrod algebra”: in the notation of [Voe2003], Remark 12.12, it corresponds to setting $\rho = \tau = 0$. The usual Steenrod algebra is given by $\rho = 0$ and $\tau = 1$.

In Sage this algebra is constructed using the ‘generic’ keyword.

Example:

```

sage: EA = SteenrodAlgebra(p=2, generic=True) ; EA
generic mod 2 Steenrod algebra, milnor basis
sage: EA[8]
Vector space spanned by (Q_0 Q_2, Q_0 Q_1 P(2), P(1,1), P(4)) over Finite Field_
→ of size 2

```

```

class sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic ( p=2,
                                                                    ba-
                                                                    sis='milnor',
                                                                    **kws)

```

Bases: `sage.combinat.free_module.CombinatorialFreeModule`

The mod p Steenrod algebra.

Users should not call this, but use the function `SteenrodAlgebra()` instead. See that function for extensive documentation.

EXAMPLES:

```

sage: sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic()
mod 2 Steenrod algebra, milnor basis
sage: sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic(5)
mod 5 Steenrod algebra, milnor basis
sage: sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic(5, 'adem')
mod 5 Steenrod algebra, serre-cartan basis

```

```

class Element ( M, x)

```

Bases: `sage.combinat.free_module.CombinatorialFreeModuleElement`

Class for elements of the Steenrod algebra. Since the Steenrod algebra class is based on `CombinatorialFreeModule`, this is based on `CombinatorialFreeModuleElement`. It has new methods reflecting its role, like `degree()` for computing the degree of an element.

EXAMPLES:

Since this class inherits from `CombinatorialFreeModuleElement`, elements can be used as iterators, and there are other useful methods:

```

sage: c = Sq(5).antipode(); c
Sq(2,1) + Sq(5)
sage: for mono, coeff in c: print((coeff, mono))
(1, (5,))
(1, (2, 1))
sage: c.monomial_coefficients()
{(2, 1): 1, (5,): 1}
sage: sorted(c.monomials(), key=lambda x: x.support())
[Sq(2,1), Sq(5)]
sage: sorted(c.support())
[(2, 1), (5,)]

```

See the documentation for this module (type `sage.algebras.steenrod.steenrod_algebra?`) for more information about elements of the Steenrod algebra.

```

additive_order ( )

```

The additive order of any nonzero element of the mod p Steenrod algebra is p .

OUTPUT: 1 (for the zero element) or p (for anything else)

EXAMPLES:

```

sage: z = Sq(4) + Sq(6) + 1
sage: z.additive_order()
2
sage: (Sq(3) + Sq(3)).additive_order()
1

```

basis_name ()

The basis name associated to self.

EXAMPLES:

```

sage: a = SteenrodAlgebra().Sq(3,2,1)
sage: a.basis_name()
'milnor'
sage: a.change_basis('adem').basis_name()
'serre-cartan'
sage: a.change_basis('wood____y').basis_name()
'woody'
sage: b = SteenrodAlgebra(p=7).basis(36)[0]
sage: b.basis_name()
'milnor'
sage: a.change_basis('adem').basis_name()
'serre-cartan'

```

change_basis (basis='milnor')

Representation of element with respect to basis.

INPUT:

- *basis* - string, basis in which to work.

OUTPUT: representation of self in given basis

The choices for *basis* are:

- 'milnor' for the Milnor basis.
- 'serre-cartan', 'serre_cartan', 'sc', 'adem', 'admissible' for the Serre-Cartan basis.
- 'wood_y' for Wood's Y basis.
- 'wood_z' for Wood's Z basis.
- 'wall' for Wall's basis.
- 'wall_long' for Wall's basis, alternate representation
- 'arnon_a' for Arnon's A basis.
- 'arnon_a_long' for Arnon's A basis, alternate representation.
- 'arnon_c' for Arnon's C basis.
- 'pst', 'pst_rlex', 'pst_llex', 'pst_deg', 'pst_revz' for various P_t^s -bases.
- 'comm', 'comm_rlex', 'comm_llex', 'comm_deg', 'comm_revz' for various commutator bases.
- 'comm_long', 'comm_rlex_long', etc., for commutator bases, alternate representations.

See documentation for this module (by browsing the reference manual or by typing `sage.algebras.steenrod.steenrod_algebra?`) for descriptions of the different bases.

EXAMPLES:

```

sage: c = Sq(2) * Sq(1)
sage: c.change_basis('milnor')
Sq(0,1) + Sq(3)
sage: c.change_basis('serre-cartan')
Sq^2 Sq^1
sage: d = Sq(0,0,1)
sage: d.change_basis('arnonc')
Sq^2 Sq^5 + Sq^4 Sq^2 Sq^1 + Sq^4 Sq^3 + Sq^7

```

coproduct (*algorithm*='milnor')

The coproduct of this element.

INPUT:

- *algorithm* – None or a string, either 'milnor' or 'serre-cartan' (or anything which will be converted to one of these by the function `get_basis_name`). If None, default to 'serre-cartan' if current basis is 'serre-cartan'; otherwise use 'milnor'.

See `SteenrodAlgebra_generic.coproduct_on_basis()` for more information on computing the coproduct.

EXAMPLES:

```
sage: a = Sq(2)
sage: a.coproduct()
1 # Sq(2) + Sq(1) # Sq(1) + Sq(2) # 1
sage: b = Sq(4)
sage: (a*b).coproduct() == (a.coproduct()) * (b.coproduct())
True

sage: c = a.change_basis('adem'); c.coproduct(algorithm='milnor')
1 # Sq^2 + Sq^1 # Sq^1 + Sq^2 # 1
sage: c = a.change_basis('adem'); c.coproduct(algorithm='adem')
1 # Sq^2 + Sq^1 # Sq^1 + Sq^2 # 1

sage: d = a.change_basis('comm_long'); d.coproduct()
1 # s_2 + s_1 # s_1 + s_2 # 1

sage: A7 = SteenrodAlgebra(p=7)
sage: a = A7.Q(1) * A7.P(1); a
Q_1 P(1)
sage: a.coproduct()
1 # Q_1 P(1) + P(1) # Q_1 + Q_1 # P(1) + Q_1 P(1) # 1
sage: a.coproduct(algorithm='adem')
1 # Q_1 P(1) + P(1) # Q_1 + Q_1 # P(1) + Q_1 P(1) # 1
```

degree ()

The degree of self.

The degree of $Sq(i_1, i_2, i_3, \dots)$ is

$$i_1 + 3i_2 + 7i_3 + \dots + (2^k - 1)i_k + \dots$$

At an odd prime p , the degree of Q_k is $2p^k - 1$ and the degree of $\mathcal{P}(i_1, i_2, \dots)$ is

$$\sum_{k \geq 0} 2(p^k - 1)i_k.$$

ALGORITHM: If `is_homogeneous()` returns True, call `SteenrodAlgebra_generic.degree_on_basis()` on the leading summand.

EXAMPLES:

```
sage: Sq(0,0,1).degree()
7
sage: (Sq(0,0,1) + Sq(7)).degree()
7
sage: (Sq(0,0,1) + Sq(2)).degree()
Traceback (most recent call last):
...
```

```
ValueError: Element is not homogeneous.
```

```
sage: A11 = SteenrodAlgebra(p=11)
sage: A11.P(1).degree()
20
sage: A11.P(1,1).degree()
260
sage: A11.Q(2).degree()
241
```

excess ()

Excess of element.

OUTPUT: `excess` - non-negative integer

The excess of a Milnor basis element $Sq(a, b, c, \dots)$ is $a + b + c + \dots$. When p is odd, the excess of $Q_0^{e_0} Q_1^{e_1} \cdots P(r_1, r_2, \dots)$ is $\sum e_i + 2 \sum r_i$. The excess of a linear combination of Milnor basis elements is the minimum of the excesses of those basis elements.

See [Kr1971] for the proofs of these assertions.

EXAMPLES:

```
sage: a = Sq(1,2,3)
sage: a.excess()
6
sage: (Sq(0,0,1) + Sq(4,1) + Sq(7)).excess()
1
sage: elt = Sq(0,0,1) + Sq(4,1) + Sq(7)
sage: M = sorted(elt.monomials(), key=lambda x: x.support())
sage: [m.excess() for m in M]
[1, 5, 7]
sage: [m for m in M]
[Sq(0,0,1), Sq(4,1), Sq(7)]
sage: B = SteenrodAlgebra(7)
sage: a = B.Q(1,2,5)
sage: b = B.P(2,2,3)
sage: a.excess()
3
sage: b.excess()
14
sage: (a + b).excess()
3
sage: (a * b).excess()
17
```

is_decomposable ()

Return True if element is decomposable, False otherwise. That is, if element is in the square of the augmentation ideal, return True; otherwise, return False.

OUTPUT: boolean

EXAMPLES:

```
sage: a = Sq(6)
sage: a.is_decomposable()
True
sage: for i in range(9):
....:     if not Sq(i).is_decomposable():
....:         print(Sq(i))
```

```

1
Sq(1)
Sq(2)
Sq(4)
Sq(8)
sage: A3 = SteenrodAlgebra(p=3, basis='adem')
sage: [A3.P(n) for n in range(30) if not A3.P(n).is_decomposable()]
[1, P^1, P^3, P^9, P^27]

```

is_homogeneous ()

Return True iff this element is homogeneous.

EXAMPLES:

```

sage: (Sq(0,0,1) + Sq(7)).is_homogeneous()
True
sage: (Sq(0,0,1) + Sq(2)).is_homogeneous()
False

```

is_nilpotent ()

True if element is not a unit, False otherwise.

EXAMPLES:

```

sage: z = Sq(4,2) + Sq(7,1) + Sq(3,0,1)
sage: z.is_nilpotent()
True
sage: u = 1 + Sq(3,1)
sage: u == 1 + Sq(3,1)
True
sage: u.is_nilpotent()
False

```

is_unit ()

True if element has a nonzero scalar multiple of $P(0)$ as a summand, False otherwise.

EXAMPLES:

```

sage: z = Sq(4,2) + Sq(7,1) + Sq(3,0,1)
sage: z.is_unit()
False
sage: u = Sq(0) + Sq(3,1)
sage: u == 1 + Sq(3,1)
True
sage: u.is_unit()
True
sage: A5 = SteenrodAlgebra(5)
sage: v = A5.P(0)
sage: (v + v + v).is_unit()
True

```

may_weight ()

May's 'weight' of element.

OUTPUT: weight - non-negative integer

If we let $F_*(A)$ be the May filtration of the Steenrod algebra, the weight of an element x is the integer k so that x is in $F_k(A)$ and not in $F_{k+1}(A)$. According to Theorem 2.6 in May's thesis [May1964], the weight of a Milnor basis element is computed as follows: first, to compute the weight of $P(r_1, r_2, \dots)$,

write each r_i in base p as $r_i = \sum_j p^j r_{ij}$. Then each nonzero binary digit r_{ij} contributes i to the weight: the weight is $\sum_{i,j} i r_{ij}$. When p is odd, the weight of Q_i is $i + 1$, so the weight of a product $Q_{i_1} Q_{i_2} \dots$ equals $(i_1 + 1) + (i_2 + 1) + \dots$. Then the weight of $Q_{i_1} Q_{i_2} \dots P(r_1, r_2, \dots)$ is the sum of $(i_1 + 1) + (i_2 + 1) + \dots$ and $\sum_{i,j} i r_{ij}$.

The weight of a sum of Milnor basis elements is the minimum of the weights of the summands.

When $p = 2$, we compute the weight on Milnor basis elements by adding up the terms in their ‘height’ - see `wall_height()` for documentation. (When p is odd, the height of an element is not defined.)

EXAMPLES:

```
sage: Sq(0).may_weight()
0
sage: a = Sq(4)
sage: a.may_weight()
1
sage: b = Sq(4)*Sq(8) + Sq(8)*Sq(4)
sage: b.may_weight()
2
sage: Sq(2,1,5).wall_height()
[2, 3, 2, 1, 1]
sage: Sq(2,1,5).may_weight()
9
sage: A5 = SteenrodAlgebra(5)
sage: a = A5.Q(1,2,4)
sage: b = A5.P(1,2,1)
sage: a.may_weight()
10
sage: b.may_weight()
8
sage: (a * b).may_weight()
18
sage: A5.P(0,0,1).may_weight()
3
```

milnor ()

Return this element in the Milnor basis; that is, as an element of the appropriate Steenrod algebra.

This just calls the method `SteenrodAlgebra_generic.milnor()`.

EXAMPLES:

```
sage: Adem = SteenrodAlgebra(basis='adem')
sage: a = Adem.basis(4)[1]; a
Sq^3 Sq^1
sage: a.milnor()
Sq(1,1)
```

prime ()

The prime associated to self.

EXAMPLES:

```
sage: a = SteenrodAlgebra().Sq(3,2,1)
sage: a.prime()
2
sage: a.change_basis('adem').prime()
2
sage: b = SteenrodAlgebra(p=7).basis(36)[0]
```

```

sage: b.prime()
7
sage: SteenrodAlgebra(p=3, basis='adem').one().prime()
3

```

wall_height ()

Wall's 'height' of element.

OUTPUT: list of non-negative integers

The height of an element of the mod 2 Steenrod algebra is a list of non-negative integers, defined as follows: if the element is a monomial in the generators $Sq(2^i)$, then the i^{th} entry in the list is the number of times $Sq(2^i)$ appears. For an arbitrary element, write it as a sum of such monomials; then its height is the maximum, ordered right-lexicographically, of the heights of those monomials.

When p is odd, the height of an element is not defined.

According to Theorem 3 in [Wal1960], the height of the Milnor basis element $Sq(r_1, r_2, \dots)$ is obtained as follows: write each r_i in binary as $r_i = \sum_j 2^j r_{ij}$. Then each nonzero binary digit r_{ij} contributes 1 to the k^{th} entry in the height, for $j \leq k \leq i + j - 1$.

EXAMPLES:

```

sage: Sq(0).wall_height()
[]
sage: a = Sq(4)
sage: a.wall_height()
[0, 0, 1]
sage: b = Sq(4)*Sq(8) + Sq(8)*Sq(4)
sage: b.wall_height()
[0, 0, 1, 1]
sage: Sq(0,0,3).wall_height()
[1, 2, 2, 1]

```

SteenrodAlgebra_generic. P (*nums)

The element $P(a, b, c, \dots)$

INPUT:

• a, b, c, \dots - non-negative integers

OUTPUT: element of the Steenrod algebra given by the Milnor single basis element $P(a, b, c, \dots)$

Note that at the prime 2, this is the same element as $Sq(a, b, c, \dots)$.

EXAMPLES:

```

sage: A = SteenrodAlgebra(2)
sage: A.P(5)
Sq(5)
sage: B = SteenrodAlgebra(3)
sage: B.P(5,1,1)
P(5,1,1)
sage: B.P(1,1,-12,1)
Traceback (most recent call last):
...
TypeError: entries must be non-negative integers

sage: SteenrodAlgebra(basis='serre-cartan').P(0,1)
Sq^2 Sq^1 + Sq^3

```

```
sage: SteenrodAlgebra(generic=True).P(2,0,1)
P(2,0,1)
```

SteenrodAlgebra_generic. **Q** (*nums)

The element $Q_{n_0}Q_{n_1}\dots$, given by specifying the subscripts.

INPUT:

• n_0, n_1, \dots - non-negative integers

OUTPUT: The element $Q_{n_0}Q_{n_1}\dots$

Note that at the prime 2, Q_n is the element $Sq(0, 0, \dots, 1)$, where the 1 is in the $(n+1)^{st}$ position.

Compare this to the method `Q_exp()`, which defines a similar element, but by specifying the tuple of exponents.

EXAMPLES:

```
sage: A2 = SteenrodAlgebra(2)
sage: A2.Q(2,3)
Sq(0,0,1,1)
sage: A5 = SteenrodAlgebra(5)
sage: A5.Q(1,4)
Q_1 Q_4
sage: A5.Q(1,4) == A5.Q_exp(0,1,0,0,1)
True
sage: H = SteenrodAlgebra(p=5, profile=[[2,1], [2,2,2]])
sage: H.Q(2)
Q_2
sage: H.Q(4)
Traceback (most recent call last):
...
ValueError: Element not in this algebra
```

SteenrodAlgebra_generic. **Q_exp** (*nums)

The element $Q_0^{e_0}Q_1^{e_1}\dots$, given by specifying the exponents.

INPUT:

• e_0, e_1, \dots - sequence of 0s and 1s

OUTPUT: The element $Q_0^{e_0}Q_1^{e_1}\dots$

Note that at the prime 2, Q_n is the element $Sq(0, 0, \dots, 1)$, where the 1 is in the $(n+1)^{st}$ position.

Compare this to the method `Q()`, which defines a similar element, but by specifying the tuple of subscripts of terms with exponent 1.

EXAMPLES:

```
sage: A2 = SteenrodAlgebra(2)
sage: A5 = SteenrodAlgebra(5)
sage: A2.Q_exp(0,0,1,1,0)
Sq(0,0,1,1)
sage: A5.Q_exp(0,0,1,1,0)
Q_2 Q_3
sage: A5.Q(2,3)
Q_2 Q_3
sage: A5.Q_exp(0,0,1,1,0) == A5.Q(2,3)
True
```

```
sage: SteenrodAlgebra(2, generic=True).Q_exp(1, 0, 1)
Q_0 Q_2
```

SteenrodAlgebra_generic. **algebra_generators** ()

Family of generators for this algebra.

OUTPUT: family of elements of this algebra

At the prime 2, the Steenrod algebra is generated by the elements Sq^{2^i} for $i \geq 0$. At odd primes, it is generated by the elements Q_0 and \mathcal{P}^{p^i} for $i \geq 0$. So if this algebra is the entire Steenrod algebra, return an infinite family made up of these elements.

For sub-Hopf algebras of the Steenrod algebra, it is not always clear what a minimal generating set is. The sub-Hopf algebra $A(n)$ is minimally generated by the elements Sq^{2^i} for $0 \leq i \leq n$ at the prime 2. At odd primes, $A(n)$ is minimally generated by Q_0 along with \mathcal{P}^{p^i} for $0 \leq i \leq n - 1$. So if this algebra is $A(n)$, return the appropriate list of generators.

For other sub-Hopf algebras: return a non-minimal generating set: the family of P_t^s 's and Q_n 's contained in the algebra.

EXAMPLES:

```
sage: A3 = SteenrodAlgebra(3, 'adem')
sage: A3.gens()
Lazy family (<bound method SteenrodAlgebra_generic_with_category.gen of mod 3_
↳Steenrod algebra, serre-cartan basis>(i))_{i in Non negative integers}
sage: A3.gens()[0]
beta
sage: A3.gens()[1]
P^1
sage: A3.gens()[2]
P^3
sage: SteenrodAlgebra(profile=[3,2,1]).gens()
Family (Sq(1), Sq(2), Sq(4))
```

In the following case, return a non-minimal generating set. (It is not minimal because $Sq(0,0,1)$ is the commutator of $Sq(1)$ and $Sq(0,2)$.)

```
sage: SteenrodAlgebra(profile=[1,2,1]).gens()
Family (Sq(1), Sq(0,1), Sq(0,2), Sq(0,0,1))
sage: SteenrodAlgebra(p=5, profile=[[2,1], [2,2,2]]).gens()
Family (Q_0, P(1), P(5))
sage: SteenrodAlgebra(profile=lambda n: n).gens()
Lazy family (<bound method SteenrodAlgebra_mod_two_with_category.gen of sub-
↳Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function [1,
↳2, 3, ..., 98, 99, +Infinity, +Infinity, +Infinity, ...]>(i))_{i in Non_
↳negative integers}
```

You may also use `algebra_generators` instead of `gens` :

```
sage: SteenrodAlgebra(p=5, profile=[[2,1], [2,2,2]]).algebra_generators()
Family (Q_0, P(1), P(5))
```

SteenrodAlgebra_generic. **an_element** ()

An element of this Steenrod algebra.

The element depends on the basis and whether there is a nontrivial profile function. (This is used by the automatic test suite, so having different elements in different bases may help in discovering bugs.)

EXAMPLES:

```

sage: SteenrodAlgebra().an_element()
Sq(2,1)
sage: SteenrodAlgebra(basis='adem').an_element()
Sq^4 Sq^2 Sq^1
sage: SteenrodAlgebra(p=5).an_element()
4 Q_1 Q_3 P(2,1)
sage: SteenrodAlgebra(basis='pst').an_element()
P^3_1
sage: SteenrodAlgebra(basis='pst', profile=[3,2,1]).an_element()
P^0_1

```

SteenrodAlgebra_generic. **antipode_on_basis** (*t*)

The antipode of a basis element of this algebra

INPUT:

- *t* – tuple, the index of a basis element of self

OUTPUT: the antipode of the corresponding basis element, as an element of self.

ALGORITHM: according to a result of Milnor's, the antipode of $Sq(n)$ is the sum of all of the Milnor basis elements in dimension n . So: convert the element to the Serre-Cartan basis, thus writing it as a sum of products of elements $Sq(n)$, and use Milnor's formula for the antipode of $Sq(n)$, together with the fact that the antipode is an antihomomorphism: if we call the antipode c , then $c(ab) = c(b)c(a)$.

At odd primes, a similar method is used: the antipode of $P(n)$ is the sum of the Milnor P basis elements in dimension $n * 2(p - 1)$, multiplied by $(-1)^n$, and the antipode of $\beta = Q_0$ is $-Q_0$. So convert to the Serre-Cartan basis, as in the $p = 2$ case.

EXAMPLES:

```

sage: A = SteenrodAlgebra()
sage: A.antipode_on_basis((4,))
Sq(1,1) + Sq(4)
sage: A.Sq(4).antipode()
Sq(1,1) + Sq(4)
sage: Adem = SteenrodAlgebra(basis='adem')
sage: Adem.Sq(4).antipode()
Sq^3 Sq^1 + Sq^4
sage: SteenrodAlgebra(basis='pst').Sq(3).antipode()
P^0_1 P^1_1 + P^0_2
sage: a = SteenrodAlgebra(basis='wall_long').Sq(10)
sage: a.antipode()
Sq^1 Sq^2 Sq^4 Sq^1 Sq^2 + Sq^2 Sq^4 Sq^1 Sq^2 Sq^1 + Sq^8 Sq^2
sage: a.antipode().antipode() == a
True

sage: SteenrodAlgebra(p=3).P(6).antipode()
P(2,1) + P(6)
sage: SteenrodAlgebra(p=3).P(6).antipode().antipode()
P(6)

```

SteenrodAlgebra_generic. **basis** (*d=None*)

Returns basis for self, either the whole basis or the basis in degree *d*.

INPUT:

- *d* – integer or None, optional (default None)

OUTPUT: If d is None, then return a basis of the algebra. Otherwise, return the basis in degree d .

EXAMPLES:

```
sage: A3 = SteenrodAlgebra(3)
sage: A3.basis(13)
Family (Q_1 P(2), Q_0 P(3))
sage: SteenrodAlgebra(2, 'adem').basis(12)
Family (Sq^12, Sq^11 Sq^1, Sq^9 Sq^2 Sq^1, Sq^8 Sq^3 Sq^1, Sq^10 Sq^2, Sq^9
↪ Sq^3, Sq^8 Sq^4)

sage: A = SteenrodAlgebra(profile=[1,2,1])
sage: A.basis(2)
Family ()
sage: A.basis(3)
Family (Sq(0,1),)
sage: SteenrodAlgebra().basis(3)
Family (Sq(0,1), Sq(3))
sage: A_pst = SteenrodAlgebra(profile=[1,2,1], basis='pst')
sage: A_pst.basis(3)
Family (P^0_2,)

sage: A7 = SteenrodAlgebra(p=7)
sage: B = SteenrodAlgebra(p=7, profile=([1,2,1], [1]))
sage: A7.basis(84)
Family (P(7),)
sage: B.basis(84)
Family ()
sage: C = SteenrodAlgebra(p=7, profile=([1], [2,2]))
sage: A7.Q(0,1) in C.basis(14)
True
sage: A7.Q(2) in A7.basis(97)
True
sage: A7.Q(2) in C.basis(97)
False
```

With no arguments, return the basis of the whole algebra. This does not print in a very helpful way, unfortunately:

```
sage: A7.basis()
Lazy family (Term map from basis key family of mod 7 Steenrod algebra, milnor
↪ basis to mod 7 Steenrod algebra, milnor basis(i))_{i in basis key family of
↪ mod 7 Steenrod algebra, milnor basis}
sage: for (idx,a) in zip((1,..,9),A7.basis()):
....:     print("{} {}".format(idx, a))
1 1
2 Q_0
3 P(1)
4 Q_1
5 Q_0 P(1)
6 Q_0 Q_1
7 P(2)
8 Q_1 P(1)
9 Q_0 P(2)
sage: D = SteenrodAlgebra(p=3, profile=([1], [2,2]))
sage: sorted(D.basis())
[1, P(1), P(2), Q_0, Q_0 P(1), Q_0 P(2), Q_0 Q_1, Q_0 Q_1 P(1), Q_0 Q_1 P(2),
↪ Q_1, Q_1 P(1), Q_1 P(2)]
```

SteenrodAlgebra_generic.**basis_name** ()

The basis name associated to self.

EXAMPLES:

```
sage: SteenrodAlgebra(p=2, profile=[1,1]).basis_name()
'milnor'
sage: SteenrodAlgebra(basis='serre-cartan').basis_name()
'serre-cartan'
sage: SteenrodAlgebra(basis='adem').basis_name()
'serre-cartan'
```

SteenrodAlgebra_generic.**coproduct** (x, algorithm='milnor')

Return the coproduct of an element x of this algebra.

INPUT:

- x – element of self
- algorithm – None or a string, either 'milnor' or 'serre-cartan' (or anything which will be converted to one of these by the function `get_basis_name`). If None, default to 'serre-cartan' if current basis is 'serre-cartan'; otherwise use 'milnor'.

This calls `coproduct_on_basis()` on the summands of x and extends linearly.

EXAMPLES:

```
sage: SteenrodAlgebra().Sq(3).coproduct()
1 # Sq(3) + Sq(1) # Sq(2) + Sq(2) # Sq(1) + Sq(3) # 1
```

The element $Sq(0, 1)$ is primitive:

```
sage: SteenrodAlgebra(basis='adem').Sq(0,1).coproduct()
1 # Sq^2 Sq^1 + 1 # Sq^3 + Sq^2 Sq^1 # 1 + Sq^3 # 1
sage: SteenrodAlgebra(basis='pst').Sq(0,1).coproduct()
1 # P^0_2 + P^0_2 # 1

sage: SteenrodAlgebra(p=3).P(4).coproduct()
1 # P(4) + P(1) # P(3) + P(2) # P(2) + P(3) # P(1) + P(4) # 1
sage: SteenrodAlgebra(p=3).P(4).coproduct(algorithm='serre-cartan')
1 # P(4) + P(1) # P(3) + P(2) # P(2) + P(3) # P(1) + P(4) # 1
sage: SteenrodAlgebra(p=3, basis='serre-cartan').P(4).coproduct()
1 # P^4 + P^1 # P^3 + P^2 # P^2 + P^3 # P^1 + P^4 # 1
sage: SteenrodAlgebra(p=11, profile=(), (2,1,2)).Q(0,2).coproduct()
1 # Q_0 Q_2 + Q_0 # Q_2 + Q_0 Q_2 # 1 + 10*Q_2 # Q_0
```

SteenrodAlgebra_generic.**coproduct_on_basis** (t, algorithm=None)

The coproduct of a basis element of this algebra

INPUT:

- t – tuple, the index of a basis element of self
- algorithm – None or a string, either 'milnor' or 'serre-cartan' (or anything which will be converted to one of these by the function `get_basis_name`). If None, default to 'milnor' unless current basis is 'serre-cartan', in which case use 'serre-cartan'.

ALGORITHM: The coproduct on a Milnor basis element $P(n_1, n_2, \dots)$ is $\sum P(i_1, i_2, \dots) \otimes P(j_1, j_2, \dots)$, summed over all $i_k + j_k = n_k$ for each k . At odd primes, each element Q_n is primitive: its coproduct is $Q_n \otimes 1 + 1 \otimes Q_n$.

One can deduce a coproduct formula for the Serre-Cartan basis from this: the coproduct on each P^n is $\sum P^i \otimes P^{n-i}$ and at odd primes β is primitive. Since the coproduct is an algebra map, one can then compute the coproduct on any Serre-Cartan basis element.

Which of these methods is used is controlled by whether `algorithm` is ‘milnor’ or ‘serre-cartan’.

OUTPUT: the coproduct of the corresponding basis element, as an element of `self tensor self`.

EXAMPLES:

```
sage: A = SteenrodAlgebra()
sage: A.coproduct_on_basis((3,))
1 # Sq(3) + Sq(1) # Sq(2) + Sq(2) # Sq(1) + Sq(3) # 1
```

`SteenrodAlgebra_generic.counit_on_basis (t)`

The counit sends all elements of positive degree to zero.

INPUT:

- `t` – tuple, the index of a basis element of `self`

EXAMPLES:

```
sage: A2 = SteenrodAlgebra(p=2)
sage: A2.counit_on_basis(())
1
sage: A2.counit_on_basis((0,0,1))
0
sage: parent(A2.counit_on_basis((0,0,1)))
Finite Field of size 2
sage: A3 = SteenrodAlgebra(p=3)
sage: A3.counit_on_basis((1,2,3), (1,1,1))
0
sage: A3.counit_on_basis((), ())
1
sage: A3.counit(A3.P(10,5))
0
sage: A3.counit(A3.P(0))
1
```

`SteenrodAlgebra_generic.degree_on_basis (t)`

The degree of the monomial specified by the tuple `t`.

INPUT:

- `t` - tuple, representing basis element in the current basis.

OUTPUT: integer, the degree of the corresponding element.

The degree of $Sq(i_1, i_2, i_3, \dots)$ is

$$i_1 + 3i_2 + 7i_3 + \dots + (2^k - 1)i_k + \dots$$

At an odd prime p , the degree of Q_k is $2p^k - 1$ and the degree of $\mathcal{P}(i_1, i_2, \dots)$ is

$$\sum_{k \geq 0} 2(p^k - 1)i_k.$$

ALGORITHM: Each basis element is represented in terms relevant to the particular basis: ‘milnor’ basis elements (at the prime 2) are given by tuples (a, b, c, \dots) corresponding to the element $Sq(a, b, c, \dots)$, while ‘pst’ basis elements are given by tuples of pairs $((a, b), (c, d), \dots)$, corresponding to the product $P_b^a P_d^c \dots$. The other bases have similar descriptions. The degree of each basis element is computed

from this data, rather than converting the element to the Milnor basis, for example, and then computing the degree.

EXAMPLES:

```
sage: SteenrodAlgebra().degree_on_basis((0,0,1))
7
sage: Sq(7).degree()
7

sage: A11 = SteenrodAlgebra(p=11)
sage: A11.degree_on_basis(((1,1)))
260
sage: A11.degree_on_basis(((2,)), ()))
241
```

`SteenrodAlgebra_generic.dimension()`

The dimension of this algebra as a vector space over \mathbf{F}_p .

If the algebra is infinite, return `+Infinity`. Otherwise, the profile function must be finite. In this case, at the prime 2, its dimension is 2^s , where s is the sum of the entries in the profile function. At odd primes, the dimension is $p^s * 2^t$ where s is the sum of the e component of the profile function and t is the number of 2's in the k component of the profile function.

EXAMPLES:

```
sage: SteenrodAlgebra(p=7).dimension()
+Infinity
sage: SteenrodAlgebra(profile=[3,2,1]).dimension()
64
sage: SteenrodAlgebra(p=3, profile=([1,1], [])).dimension()
9
sage: SteenrodAlgebra(p=5, profile=([1], [2,2])).dimension()
20
```

`SteenrodAlgebra_generic.gen(i=0)`

The i th generator of this algebra.

INPUT:

- i - non-negative integer

OUTPUT: the i th generator of this algebra

For the full Steenrod algebra, the i^{th} generator is $Sq(2^i)$ at the prime 2; when p is odd, the 0th generator is $\beta = Q(0)$, and for $i > 0$, the i^{th} generator is $P(p^{i-1})$.

For sub-Hopf algebras of the Steenrod algebra, it is not always clear what a minimal generating set is. The sub-Hopf algebra $A(n)$ is minimally generated by the elements Sq^{2^i} for $0 \leq i \leq n$ at the prime 2. At odd primes, $A(n)$ is minimally generated by Q_0 along with \mathcal{P}^{p^i} for $0 \leq i \leq n-1$. So if this algebra is $A(n)$, return the appropriate generator.

For other sub-Hopf algebras: they are generated (but not necessarily minimally) by the P_t^s 's (and Q_n 's, if p is odd) that they contain. So order the P_t^s 's (and Q_n 's) in the algebra by degree and return the i -th one.

EXAMPLES:

```
sage: A = SteenrodAlgebra(2)
sage: A.gen(4)
Sq(16)
sage: A.gen(200)
```

```

Sq(1606938044258990275541962092341162602522202993782792835301376)
sage: SteenrodAlgebra(2, basis='adem').gen(2)
Sq^4
sage: SteenrodAlgebra(2, basis='pst').gen(2)
P^2_1
sage: B = SteenrodAlgebra(5)
sage: B.gen(0)
Q_0
sage: B.gen(2)
P(5)

sage: SteenrodAlgebra(profile=[2,1]).gen(1)
Sq(2)
sage: SteenrodAlgebra(profile=[1,2,1]).gen(1)
Sq(0,1)
sage: SteenrodAlgebra(profile=[1,2,1]).gen(5)
Traceback (most recent call last):
...
ValueError: This algebra only has 4 generators, so call gen(i) with 0 <= i < 4

sage: D = SteenrodAlgebra(profile=lambda n: n)
sage: [D.gen(n) for n in range(5)]
[Sq(1), Sq(0,1), Sq(0,2), Sq(0,0,1), Sq(0,0,2)]
sage: D3 = SteenrodAlgebra(p=3, profile=(lambda n: n, lambda n: 2))
sage: [D3.gen(n) for n in range(9)]
[Q_0, P(1), Q_1, P(0,1), Q_2, P(0,3), P(0,0,1), Q_3, P(0,0,3)]
sage: D3 = SteenrodAlgebra(p=3, profile=(lambda n: n, lambda n: 1 if n<1 else_
↪2))
sage: [D3.gen(n) for n in range(9)]
[P(1), Q_1, P(0,1), Q_2, P(0,3), P(0,0,1), Q_3, P(0,0,3), P(0,0,0,1)]
sage: SteenrodAlgebra(p=5, profile=[[2,1], [2,2,2]], basis='pst').gen(2)
P^1_1

```

`SteenrodAlgebra_generic.gens()`

Family of generators for this algebra.

OUTPUT: family of elements of this algebra

At the prime 2, the Steenrod algebra is generated by the elements Sq^{2^i} for $i \geq 0$. At odd primes, it is generated by the elements Q_0 and \mathcal{P}^{p^i} for $i \geq 0$. So if this algebra is the entire Steenrod algebra, return an infinite family made up of these elements.

For sub-Hopf algebras of the Steenrod algebra, it is not always clear what a minimal generating set is. The sub-Hopf algebra $A(n)$ is minimally generated by the elements Sq^{2^i} for $0 \leq i \leq n$ at the prime 2. At odd primes, $A(n)$ is minimally generated by Q_0 along with \mathcal{P}^{p^i} for $0 \leq i \leq n-1$. So if this algebra is $A(n)$, return the appropriate list of generators.

For other sub-Hopf algebras: return a non-minimal generating set: the family of P_t^s 's and Q_n 's contained in the algebra.

EXAMPLES:

```

sage: A3 = SteenrodAlgebra(3, 'adem')
sage: A3.gens()
Lazy family (<bound method SteenrodAlgebra_generic_with_category.gen of mod 3_
↪Steenrod algebra, serre-cartan basis>(i))_{i in Non negative integers}
sage: A3.gens()[0]
beta
sage: A3.gens()[1]

```

```
P^1
sage: A3.gens() [2]
P^3
sage: SteenrodAlgebra(profile=[3,2,1]).gens()
Family (Sq(1), Sq(2), Sq(4))
```

In the following case, return a non-minimal generating set. (It is not minimal because $Sq(0,0,1)$ is the commutator of $Sq(1)$ and $Sq(0,2)$.)

```
sage: SteenrodAlgebra(profile=[1,2,1]).gens()
Family (Sq(1), Sq(0,1), Sq(0,2), Sq(0,0,1))
sage: SteenrodAlgebra(p=5, profile=[[2,1], [2,2,2]]).gens()
Family (Q_0, P(1), P(5))
sage: SteenrodAlgebra(profile=lambda n: n).gens()
Lazy family (<bound method SteenrodAlgebra_mod_two_with_category.gen of sub-
Hopf algebra of mod 2 Steenrod algebra, milnor basis, profile function [1,
2, 3, ..., 98, 99, +Infinity, +Infinity, +Infinity, ...]>(i))_{i in Non-
negative integers}
```

You may also use `algebra_generators` instead of `gens` :

```
sage: SteenrodAlgebra(p=5, profile=[[2,1], [2,2,2]]).algebra_generators()
Family (Q_0, P(1), P(5))
```

`SteenrodAlgebra_generic.homogeneous_component (n)`

Return the n th homogeneous piece of the Steenrod algebra.

INPUT:

- n - integer

OUTPUT: a vector space spanned by the basis for this algebra in dimension n

EXAMPLES:

```
sage: A = SteenrodAlgebra()
sage: A.homogeneous_component(4)
Vector space spanned by (Sq(1,1), Sq(4)) over Finite Field of size 2
sage: SteenrodAlgebra(profile=[2,1,0]).homogeneous_component(4)
Vector space spanned by (Sq(1,1),) over Finite Field of size 2
```

The notation `A[n]` may also be used:

```
sage: A[5]
Vector space spanned by (Sq(2,1), Sq(5)) over Finite Field of size 2
sage: SteenrodAlgebra(basis='wall')[4]
Vector space spanned by (Q^1_0 Q^0_0, Q^2_2) over Finite Field of size 2
sage: SteenrodAlgebra(p=5)[17]
Vector space spanned by (Q_1 P(1), Q_0 P(2)) over Finite Field of size 5
```

Note that `A[n]` is just a vector space, not a Hopf algebra, so its elements don't have products, coproducts, or antipodes defined on them. If you want to use operations like this on elements of some `A[n]`, then convert them back to elements of `A`:

```
sage: A[5].basis()
Finite family {(5,): milnor[(5,)], (2, 1): milnor[(2, 1)]}
sage: a = list(A[5].basis())[1]
sage: a # not in A, doesn't print like an element of A
milnor[(5,)]
```

```

sage: A(a) # in A
Sq(5)
sage: A(a) * A(a)
Sq(7,1)
sage: a * A(a) # only need to convert one factor
Sq(7,1)
sage: a.antipode() # not defined
Traceback (most recent call last):
...
AttributeError: 'CombinatorialFreeModule_with_category.element_class' object
↳ has no attribute 'antipode'
sage: A(a).antipode() # convert to elt of A, then compute antipode
Sq(2,1) + Sq(5)

sage: G = SteenrodAlgebra(p=5, profile=[[2,1], [2,2,2]], basis='pst')

```

`SteenrodAlgebra_generic.is_commutative()`

True if self is graded commutative, as determined by the profile function. In particular, a sub-Hopf algebra of the mod 2 Steenrod algebra is commutative if and only if there is an integer $n > 0$ so that its profile function e satisfies

- $e(i) = 0$ for $i < n$,
- $e(i) \leq n$ for $i \geq n$.

When p is odd, there must be an integer $n \geq 0$ so that the profile functions e and k satisfy

- $e(i) = 0$ for $i < n$,
- $e(i) \leq n$ for $i \geq n$.
- $k(i) = 1$ for $i < n$.

EXAMPLES:

```

sage: A = SteenrodAlgebra(p=3)
sage: A.is_commutative()
False
sage: SteenrodAlgebra(profile=[2,1]).is_commutative()
False
sage: SteenrodAlgebra(profile=[0,2,2,1]).is_commutative()
True

```

Note that if the profile function is specified by a function, then by default it has infinite truncation type: the profile function is assumed to be infinite after the 100th term.

```

sage: SteenrodAlgebra(profile=lambda n: 1).is_commutative()
False
sage: SteenrodAlgebra(profile=lambda n: 1, truncation_type=0).is_commutative()
True

sage: SteenrodAlgebra(p=5, profile=([0,2,2,1], [])).is_commutative()
True
sage: SteenrodAlgebra(p=5, profile=([0,2,2,1], [1,1,2])).is_commutative()
True
sage: SteenrodAlgebra(p=5, profile=([0,2,1], [1,2,2,2])).is_commutative()
False

```

`SteenrodAlgebra_generic.is_division_algebra()`

The only way this algebra can be a division algebra is if it is the ground field \mathbf{F}_p .

EXAMPLES:

```
sage: SteenrodAlgebra(11).is_division_algebra()
False
sage: SteenrodAlgebra(profile=lambda n: 0, truncation_type=0).is_division_
↪ algebra()
True
```

`SteenrodAlgebra_generic.is_field (proof=True)`

The only way this algebra can be a field is if it is the ground field \mathbf{F}_p .

EXAMPLES:

```
sage: SteenrodAlgebra(11).is_field()
False
sage: SteenrodAlgebra(profile=lambda n: 0, truncation_type=0).is_field()
True
```

`SteenrodAlgebra_generic.is_finite ()`

True if this algebra is finite-dimensional.

Therefore true if the profile function is finite, and in particular the `truncation_type` must be finite.

EXAMPLES:

```
sage: A = SteenrodAlgebra(p=3)
sage: A.is_finite()
False
sage: SteenrodAlgebra(profile=[3,2,1]).is_finite()
True
sage: SteenrodAlgebra(profile=lambda n: n).is_finite()
False
```

`SteenrodAlgebra_generic.is_generic ()`

The algebra is generic if it is based on the odd-primary relations, i.e. if its dual is a quotient of

$$A_* = \mathbf{F}_p[\xi_1, \xi_2, \xi_3, \dots] \otimes \Lambda(\tau_0, \tau_1, \dots)$$

Sage also allows this for $p = 2$. Only the usual Steenrod algebra at the prime 2 and its sub algebras are non-generic.

EXAMPLES:

```
sage: SteenrodAlgebra(3).is_generic()
True
sage: SteenrodAlgebra(2).is_generic()
False
sage: SteenrodAlgebra(2, generic=True).is_generic()
True
```

`SteenrodAlgebra_generic.is_integral_domain (proof=True)`

The only way this algebra can be an integral domain is if it is the ground field \mathbf{F}_p .

EXAMPLES:

```
sage: SteenrodAlgebra(11).is_integral_domain()
False
sage: SteenrodAlgebra(profile=lambda n: 0, truncation_type=0).is_integral_
↪ domain()
True
```

SteenrodAlgebra_generic.**is_noetherian** ()

This algebra is noetherian if and only if it is finite.

EXAMPLES:

```
sage: SteenrodAlgebra(3).is_noetherian()
False
sage: SteenrodAlgebra(profile=[1,2,1]).is_noetherian()
True
sage: SteenrodAlgebra(profile=lambda n: n+2).is_noetherian()
False
```

SteenrodAlgebra_generic.**milnor** ()

Convert an element of this algebra to the Milnor basis

INPUT:

- x - an element of this algebra

OUTPUT: x converted to the Milnor basis

ALGORITHM: use the method `_milnor_on_basis` and linearity.

EXAMPLES:

```
sage: Adem = SteenrodAlgebra(basis='adem')
sage: a = Adem.Sq(2) * Adem.Sq(1)
sage: Adem.milnor(a)
Sq(0,1) + Sq(3)
```

SteenrodAlgebra_generic.**ngens** ()

Number of generators of self.

OUTPUT: number or Infinity

The Steenrod algebra is infinitely generated. A sub-Hopf algebra may be finitely or infinitely generated; in general, it is not clear what a minimal generating set is, nor the cardinality of that set. So: if the algebra is infinite-dimensional, this returns Infinity. If the algebra is finite-dimensional and is equal to one of the sub-Hopf algebras $A(n)$, then their minimal generating set is known, and this returns the cardinality of that set. Otherwise, any sub-Hopf algebra is (not necessarily minimally) generated by the P_t^s 's that it contains (along with the Q_n 's it contains, at odd primes), so this returns the number of P_t^s 's and Q_n 's in the algebra.

EXAMPLES:

```
sage: A = SteenrodAlgebra(3)
sage: A.ngens()
+Infinity
sage: SteenrodAlgebra(profile=lambda n: n).ngens()
+Infinity
sage: SteenrodAlgebra(profile=[3,2,1]).ngens() # A(2)
3
sage: SteenrodAlgebra(profile=[3,2,1], basis='pst').ngens()
3
sage: SteenrodAlgebra(p=3, profile=[[3,2,1], [2,2,2,2]]).ngens() # A(3) at p=3
4
sage: SteenrodAlgebra(profile=[1,2,1,1]).ngens()
5
```

SteenrodAlgebra_generic.**one_basis** ()

The index of the element 1 in the basis for the Steenrod algebra.

EXAMPLES:

```
sage: SteenrodAlgebra(p=2).one_basis()
()
sage: SteenrodAlgebra(p=7).one_basis()
((), ())
```

`SteenrodAlgebra_generic.order()`

The order of this algebra.

This is computed by computing its vector space dimension d and then returning p^d .

EXAMPLES:

```
sage: SteenrodAlgebra(p=7).order()
+Infinity
sage: SteenrodAlgebra(profile=[2,1]).dimension()
8
sage: SteenrodAlgebra(profile=[2,1]).order()
256
sage: SteenrodAlgebra(p=3, profile=[[1], []]).dimension()
3
sage: SteenrodAlgebra(p=3, profile=[[1], []]).order()
27
sage: SteenrodAlgebra(p=5, profile=[[], [2, 2]]).dimension()
4
sage: SteenrodAlgebra(p=5, profile=[[], [2, 2]]).order() == 5**4
True
```

`SteenrodAlgebra_generic.prime()`

The prime associated to self.

EXAMPLES:

```
sage: SteenrodAlgebra(p=2, profile=[1,1]).prime()
2
sage: SteenrodAlgebra(p=7).prime()
7
```

`SteenrodAlgebra_generic.product_on_basis(t1, t2)`

The product of two basis elements of this algebra

INPUT:

• t_1, t_2 – tuples, the indices of two basis elements of self

OUTPUT: the product of the two corresponding basis elements, as an element of self

ALGORITHM: If the two elements are represented in the Milnor basis, use Milnor multiplication as implemented in `sage.algebras.steenrod.steenrod_algebra_mult`. If the two elements are represented in the Serre-Cartan basis, then multiply them using Adem relations (also implemented in `sage.algebras.steenrod.steenrod_algebra_mult`). This provides a good way of checking work – multiply Milnor elements, then convert them to Adem elements and multiply those, and see if the answers correspond.

If the two elements are represented in some other basis, then convert them both to the Milnor basis and multiply.

EXAMPLES:

```

sage: Milnor = SteenrodAlgebra()
sage: Milnor.product_on_basis((2,), (2,))
Sq(1,1)
sage: Adem = SteenrodAlgebra(basis='adem')
sage: Adem.Sq(2) * Adem.Sq(2) # indirect doctest
Sq^3 Sq^1

```

When multiplying elements from different bases, the left-hand factor determines the form of the output:

```

sage: Adem.Sq(2) * Milnor.Sq(2)
Sq^3 Sq^1
sage: Milnor.Sq(2) * Adem.Sq(2)
Sq(1,1)

```

`SteenrodAlgebra_generic.profile (i, component=0)`

Profile function for this algebra.

INPUT:

- i - integer
- `component` - either 0 or 1, optional (default 0)

OUTPUT: integer or ∞

See the documentation for `sage.algebras.steenrod.steenrod_algebra` and `SteenrodAlgebra()` for information on profile functions.

This applies the profile function to the integer i . Thus when $p = 2$, i must be a positive integer. When p is odd, there are two profile functions, e and k (in the notation of the aforementioned documentation), corresponding, respectively to `component=0` and `component=1`. So when p is odd and `component` is 0, i must be positive, while when `component` is 1, i must be non-negative.

EXAMPLES:

```

sage: SteenrodAlgebra().profile(3)
+Infinity
sage: SteenrodAlgebra(profile=[3,2,1]).profile(1)
3
sage: SteenrodAlgebra(profile=[3,2,1]).profile(2)
2

```

When the profile is specified by a list, the default behavior is to return zero values outside the range of the list. This can be overridden if the algebra is created with an infinite `truncation_type`:

```

sage: SteenrodAlgebra(profile=[3,2,1]).profile(9)
0
sage: SteenrodAlgebra(profile=[3,2,1], truncation_type=Infinity).profile(9)
+Infinity

sage: B = SteenrodAlgebra(p=3, profile=(lambda n: n, lambda n: 1))
sage: B.profile(3)
3
sage: B.profile(3, component=1)
1

sage: EA = SteenrodAlgebra(generic=True, profile=(lambda n: n, lambda n: 1))
sage: EA.profile(4)
4

```



```
sage: EA.profile(2, component=1)
1
```

`SteenrodAlgebra_generic.pst (s, t)`

The Margolis element P_t^s .

INPUT:

- `s` - non-negative integer
- `t` - positive integer
- `p` - positive prime number

OUTPUT: element of the Steenrod algebra

This returns the Margolis element P_t^s of the mod p Steenrod algebra: the element equal to $P(0, 0, \dots, 0, p^s)$, where the p^s is in position t .

EXAMPLES:

```
sage: A2 = SteenrodAlgebra(2)
sage: A2.pst(3, 5)
Sq(0, 0, 0, 0, 8)
sage: A2.pst(1, 2) == Sq(4)*Sq(2) + Sq(2)*Sq(4)
True
sage: SteenrodAlgebra(5).pst(3, 5)
P(0, 0, 0, 0, 125)
```

`SteenrodAlgebra_generic.top_class ()`

Highest dimensional basis element. This is only defined if the algebra is finite.

EXAMPLES:

```
sage: SteenrodAlgebra(2, profile=(3, 2, 1)).top_class()
Sq(7, 3, 1)
sage: SteenrodAlgebra(3, profile=((2, 2, 1), (1, 2, 2, 2, 2))).top_class()
Q_1 Q_2 Q_3 Q_4 P(8, 8, 2)
```

```
class sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_mod_two ( p=2,
                                                                    ba-
                                                                    sis='milnor',
                                                                    **kws)
```

Bases: `sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic`

The mod 2 Steenrod algebra.

Users should not call this, but use the function `SteenrodAlgebra()` instead. See that function for extensive documentation. (This differs from `SteenrodAlgebra_generic` only in that it has a method `Sq()` for defining elements.)

Sq (**nums*)

Milnor element $Sq(a, b, c, \dots)$.

INPUT:

- `a, b, c, \dots` - non-negative integers

OUTPUT: element of the Steenrod algebra

This returns the Milnor basis element $Sq(a, b, c, \dots)$.

EXAMPLES:

```
sage: A = SteenrodAlgebra(2)
sage: A.Sq(5)
Sq(5)
sage: A.Sq(5,0,2)
Sq(5,0,2)
```

Entries must be non-negative integers; otherwise, an error results.

4.18 Steenrod algebra bases

AUTHORS:

- John H. Palmieri (2008-07-30): version 0.9
- John H. Palmieri (2010-06-30): version 1.0
- Simon King (2011-10-25): Fix the use of cached functions

This package defines functions for computing various bases of the Steenrod algebra, and for converting between the Milnor basis and any other basis.

This packages implements a number of different bases, at least at the prime 2. The Milnor and Serre-Cartan bases are the most familiar and most standard ones, and all of the others are defined in terms of one of these. The bases are described in the documentation for the function `steenrod_algebra_basis()`; also see the papers by Monks [Mon1998] and Wood [Woo1998] for more information about them. For commutator bases, see the preprint by Palmieri and Zhang [PZ2008].

- ‘milnor’: Milnor basis.
- ‘serre-cartan’ or ‘adem’ or ‘admissible’: Serre-Cartan basis.

Most of the rest of the bases are only defined when $p = 2$. The only exceptions are the P_t^s -bases and the commutator bases, which are defined at all primes.

- ‘wood_y’: Wood’s Y basis.
- ‘wood_z’: Wood’s Z basis.
- ‘wall’, ‘wall_long’: Wall’s basis.
- ‘arnon_a’, ‘arnon_a_long’: Arnon’s A basis.
- ‘arnon_c’: Arnon’s C basis.
- ‘pst’, ‘pst_rlex’, ‘pst_llex’, ‘pst_deg’, ‘pst_revz’: various P_t^s -bases.
- ‘comm’, ‘comm_rlex’, ‘comm_llex’, ‘comm_deg’, ‘comm_revz’, or these with ‘_long’ appended: various commutator bases.

The main functions provided here are

- `steenrod_algebra_basis()` . This computes a tuple representing basis elements for the Steenrod algebra in a given degree, at a given prime, with respect to a given basis. It is a cached function.
- `convert_to_milnor_matrix()` . This returns the change-of-basis matrix, in a given degree, from any basis to the Milnor basis. It is a cached function.
- `convert_from_milnor_matrix()` . This returns the inverse of the previous matrix.

INTERNAL DOCUMENTATION:

If you want to implement a new basis for the Steenrod algebra:

In the file `steenrod_algebra.py`:

For the class `SteenrodAlgebra_generic`, add functionality to the methods:

- `_repr_term`
- `degree_on_basis`
- `_milnor_on_basis`
- `an_element`

In the file `steenrod_algebra_misc.py`:

- add functionality to `get_basis_name`: this should accept as input various synonyms for the basis, and its output should be a canonical name for the basis.
- add a function `BASIS_mono_to_string` like `milnor_mono_to_string` or one of the other similar functions.

In this file `steenrod_algebra_bases.py`:

- add appropriate lines to `steenrod_algebra_basis()`.
- add a function to compute the basis in a given dimension (to be called by `steenrod_algebra_basis()`).
- modify `steenrod_basis_error_check()` so it checks the new basis.

If the basis has an intrinsic way of defining a product, implement it in the file `steenrod_algebra_mult.py` and also in the `product_on_basis` method for `SteenrodAlgebra_generic` in `steenrod_algebra.py`.

`sage.algebras.steenrod.steenrod_algebra_bases.arnonC_basis (n, bound=1)`

Arnon's C basis in dimension n .

INPUT:

- n - non-negative integer
- $bound$ - positive integer (optional)

OUTPUT: tuple of basis elements in dimension n

The elements of Arnon's C basis are monomials of the form $Sq^{t_1} \dots Sq^{t_m}$ where for each i , we have $t_i \leq 2t_{i+1}$ and $2^i | t_{m-i}$.

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_bases import arnonC_basis
sage: arnonC_basis(7)
((7,), (2, 5), (4, 3), (4, 2, 1))
```

If optional argument `bound` is present, include only those monomials whose first term is at least as large as `bound`:

```
sage: arnonC_basis(7, 3)
((7,), (4, 3), (4, 2, 1))
```

`sage.algebras.steenrod.steenrod_algebra_bases.atomic_basis (n, basis, **kws)`

Basis for dimension n made of elements in 'atomic' degrees: degrees of the form $2^i(2^j - 1)$.

This works at the prime 2 only.

INPUT:

- n - non-negative integer
- `basis` - string, the name of the basis

- `profile` - profile function (optional, default None). Together with `truncation_type`, specify the profile function to be used; None means the profile function for the entire Steenrod algebra. See `sage.algebras.steenrod.steenrod_algebra` and `SteenrodAlgebra()` for information on profile functions.
- `truncation_type` - truncation type, either 0 or Infinity (optional, default Infinity if no profile function is specified, 0 otherwise).

OUTPUT: tuple of basis elements in dimension n

The atomic bases include Wood's Y and Z bases, Wall's basis, Arnon's A basis, the P_t^s -bases, and the commutator bases. (All of these bases are constructed similarly, hence their constructions have been consolidated into a single function. Also, see the documentation for 'steenrod_algebra_basis' for descriptions of them.) For P_t^s -bases, you may also specify a profile function and truncation type; profile functions are ignored for the other bases.

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_bases import atomic_basis
sage: atomic_basis(6, 'woody')
((1, 0), (0, 1), (0, 0)), ((2, 0), (1, 0)), ((1, 1),))
sage: atomic_basis(8, 'woodz')
((2, 0), (0, 1), (0, 0)), ((0, 2), (0, 0)), ((1, 1), (1, 0)), ((3, 0),))
sage: atomic_basis(6, 'woodz') == atomic_basis(6, 'woody')
True
sage: atomic_basis(9, 'woodz') == atomic_basis(9, 'woody')
False
```

Wall's basis:

```
sage: atomic_basis(8, 'wall')
((2, 2), (1, 0), (0, 0)), ((2, 0), (0, 0)), ((2, 1), (1, 1)), ((3, 3),))
```

Arnon's A basis:

```
sage: atomic_basis(7, 'arnona')
(((0, 0), (1, 1), (2, 2)), ((0, 0), (2, 1)), ((1, 0), (2, 2)), ((2, 0),))
```

P_t^s -bases:

```
sage: atomic_basis(7, 'pst_rlex')
(((0, 1), (1, 1), (2, 1)), ((0, 1), (1, 2)), ((2, 1), (0, 2)), ((0, 3),))
sage: atomic_basis(7, 'pst_llex')
(((0, 1), (1, 1), (2, 1)), ((0, 1), (1, 2)), ((0, 2), (2, 1)), ((0, 3),))
sage: atomic_basis(7, 'pst_deg')
(((0, 1), (1, 1), (2, 1)), ((0, 1), (1, 2)), ((0, 2), (2, 1)), ((0, 3),))
sage: atomic_basis(7, 'pst_revz')
(((0, 1), (1, 1), (2, 1)), ((0, 1), (1, 2)), ((0, 2), (2, 1)), ((0, 3),))
```

Commutator bases:

```
sage: atomic_basis(7, 'comm_rlex')
(((0, 1), (1, 1), (2, 1)), ((0, 1), (1, 2)), ((2, 1), (0, 2)), ((0, 3),))
sage: atomic_basis(7, 'comm_llex')
(((0, 1), (1, 1), (2, 1)), ((0, 1), (1, 2)), ((0, 2), (2, 1)), ((0, 3),))
sage: atomic_basis(7, 'comm_deg')
(((0, 1), (1, 1), (2, 1)), ((0, 1), (1, 2)), ((0, 2), (2, 1)), ((0, 3),))
sage: atomic_basis(7, 'comm_revz')
(((0, 1), (1, 1), (2, 1)), ((0, 1), (1, 2)), ((0, 2), (2, 1)), ((0, 3),))
```

`sage.algebras.steenrod.steenrod_algebra_bases.atomic_basis_odd` (*n*, *basis*, *p*,
***kws*)

P_t^s -bases and commutator basis in dimension n at odd primes.

This function is called `atomic_basis_odd` in analogy with `atomic_basis()`.

INPUT:

- *n* - non-negative integer
- *basis* - string, the name of the basis
- *p* - positive prime number
- *profile* - profile function (optional, default None). Together with `truncation_type`, specify the profile function to be used; None means the profile function for the entire Steenrod algebra. See [sage.algebras.steenrod.steenrod_algebra](#) and `SteenrodAlgebra()` for information on profile functions.
- *truncation_type* - truncation type, either 0 or Infinity (optional, default Infinity if no profile function is specified, 0 otherwise).

OUTPUT: tuple of basis elements in dimension n

The only possible difference in the implementations for P_t^s bases and commutator bases is that the former make sense, and require filtering, if there is a nontrivial profile function. This function is called by `steenrod_algebra_basis()`, and it will not be called for commutator bases if there is a profile function, so we treat the two bases exactly the same.

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_bases import atomic_basis_odd
sage: atomic_basis_odd(8, 'pst_rlex', 3)
(((), ((0, 1), 2)),)

sage: atomic_basis_odd(18, 'pst_rlex', 3)
(((0, 2), ()), ((0, 1), ((1, 1), 1)))
sage: atomic_basis_odd(18, 'pst_rlex', 3, profile=(((), (2,2,2)))
(((0, 2), ()),)
```

`sage.algebras.steenrod.steenrod_algebra_bases.convert_from_milnor_matrix` (*n*,
ba-
sis,
p=2,
generic='auto')

Change-of-basis matrix, Milnor to 'basis', in dimension n .

INPUT:

- *n* - non-negative integer, the dimension
- *basis* - string, the basis to which to convert
- *p* - positive prime number (optional, default 2)

OUTPUT: `matrix` - change-of-basis matrix, a square matrix over $\text{GF}(p)$

Note: This is called internally. It is not intended for casual users, so no error checking is made on the integer n , the basis name, or the prime.

EXAMPLES:

```

sage: from sage.algebras.steenrod.steenrod_algebra_bases import convert_from_
      ↪milnor_matrix, convert_to_milnor_matrix
sage: convert_from_milnor_matrix(12, 'wall')
[1 0 0 1 0 0 0]
[0 0 1 1 0 0 0]
[0 0 0 1 0 1 1]
[0 0 0 1 0 0 0]
[1 0 1 0 1 0 0]
[1 1 1 0 0 0 0]
[1 0 1 0 1 0 1]
sage: convert_from_milnor_matrix(38, 'serre_cartan')
72 x 72 dense matrix over Finite Field of size 2 (use the '.str()' method to see
      ↪the entries)
sage: x = convert_to_milnor_matrix(20, 'wood_y')
sage: y = convert_from_milnor_matrix(20, 'wood_y')
sage: x*y
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]

```

The function takes an optional argument, the prime p over which to work:

```

sage: convert_from_milnor_matrix(17, 'adem', 3)
[2 1 1 2]
[0 2 0 1]
[1 2 0 0]
[0 1 0 0]

```

```

sage.algebras.steenrod.steenrod_algebra_bases. convert_to_milnor_matrix (  $n$ ,
                                                                 ba-
                                                                 sis,
                                                                  $p=2$ ,
                                                                 generic='auto')

```

Change-of-basis matrix, ‘basis’ to Milnor, in dimension n , at the prime p .

INPUT:

- n - non-negative integer, the dimension
- *basis* - string, the basis from which to convert
- p - positive prime number (optional, default 2)

OUTPUT:

matrix - change-of-basis matrix, a square matrix over $\text{GF}(p)$

EXAMPLES:

```

sage: from sage.algebras.steenrod.steenrod_algebra_bases import convert_to_milnor_
      ↪matrix
sage: convert_to_milnor_matrix(5, 'adem') # indirect doctest
[0 1]
[1 1]
sage: convert_to_milnor_matrix(45, 'milnor')
111 x 111 dense matrix over Finite Field of size 2 (use the '.str()' method to
      ↪see the entries)
sage: convert_to_milnor_matrix(12, 'wall')
[1 0 0 1 0 0 0]
[1 1 0 0 0 1 0]
[0 1 0 1 0 0 0]
[0 0 0 1 0 0 0]
[1 1 0 0 1 0 0]
[0 0 1 1 1 0 1]
[0 0 0 0 1 0 1]

```

The function takes an optional argument, the prime p over which to work:

```

sage: convert_to_milnor_matrix(17, 'adem', 3)
[0 0 1 1]
[0 0 0 1]
[1 1 1 1]
[0 1 0 1]
sage: convert_to_milnor_matrix(48, 'adem', 5)
[0 1]
[1 1]
sage: convert_to_milnor_matrix(36, 'adem', 3)
[0 0 1]
[0 1 0]
[1 2 0]

```

`sage.algebras.steenrod.steenrod_algebra_bases.milnor_basis (n, p=2, **kws)`
 Milnor basis in dimension n with profile function `profile`.

INPUT:

- `n` - non-negative integer
- `p` - positive prime number (optional, default 2)
- `profile` - profile function (optional, default None). Together with `truncation_type`, specify the profile function to be used; None means the profile function for the entire Steenrod algebra. See [sage.algebras.steenrod.steenrod_algebra](#) and [SteenrodAlgebra](#) for information on profile functions.
- `truncation_type` - truncation type, either 0 or Infinity (optional, default Infinity if no profile function is specified, 0 otherwise)

OUTPUT: tuple of mod p Milnor basis elements in dimension n

At the prime 2, the Milnor basis consists of symbols of the form $Sq(m_1, m_2, \dots, m_t)$, where each m_i is a non-negative integer and if $t > 1$, then $m_t \neq 0$. At odd primes, it consists of symbols of the form $Q_{e_1} Q_{e_2} \dots P(m_1, m_2, \dots, m_t)$, where $0 \leq e_1 < e_2 < \dots$, each m_i is a non-negative integer, and if $t > 1$, then $m_t \neq 0$.

EXAMPLES:

```

sage: from sage.algebras.steenrod.steenrod_algebra_bases import milnor_basis
sage: milnor_basis(7)
((0, 0, 1), (1, 2), (4, 1), (7,))
sage: milnor_basis(7, 2)
((0, 0, 1), (1, 2), (4, 1), (7,))
sage: milnor_basis(4, 2)
((1, 1), (4,))
sage: milnor_basis(4, 2, profile=[2,1])
((1, 1),)
sage: milnor_basis(4, 2, profile=(), truncation_type=0)
()
sage: milnor_basis(4, 2, profile=(), truncation_type=Infinity)
((1, 1), (4,))
sage: milnor_basis(9, 3)
(((1,), (1,)), ((0,), (2,)))
sage: milnor_basis(17, 3)
(((2,), ()), ((1,), (3,)), ((0,), (0, 1)), ((0,), (4,)))
sage: milnor_basis(48, p=5)
(((, (0, 1)), ((, (6,)))
sage: len(milnor_basis(100,3))
13
sage: len(milnor_basis(200,7))
0
sage: len(milnor_basis(240,7))
3
sage: len(milnor_basis(240,7, profile=(), truncation_type=Infinity))
3
sage: len(milnor_basis(240,7, profile=(), truncation_type=0))
0

```

sage.algebras.steenrod.steenrod_algebra_bases. **restricted_partitions** (*n*, *l*,
no_repeats=False)

List of ‘restricted’ partitions of *n*: partitions with parts taken from list.

INPUT:

- *n* - non-negative integer
- *l* - list of positive integers
- *no_repeats* - boolean (optional, default = False), if True, only return partitions with no repeated parts

OUTPUT: list of lists

One could also use `Partitions(n, parts_in=l)`, but this function may be faster. Also, while `Partitions(n, parts_in=l, max_slope=-1)` should in theory return the partitions of *n* with parts in *l* with no repetitions, the `max_slope=-1` argument is ignored, so it doesn’t work. (At the moment, the `no_repeats=True` case is the only one used in the code.)

EXAMPLES:

```

sage: from sage.algebras.steenrod.steenrod_algebra_bases import restricted_
      ↪ partitions
sage: restricted_partitions(10, [7,5,1])
[[7, 1, 1, 1], [5, 5], [5, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1]]
sage: restricted_partitions(10, [6,5,4,3,2,1], no_repeats=True)
[[6, 4], [6, 3, 1], [5, 4, 1], [5, 3, 2], [4, 3, 2, 1]]
sage: restricted_partitions(10, [6,4,2])
[[6, 4], [6, 2, 2], [4, 4, 2], [4, 2, 2, 2], [2, 2, 2, 2, 2]]

```



```
sage: restricted_partitions(10, [6,4,2], no_repeats=True)
[[6, 4]]
```

'l' may have repeated elements. If 'no_repeats' is False, this has no effect. If 'no_repeats' is True, and if the repeated elements appear consecutively in 'l', then each element may be used only as many times as it appears in 'l':

```
sage: restricted_partitions(10, [6,4,2,2], no_repeats=True)
[[6, 4], [6, 2, 2]]
sage: restricted_partitions(10, [6,4,2,2,2], no_repeats=True)
[[6, 4], [6, 2, 2], [4, 2, 2, 2]]
```

(If the repeated elements don't appear consecutively, the results are likely meaningless, containing several partitions more than once, for example.)

In the following examples, 'no_repeats' is False:

```
sage: restricted_partitions(10, [6,4,2])
[[6, 4], [6, 2, 2], [4, 4, 2], [4, 2, 2, 2], [2, 2, 2, 2, 2]]
sage: restricted_partitions(10, [6,4,2,2,2])
[[6, 4], [6, 2, 2], [4, 4, 2], [4, 2, 2, 2], [2, 2, 2, 2, 2]]
sage: restricted_partitions(10, [6,4,4,4,2,2,2,2,2,2])
[[6, 4], [6, 2, 2], [4, 4, 2], [4, 2, 2, 2], [2, 2, 2, 2, 2]]
```

```
sage.algebras.steenrod.steenrod_algebra_bases. serre_cartan_basis ( n,      p=2,
                                                                    bound=1,
                                                                    **kwds)
```

Serre-Cartan basis in dimension n .

INPUT:

- n - non-negative integer
- $bound$ - positive integer (optional)
- $prime$ - positive prime number (optional, default 2)

OUTPUT: tuple of mod p Serre-Cartan basis elements in dimension n

The Serre-Cartan basis consists of 'admissible monomials in the Steenrod squares'. Thus at the prime 2, it consists of monomials $Sq^{m_1} Sq^{m_2} \dots Sq^{m_t}$ with $m_i \geq 2m_{i+1}$ for each i . At odd primes, it consists of monomials $\beta^{e_0} P^{s_1} \beta^{e_1} P^{s_2} \dots P^{s_k} \beta^{e_k}$ with each e_i either 0 or 1, $s_i \geq ps_{i+1} + e_i$ for all i , and $s_k \geq 1$.

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_bases import serre_cartan_basis
sage: serre_cartan_basis(7)
((7,), (6, 1), (4, 2, 1), (5, 2))
sage: serre_cartan_basis(13,3)
((1, 3, 0), (0, 3, 1))
sage: serre_cartan_basis(50,5)
((1, 5, 0, 1, 1), (1, 6, 1))
```

If optional argument $bound$ is present, include only those monomials whose last term is at least $bound$ (when $p=2$), or those for which $s_k - e_k \geq bound$ (when p is odd).

```
sage: serre_cartan_basis(7, bound=2)
((7,), (5, 2))
sage: serre_cartan_basis(13, 3, bound=3)
((1, 3, 0),)
```

```
sage.algebras.steenrod.steenrod_algebra_bases. steenrod_algebra_basis ( n,
                                                                    ba-
                                                                    sis='milnor',
                                                                    p=2,
                                                                    **kwds)
```

Basis for the Steenrod algebra in degree n .

INPUT:

- `n` - non-negative integer
- `basis` - string, which basis to use (optional, default = 'milnor')
- `p` - positive prime number (optional, default = 2)
- `profile` - profile function (optional, default None). This is just passed on to the functions `milnor_basis()` and `pst_basis()`.
- `truncation_type` - truncation type, either 0 or Infinity (optional, default Infinity if no profile function is specified, 0 otherwise). This is just passed on to the function `milnor_basis()`.
- `generic` - boolean (optional, default = None)

OUTPUT:

Tuple of objects representing basis elements for the Steenrod algebra in dimension n .

The choices for the string `basis` are as follows; see the documentation for `sage.algebras.steenrod.steenrod_algebra` for details on each basis:

- 'milnor': Milnor basis.
- 'serre-cartan' or 'adem' or 'admissible': Serre-Cartan basis.
- 'pst', 'pst_rlex', 'pst_llex', 'pst_deg', 'pst_revz': various P_t^s -bases.
- 'comm', 'comm_rlex', 'comm_llex', 'comm_deg', 'comm_revz', or any of these with '_long' appended: various commutator bases.

The rest of these bases are only defined when $p = 2$.

- 'wood_y': Wood's Y basis.
- 'wood_z': Wood's Z basis.
- 'wall' or 'wall_long': Wall's basis.
- 'arnon_a' or 'arnon_a_long': Arnon's A basis.
- 'arnon_c': Arnon's C basis.

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_bases import steenrod_algebra_
      ↪basis
sage: steenrod_algebra_basis(7, 'milnor') # indirect doctest
((0, 0, 1), (1, 2), (4, 1), (7,))
sage: steenrod_algebra_basis(5) # milnor basis is the default
((2, 1), (5,))
```

Bases in negative dimensions are empty:

```
sage: steenrod_algebra_basis(-2, 'wall')
()
```

The third (optional) argument to ‘steenrod_algebra_basis’ is the prime p :

```
sage: steenrod_algebra_basis(9, 'milnor', p=3)
((1,), (1,)), ((0,), (2,)))
sage: steenrod_algebra_basis(9, 'milnor', 3)
((1,), (1,)), ((0,), (2,)))
sage: steenrod_algebra_basis(17, 'milnor', 3)
((2,), ()), ((1,), (3,)), ((0,), (0, 1)), ((0,), (4,)))
```

Other bases:

```
sage: steenrod_algebra_basis(7, 'admissible')
((7,), (6, 1), (4, 2, 1), (5, 2))
sage: steenrod_algebra_basis(13, 'admissible', p=3)
((1, 3, 0), (0, 3, 1))
sage: steenrod_algebra_basis(5, 'wall')
((2, 2), (0, 0)), ((1, 1), (1, 0))
sage: steenrod_algebra_basis(5, 'wall_long')
((2, 2), (0, 0)), ((1, 1), (1, 0))
sage: steenrod_algebra_basis(5, 'pst-rlex')
((0, 1), (2, 1)), ((1, 1), (0, 2)))
```

```
sage.algebras.steenrod.steenrod_algebra_bases. steenrod_basis_error_check ( dim,
                                                                    p,
                                                                    **kwds)
```

This performs crude error checking.

INPUT:

- dim - non-negative integer
- p - positive prime number

OUTPUT: None

This checks to see if the different bases have the same length, and if the change-of-basis matrices are invertible. If something goes wrong, an error message is printed.

This function checks at the prime p as the dimension goes up from 0 to dim .

If you set the Sage verbosity level to a positive integer (using `set_verbosity(n)`), then some extra messages will be printed.

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_bases import steenrod_basis_
      ↪ error_check
sage: steenrod_basis_error_check(15, 2) # long time
sage: steenrod_basis_error_check(15, 2, generic=True) # long time
sage: steenrod_basis_error_check(40, 3) # long time
sage: steenrod_basis_error_check(80, 5) # long time
```

```
sage.algebras.steenrod.steenrod_algebra_bases. xi_degrees ( n, p=2, reverse=True)
Decreasing list of degrees of the  $\xi_i$ 's, starting in degree  $n$ .
```

INPUT:

- n - integer
- p - prime number, optional (default 2)
- *reverse* - bool, optional (default True)

OUTPUT: `list` - list of integers

When $p = 2$: decreasing list of the degrees of the ξ_i 's with degree at most n .

At odd primes: decreasing list of these degrees, each divided by $2(p - 1)$.

If `reverse` is `False`, then return an increasing list rather than a decreasing one.

EXAMPLES:

```
sage: sage.algebras.steenrod.steenrod_algebra_bases.xi_degrees(17)
[15, 7, 3, 1]
sage: sage.algebras.steenrod.steenrod_algebra_bases.xi_degrees(17, reverse=False)
[1, 3, 7, 15]
sage: sage.algebras.steenrod.steenrod_algebra_bases.xi_degrees(17,p=3)
[13, 4, 1]
sage: sage.algebras.steenrod.steenrod_algebra_bases.xi_degrees(400,p=17)
[307, 18, 1]
```

4.19 Miscellaneous functions for the Steenrod algebra and its elements

AUTHORS:

- John H. Palmieri (2008-07-30): initial version (as the file `steenrod_algebra_element.py`)
- John H. Palmieri (2010-06-30): initial version of `steenrod_misc.py`. Implemented profile functions. Moved most of the methods for elements to the `Element` subclass of `sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic`.

The main functions here are

- `get_basis_name()` . This function takes a string as input and attempts to interpret it as the name of a basis for the Steenrod algebra; it returns the canonical name attached to that basis. This allows for the use of synonyms when defining bases, while the resulting algebras will be identical.
- `normalize_profile()` . This function returns the canonical (and hashable) description of any profile function. See `sage.algebras.steenrod.steenrod_algebra` and `SteenrodAlgebra` for information on profile functions.
- functions named `*_mono_to_string` where `*` is a basis name (`milnor_mono_to_string()` , etc.). These convert tuples representing basis elements to strings, for `_repr_` and `_latex_` methods.

```
sage.algebras.steenrod.steenrod_algebra_misc. arnonA_long_mono_to_string ( mono,
                                                                    la-
                                                                    tex=False,
                                                                    p=2)
```

Alternate string representation of element of Arnon's A basis.

This is used by the `_repr_` and `_latex_` methods.

INPUT:

- `mono` - tuple of pairs of non-negative integers (m, k) with $m \geq k$
- `latex` - boolean (optional, default `False`), if true, output LaTeX string

OUTPUT: `string` - concatenation of strings of the form $Sq(2^m)$

EXAMPLES:

```

sage: from sage.algebras.steenrod.steenrod_algebra_misc import arnonA_long_mono_
      ↪to_string
sage: arnonA_long_mono_to_string(((1,2),(3,0)))
'Sq^{8} Sq^{4} Sq^{2} Sq^{1}'
sage: arnonA_long_mono_to_string(((1,2),(3,0)), latex=True)
'\text{Sq}^{8} \text{Sq}^{4} \text{Sq}^{2} \text{Sq}^{1}'

```

The empty tuple represents the unit element:

```

sage: arnonA_long_mono_to_string(())
'1'

```

```

sage.algebras.steenrod.steenrod_algebra_misc. arnonA_mono_to_string ( mono,
                                                                    la-
                                                                    tex=False,
                                                                    p=2)

```

String representation of element of Arnon's A basis.

This is used by the `_repr_` and `_latex_` methods.

INPUT:

- `mono` - tuple of pairs of non-negative integers (m,k) with $m \geq k$
- `latex` - boolean (optional, default False), if true, output LaTeX string

OUTPUT: `string` - concatenation of strings of the form $X^{\{m\}}_{\{k\}}$ for each pair (m,k)

EXAMPLES:

```

sage: from sage.algebras.steenrod.steenrod_algebra_misc import arnonA_mono_to_
      ↪string
sage: arnonA_mono_to_string(((1,2),(3,0)))
'X^{1}_{2} X^{3}_{0}'
sage: arnonA_mono_to_string(((1,2),(3,0)), latex=True)
'X^{1}_{2} X^{3}_{0}'

```

The empty tuple represents the unit element:

```

sage: arnonA_mono_to_string(())
'1'

```

```

sage.algebras.steenrod.steenrod_algebra_misc. comm_long_mono_to_string ( mono,
                                                                    p,
                                                                    la-
                                                                    tex=False,
                                                                    generic=False)

```

Alternate string representation of element of a commutator basis.

Okay in low dimensions, but gets unwieldy as the dimension increases.

INPUT:

- `mono` - tuple of pairs of integers (s,t) with $s \geq 0, t > 0$
- `latex` - boolean (optional, default False), if true, output LaTeX string
- `generic` - whether to format generically, or for the prime 2 (default)

OUTPUT: `string` - concatenation of strings of the form $s_{\{2^s \dots 2^{(s+t-1)}\}}$ for each pair (s,t)

EXAMPLES:

```

sage: from sage.algebras.steenrod.steenrod_algebra_misc import comm_long_mono_to_
      ↪string
sage: comm_long_mono_to_string(((1,2),(0,3)), 2)
's_{24} s_{124}'
sage: comm_long_mono_to_string(((1,2),(0,3)), 2, latex=True)
's_{24} s_{124}'
sage: comm_long_mono_to_string(((1, 4), ((1,2), 1), ((0,3), 2))), 5, generic=True)
'Q_{1} Q_{4} s_{5,25} s_{1,5,25}^2'
sage: comm_long_mono_to_string(((1, 4), ((1,2), 1), ((0,3), 2))), 3, latex=True, ↪
      ↪generic=True)
'Q_{1} Q_{4} s_{3,9} s_{1,3,9}^2'

```

The empty tuple represents the unit element:

```

sage: comm_long_mono_to_string((), p=2)
'1'

```

`sage.algebras.steenrod.steenrod_algebra_misc. comm_mono_to_string (mono, latex=False, generic=False)`

String representation of element of a commutator basis.

This is used by the `_repr_` and `_latex_` methods.

INPUT:

- `mono` - tuple of pairs of integers (s,t) with $s \geq 0, t \geq 0$
- `latex` - boolean (optional, default False), if true, output LaTeX string
- `generic` - whether to format generically, or for the prime 2 (default)

OUTPUT: string - concatenation of strings of the form `c_{s,t}` for each pair (s,t)

EXAMPLES:

```

sage: from sage.algebras.steenrod.steenrod_algebra_misc import comm_mono_to_string
sage: comm_mono_to_string(((1,2),(0,3)), generic=False)
'c_{1,2} c_{0,3}'
sage: comm_mono_to_string(((1,2),(0,3)), latex=True)
'c_{1,2} c_{0,3}'
sage: comm_mono_to_string(((1, 4), ((1,2), 1), ((0,3), 2))), generic=True)
'Q_{1} Q_{4} c_{1,2} c_{0,3}^2'
sage: comm_mono_to_string(((1, 4), ((1,2), 1), ((0,3), 2))), latex=True, ↪
      ↪generic=True)
'Q_{1} Q_{4} c_{1,2} c_{0,3}^2'

```

The empty tuple represents the unit element:

```

sage: comm_mono_to_string(())
'1'

```

`sage.algebras.steenrod.steenrod_algebra_misc. convert_perm (m)`

Convert tuple `m` of non-negative integers to a permutation in one-line form.

INPUT:

- `m` - tuple of non-negative integers with no repetitions

OUTPUT: list - conversion of `m` to a permutation of the set `1,2,...,len(m)`

If $m = (3, 7, 4)$, then one can view m as representing the permutation of the set $(3, 4, 7)$ sending 3 to 3, 4 to 7, and 7 to 4. This function converts m to the list $[1, 3, 2]$, which represents essentially the same permutation, but of the set $(1, 2, 3)$. This list can then be passed to `Permutation`, and its signature can be computed.

EXAMPLES:

```
sage: sage.algebras.steenrod.steenrod_algebra_misc.convert_perm((3,7,4))
[1, 3, 2]
sage: sage.algebras.steenrod.steenrod_algebra_misc.convert_perm((5,0,6,3))
[3, 1, 4, 2]
```

```
sage.algebras.steenrod.steenrod_algebra_misc.get_basis_name (basis, p,
                                                             generic=None)
```

Return canonical basis named by string `basis` at the prime `p`.

INPUT:

- `basis` - string
- `p` - positive prime number
- `generic` - boolean, optional, default to 'None'

OUTPUT:

- `basis_name` - string

Specify the names of the implemented bases. The input is converted to lower-case, then processed to return the canonical name for the basis.

For the Milnor and Serre-Cartan bases, use the list of synonyms defined by the variables `_steenrod_milnor_basis_names` and `_steenrod_serre_cartan_basis_names`. Their canonical names are 'milnor' and 'serre-cartan', respectively.

For the other bases, use pattern-matching rather than a list of synonyms:

- Search for 'wood' and 'y' or 'wood' and 'z' to get the Wood bases. Canonical names 'woody', 'woodz'.
- Search for 'arnon' and 'c' for the Arnon C basis. Canonical name: 'arnonc'.
- Search for 'arnon' (and no 'c') for the Arnon A basis. Also see if 'long' is present, for the long form of the basis. Canonical names: 'arnona', 'arnona_long'.
- Search for 'wall' for the Wall basis. Also see if 'long' is present. Canonical names: 'wall', 'wall_long'.
- Search for 'pst' for P^s_t bases, then search for the order type: 'rlex', 'llex', 'deg', 'revz'. Canonical names: 'pst_rlex', 'pst_llex', 'pst_deg', 'pst_revz'.
- For commutator types, search for 'comm', an order type, and also check to see if 'long' is present. Canonical names: 'comm_rlex', 'comm_llex', 'comm_deg', 'comm_revz', 'comm_rlex_long', 'comm_llex_long', 'comm_deg_long', 'comm_revz_long'.

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_misc import get_basis_name
sage: get_basis_name('adem', 2)
'serre-cartan'
sage: get_basis_name('milnor', 2)
'milnor'
sage: get_basis_name('MiLNoR', 5)
'milnor'
sage: get_basis_name('pst-llex', 2)
'pst_llex'
sage: get_basis_name('wood_abcdedfg_y', 2)
```

```

'woody'
sage: get_basis_name('wood', 2)
Traceback (most recent call last):
...
ValueError: wood is not a recognized basis at the prime 2.
sage: get_basis_name('arnon--hello--long', 2)
'arnona_long'
sage: get_basis_name('arnona_long', p=5)
Traceback (most recent call last):
...
ValueError: arnona_long is not a recognized basis at the prime 5.
sage: get_basis_name('NOT_A_BASIS', 2)
Traceback (most recent call last):
...
ValueError: not_a_basis is not a recognized basis at the prime 2.
sage: get_basis_name('woody', 2, generic=True)
Traceback (most recent call last):
...
ValueError: woody is not a recognized basis for the generic Steenrod algebra at
↳the prime 2.

```

sage.algebras.steenrod.steenrod_algebra_misc.**is_valid_profile** (*profile*, *truncation_type*, *p*=2, *generic*=None)

True if *profile*, together with *truncation_type*, is a valid profile at the prime *p*.

INPUT:

- *profile* - when $p = 2$, a tuple or list of numbers; when p is odd, a pair of such lists
- *truncation_type* - either 0 or ∞
- p - prime number, optional, default 2
- *generic* - boolean, optional, default None

OUTPUT: True if the profile function is valid, False otherwise.

See the documentation for `sage.algebras.steenrod.steenrod_algebra` for descriptions of profile functions and how they correspond to sub-Hopf algebras of the Steenrod algebra. Briefly: at the prime 2, a profile function e is valid if it satisfies the condition

- $e(r) \geq \min(e(r-i) - i, e(i))$ for all $0 < i < r$.

At odd primes, a pair of profile functions e and k are valid if they satisfy

- $e(r) \geq \min(e(r-i) - i, e(i))$ for all $0 < i < r$.
- if $k(i+j) = 1$, then either $e(i) \leq j$ or $k(j) = 1$ for all $i \geq 1, j \geq 0$.

In this function, profile functions are lists or tuples, and *truncation_type* is appended as the last element of the list e before testing.

EXAMPLES:

$p = 2$:

```

sage: from sage.algebras.steenrod.steenrod_algebra_misc import is_valid_profile
sage: is_valid_profile([3,2,1], 0)
True
sage: is_valid_profile([3,2,1], Infinity)
True
sage: is_valid_profile([1,2,3], 0)

```



```

False
sage: is_valid_profile([6,2,0], Infinity)
False
sage: is_valid_profile([0,3], 0)
False
sage: is_valid_profile([0,0,4], 0)
False
sage: is_valid_profile([0,0,0,4,0], 0)
True

```

Odd primes:

```

sage: is_valid_profile(([0,0,0], [2,1,1,1,2,2]), 0, p=3)
True
sage: is_valid_profile([1], [2,2]), 0, p=3)
True
sage: is_valid_profile([1], [2]), 0, p=7)
False
sage: is_valid_profile([1,2,1], [], 0, p=7)
True
sage: is_valid_profile([0,0,0], [2,1,1,1,2,2]), 0, p=2, generic=True)
True

```

```

sage.algebras.steenrod.steenrod_algebra_misc.milnor_mono_to_string ( mono,
                                                                    la-
                                                                    tex=False,
                                                                    generic=False)

```

String representation of element of the Milnor basis.

This is used by the `_repr_` and `_latex_` methods.

INPUT:

- `mono` - if *generic* = *False*, tuple of non-negative integers (a,b,c,...); if *generic* = *True*, pair of tuples of non-negative integers ((e0, e1, e2, ...), (r1, r2, ...))
- `latex` - boolean (optional, default *False*), if true, output LaTeX string
- `generic` - whether to format generically, or for the prime 2 (default)

OUTPUT: `rep` - string

This returns a string like `Sq(a,b,c,...)` when *generic* = *False*, or a string like `Q_e0 Q_e1 Q_e2 ... P(r1,r2,...)` when *generic* = *True*.

EXAMPLES:

```

sage: from sage.algebras.steenrod.steenrod_algebra_misc import milnor_mono_to_
      ↪ string
sage: milnor_mono_to_string((1,2,3,4))
'Sq(1,2,3,4)'
sage: milnor_mono_to_string((1,2,3,4), latex=True)
'\text{Sq}(1,2,3,4)'
sage: milnor_mono_to_string(((1,0), (2,3,1)), generic=True)
'Q_{1} Q_{0} P(2,3,1)'
sage: milnor_mono_to_string(((1,0), (2,3,1)), latex=True, generic=True)
'Q_{1} Q_{0} \mathcal{P}(2,3,1)'

```

The empty tuple represents the unit element:

```
sage: milnor_mono_to_string(())
'1'
sage: milnor_mono_to_string((), generic=True)
'1'
```

```
sage.algebras.steenrod.steenrod_algebra_misc.normalize_profile(profile, precision=None,
                                                                truncation_type='auto',
                                                                p=2,
                                                                generic=None)
```

Given a profile function and related data, return it in a standard form, suitable for hashing and caching as data defining a sub-Hopf algebra of the Steenrod algebra.

INPUT:

- *profile* - a profile function in form specified below
- *precision* - integer or None, optional, default None
- *truncation_type* - 0 or ∞ or 'auto', optional, default 'auto'
- *p* - prime, optional, default 2
- *generic* - boolean, optional, default None

OUTPUT: a triple *profile*, *precision*, *truncation_type*, in standard form as described below.

The “standard form” is as follows: *profile* should be a tuple of integers (or ∞) with no trailing zeroes when $p = 2$, or a pair of such when p is odd or *generic* is True. *precision* should be a positive integer. *truncation_type* should be 0 or ∞ . Furthermore, this must be a valid profile, as determined by the function `is_valid_profile()`. See also the documentation for the module `sage.algebras.steenrod.steenrod_algebra` for information about profile functions.

For the inputs: when $p = 2$, *profile* should be a valid profile function, and it may be entered in any of the following forms:

- a list or tuple, e.g., `[3, 2, 1, 1]`
- a function from positive integers to non-negative integers (and ∞), e.g., `lambda n: n+2`. This corresponds to the list `[3, 4, 5, ...]`.
- None or Infinity - use this for the profile function for the whole Steenrod algebra. This corresponds to the list `[Infinity, Infinity, Infinity, ...]`

To make this hashable, it gets turned into a tuple. In the first case it is clear how to do this; also in this case, *precision* is set to be one more than the length of this tuple. In the second case, construct a tuple of length one less than *precision* (default value 100). In the last case, the empty tuple is returned and *precision* is set to 1.

Once a sub-Hopf algebra of the Steenrod algebra has been defined using such a profile function, if the code requires any remaining terms (say, terms after the 100th), then they are given by *truncation_type* if that is 0 or ∞ . If *truncation_type* is 'auto', then in the case of a tuple, it gets set to 0, while for the other cases it gets set to ∞ .

See the examples below.

When p is odd, *profile* is a pair of “functions”, so it may have the following forms:

- a pair of lists or tuples, the second of which takes values in the set $\{1, 2\}$, e.g., `([3, 2, 1, 1], [1, 1, 2, 2, 1])`.

- a pair of functions, one (called e) from positive integers to non-negative integers (and ∞), one (called k) from non-negative integers to the set $\{1, 2\}$, e.g., `(lambda n: n+2, lambda n: 1)`. This corresponds to the pair `([3, 4, 5, ...], [1, 1, 1, ...])`.
- `None` or `Infinity` - use this for the profile function for the whole Steenrod algebra. This corresponds to the pair `([Infinity, Infinity, Infinity, ...], [2, 2, 2, ...])`.

You can also mix and match the first two, passing a pair with first entry a list and second entry a function, for instance. The values of `precision` and `truncation_type` are determined by the first entry.

EXAMPLES:

$p = 2$:

```
sage: from sage.algebras.steenrod.steenrod_algebra_misc import normalize_profile
sage: normalize_profile([1, 2, 1, 0, 0])
((1, 2, 1), 0)
```

The full mod 2 Steenrod algebra:

```
sage: normalize_profile(Infinity)
((), +Infinity)
sage: normalize_profile(None)
((), +Infinity)
sage: normalize_profile(lambda n: Infinity)
((), +Infinity)
```

The `precision` argument has no effect when the first argument is a list or tuple:

```
sage: normalize_profile([1, 2, 1, 0, 0], precision=12)
((1, 2, 1), 0)
```

If the first argument is a function, then construct a list of length one less than `precision`, by plugging in the numbers 1, 2, ..., `precision - 1`:

```
sage: normalize_profile(lambda n: 4-n, precision=4)
((3, 2, 1), +Infinity)
sage: normalize_profile(lambda n: 4-n, precision=4, truncation_type=0)
((3, 2, 1), 0)
```

Negative numbers in profile functions are turned into zeroes:

```
sage: normalize_profile(lambda n: 4-n, precision=6)
((3, 2, 1, 0, 0), +Infinity)
```

If it doesn't give a valid profile, an error is raised:

```
sage: normalize_profile(lambda n: 3, precision=4, truncation_type=0)
Traceback (most recent call last):
...
ValueError: Invalid profile
sage: normalize_profile(lambda n: 3, precision=4, truncation_type = Infinity)
((3, 3, 3), +Infinity)
```

When p is odd, the behavior is similar:

```
sage: normalize_profile([2, 1], [2, 2, 2], p=13)
(((2, 1), (2, 2, 2)), 0)
```

The full mod p Steenrod algebra:

```
sage: normalize_profile(None, p=7)
(((), ()), +Infinity)
sage: normalize_profile(Infinity, p=11)
(((), ()), +Infinity)
sage: normalize_profile((lambda n: Infinity, lambda n: 2), p=17)
(((), ()), +Infinity)
```

Note that as at the prime 2, the `precision` argument has no effect on a list or tuple in either entry of profile. If `truncation_type` is 'auto', then it gets converted to either 0 or +Infinity depending on the *first* entry of profile:

```
sage: normalize_profile([2,1], [2,2,2]), precision=84, p=13)
((2, 1), (2, 2, 2)), 0)
sage: normalize_profile((lambda n: 0, lambda n: 2), precision=4, p=11)
((0, 0, 0), ()), +Infinity)
sage: normalize_profile((lambda n: 0, (1,1,1,1,1,1,1)), precision=4, p=11)
((0, 0, 0), (1, 1, 1, 1, 1, 1, 1)), +Infinity)
sage: normalize_profile((4,3,2,1), lambda n: 2), precision=6, p=11)
((4, 3, 2, 1), (2, 2, 2, 2, 2)), 0)
sage: normalize_profile((4,3,2,1), lambda n: 1), precision=3, p=11, truncation_
    type=Infinity)
((4, 3, 2, 1), (1, 1)), +Infinity)
```

As at the prime 2, negative numbers in the first component are converted to zeroes. Numbers in the second component must be either 1 and 2, or else an error is raised:

```
sage: normalize_profile((lambda n: -n, lambda n: 1), precision=4, p=11)
((0, 0, 0), (1, 1, 1)), +Infinity)
sage: normalize_profile([0,0,0], [1,2,3,2,1]), p=11)
Traceback (most recent call last):
...
ValueError: Invalid profile
```

`sage.algebras.steenrod.steenrod_algebra_misc.pst_mono_to_string (mono, latex=False, generic=False)`

String representation of element of a P_t^s -basis.

This is used by the `_repr_` and `_latex_` methods.

INPUT:

- `mono` - tuple of pairs of integers (s,t) with $s \geq 0, t > 0$
- `latex` - boolean (optional, default False), if true, output LaTeX string
- `generic` - whether to format generically, or for the prime 2 (default)

OUTPUT: string - concatenation of strings of the form $P^{\{s\}}_{\{t\}}$ for each pair (s,t)

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_misc import pst_mono_to_string
sage: pst_mono_to_string((1,2), (0,3)), generic=False)
'P^{1}_{2} P^{0}_{3}'
sage: pst_mono_to_string((1,2), (0,3)), latex=True, generic=False)
'P^{\{1\}}_{\{2\}} P^{\{0\}}_{\{3\}}'
sage: pst_mono_to_string((1,4), ((1,2), 1), ((0,3), 2)), generic=True)
'Q_{1} Q_{4} P^{\{1\}}_{\{2\}} (P^{\{0\}}_{\{3\}})^2'
```

```
sage: pst_mono_to_string(((1,4), ((1,2), 1), ((0,3), 2))), latex=True,
↳generic=True)
'Q_{1} Q_{4} P^{1}_{2} (P^{0}_{3})^{2}'
```

The empty tuple represents the unit element:

```
sage: pst_mono_to_string(())
'1'
```

```
sage.algebras.steenrod.steenrod_algebra_misc. serre_cartan_mono_to_string ( mono,
                                                                    la-
                                                                    tex=False,
                                                                    generic=False)
```

String representation of element of the Serre-Cartan basis.

This is used by the `_repr_` and `_latex_` methods.

INPUT:

- `mono` - tuple of positive integers (a,b,c,...) when *generic* = *False*, or tuple (e0, n1, e1, n2, ...) when *generic* = *True*, where each *ei* is 0 or 1, and each *ni* is positive
- `latex` - boolean (optional, default *False*), if true, output LaTeX string
- `generic` - whether to format generically, or for the prime 2 (default)

OUTPUT: `rep` - string

This returns a string like $Sq^{\{a\}} Sq^{\{b\}} Sq^{\{c\}} \dots$ when *generic* = *False*, or a string like $\beta^{e_0} P^{n_1} \beta^{e_1} P^{n_2} \dots$ when *generic* = *True*. is odd.

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_misc import serre_cartan_mono_
↳to_string
sage: serre_cartan_mono_to_string((1,2,3,4))
'Sq^{1} Sq^{2} Sq^{3} Sq^{4}'
sage: serre_cartan_mono_to_string((1,2,3,4), latex=True)
'\\text{Sq}^{1} \\text{Sq}^{2} \\text{Sq}^{3} \\text{Sq}^{4}'
sage: serre_cartan_mono_to_string((0,5,1,1,0), generic=True)
'P^{5} beta P^{1}'
sage: serre_cartan_mono_to_string((0,5,1,1,0), generic=True, latex=True)
'\\mathcal{P}^{5} \\beta \\mathcal{P}^{1}'
```

The empty tuple represents the unit element 1:

```
sage: serre_cartan_mono_to_string(())
'1'
sage: serre_cartan_mono_to_string((), generic=True)
'1'
```

```
sage.algebras.steenrod.steenrod_algebra_misc. wall_long_mono_to_string ( mono,
                                                                    la-
                                                                    tex=False)
```

Alternate string representation of element of Wall's basis.

This is used by the `_repr_` and `_latex_` methods.

INPUT:

- `mono` - tuple of pairs of non-negative integers (m,k) with $m \geq k$

- `latex` - boolean (optional, default False), if true, output LaTeX string

OUTPUT: string - concatenation of strings of the form $Sq^{(2^m)}$

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_misc import wall_long_mono_to_
      ↪ string
sage: wall_long_mono_to_string(((1,2), (3,0)))
'Sq^{1} Sq^{2} Sq^{4} Sq^{8}'
sage: wall_long_mono_to_string(((1,2), (3,0)), latex=True)
'\text{Sq}^{\{1\}} \text{Sq}^{\{2\}} \text{Sq}^{\{4\}} \text{Sq}^{\{8\}}'
```

The empty tuple represents the unit element:

```
sage: wall_long_mono_to_string(())
'1'
```

`sage.algebras.steenrod.steenrod_algebra_misc.wall_mono_to_string (mono, latex=False)`

String representation of element of Wall's basis.

This is used by the `_repr_` and `_latex_` methods.

INPUT:

- `mono` - tuple of pairs of non-negative integers (m,k) with $m \geq k$
- `latex` - boolean (optional, default False), if true, output LaTeX string

OUTPUT: string - concatenation of strings $Q^{\{m\}}_{\{k\}}$ for each pair (m,k)

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_misc import wall_mono_to_string
sage: wall_mono_to_string(((1,2), (3,0)))
'Q^{\{1\}}_{\{2\}} Q^{\{3\}}_{\{0\}}'
sage: wall_mono_to_string(((1,2), (3,0)), latex=True)
'Q^{\{1\}}_{\{2\}} Q^{\{3\}}_{\{0\}}'
```

The empty tuple represents the unit element:

```
sage: wall_mono_to_string(())
'1'
```

`sage.algebras.steenrod.steenrod_algebra_misc.wood_mono_to_string (mono, latex=False)`

String representation of element of Wood's Y and Z bases.

This is used by the `_repr_` and `_latex_` methods.

INPUT:

- `mono` - tuple of pairs of non-negative integers (s,t)
- `latex` - boolean (optional, default False), if true, output LaTeX string

OUTPUT: string - concatenation of strings of the form $Sq^{2^s} (2^{t+1}-1)$ for each pair (s,t)

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_misc import wood_mono_to_string
sage: wood_mono_to_string(((1,2), (3,0)))
'Sq^{14} Sq^{8}'
```

```
sage: wood_mono_to_string(((1,2),(3,0)), latex=True)
'\text{Sq}^{\{14\}} \text{Sq}^{\{8\}}'
```

The empty tuple represents the unit element:

```
sage: wood_mono_to_string(())
'1'
```

4.20 Multiplication for elements of the Steenrod algebra

AUTHORS:

- John H. Palmieri (2008-07-30: version 0.9) initial version: Milnor multiplication.
- John H. Palmieri (2010-06-30: version 1.0) multiplication of Serre-Cartan basis elements using the Adem relations. - Simon King (2011-10-25): Fix the use of cached functions.

Milnor multiplication, $p = 2$

See Milnor's paper [Mil1958] for proofs, etc.

To multiply Milnor basis elements $\text{Sq}(r_1, r_2, \dots)$ and $\text{Sq}(s_1, s_2, \dots)$ at the prime 2, form all possible matrices M with rows and columns indexed starting at 0, with position (0,0) deleted (or ignored), with s_i equal to the sum of column i for each i , and with r_j equal to the 'weighted' sum of row j . The weights are as follows: elements from column i are multiplied by 2^i . For example, to multiply $\text{Sq}(2)$ and $\text{Sq}(1, 1)$, form the matrices

$$\begin{vmatrix} * & 1 & 1 \\ 2 & 0 & 0 \end{vmatrix} \quad \text{and} \quad \begin{vmatrix} * & 0 & 1 \\ 0 & 1 & 0 \end{vmatrix}$$

(The $*$ is the ignored (0,0)-entry of the matrix.) For each such matrix M , compute a multinomial coefficient, mod 2: for each diagonal $\{m_{ij} : i + j = n\}$, compute $(\sum m_{i,j}!)/(m_{0,n}!m_{1,n-1}!\dots m_{n,0}!)$. Multiply these together for all n . (To compute this mod 2, view the entries of the matrix as their base 2 expansions; then this coefficient is zero if and only if there is some diagonal containing two numbers which have a summand in common in their base 2 expansion. For example, if 3 and 10 are in the same diagonal, the coefficient is zero, because $3 = 1 + 2$ and $10 = 2 + 8$: they both have a summand of 2.)

Now, for each matrix with multinomial coefficient 1, let t_n be the sum of the n th diagonal in the matrix; then

$$\text{Sq}(r_1, r_2, \dots)\text{Sq}(s_1, s_2, \dots) = \sum \text{Sq}(t_1, t_2, \dots)$$

The function `milnor_multiplication()` takes as input two tuples of non-negative integers, r and s , which represent $\text{Sq}(r) = \text{Sq}(r_1, r_2, \dots)$ and $\text{Sq}(s) = \text{Sq}(s_1, s_2, \dots)$; it returns as output a dictionary whose keys are tuples $t = (t_1, t_2, \dots)$ of non-negative integers, and for each tuple the associated value is the coefficient of $\text{Sq}(t)$ in the product formula. (Since we are working mod 2, this coefficient is 1 – if it is zero, the element is omitted from the dictionary altogether).

Milnor multiplication, odd primes

As for the $p = 2$ case, see Milnor's paper [Mil1958] for proofs.

Fix an odd prime p . There are three steps to multiply Milnor basis elements $Q_{f_1}Q_{f_2}\dots\mathcal{P}(q_1, q_2, \dots)$ and $Q_{g_1}Q_{g_2}\dots\mathcal{P}(s_1, s_2, \dots)$: first, use the formula

$$\mathcal{P}(q_1, q_2, \dots)Q_k = Q_k\mathcal{P}(q_1, q_2, \dots) + Q_{k+1}\mathcal{P}(q_1 - p^k, q_2, \dots) + Q_{k+2}\mathcal{P}(q_1, q_2 - p^k, \dots) + \dots$$

Second, use the fact that the Q_k 's form an exterior algebra: $Q_k^2 = 0$ for all k , and if $i \neq j$, then Q_i and Q_j anticommute: $Q_i Q_j = -Q_j Q_i$. After these two steps, the product is a linear combination of terms of the form

$$Q_{e_1} Q_{e_2} \dots \mathcal{P}(r_1, r_2, \dots) \mathcal{P}(s_1, s_2, \dots).$$

Finally, use Milnor matrices to multiply the pairs of $\mathcal{P}(\dots)$ terms, as at the prime 2: form all possible matrices M with rows and columns indexed starting at 0, with position (0,0) deleted (or ignored), with s_i equal to the sum of column i for each i , and with r_j equal to the weighted sum of row j : elements from column i are multiplied by p^i . For example when $p = 5$, to multiply $\mathcal{P}(5)$ and $\mathcal{P}(1, 1)$, form the matrices

$$\begin{vmatrix} * & 1 & 1 \\ 5 & 0 & 0 \end{vmatrix} \quad \text{and} \quad \begin{vmatrix} * & 0 & 1 \\ 0 & 1 & 0 \end{vmatrix}$$

For each such matrix M , compute a multinomial coefficient, mod p : for each diagonal $\{m_{ij} : i + j = n\}$, compute $(\sum m_{i,j}!) / (m_{0,n}! m_{1,n-1}! \dots m_{n,0}!)$. Multiply these together for all n .

Now, for each matrix with nonzero multinomial coefficient b_M , let t_n be the sum of the n -th diagonal in the matrix; then

$$\mathcal{P}(r_1, r_2, \dots) \mathcal{P}(s_1, s_2, \dots) = \sum b_M \mathcal{P}(t_1, t_2, \dots)$$

For example when $p = 5$, we have

$$\mathcal{P}(5) \mathcal{P}(1, 1) = \mathcal{P}(6, 1) + 2\mathcal{P}(0, 2).$$

The function `milnor_multiplication()` takes as input two pairs of tuples of non-negative integers, (g, q) and (f, s) , which represent $Q_{g_1} Q_{g_2} \dots \mathcal{P}(q_1, q_2, \dots)$ and $Q_{f_1} Q_{f_2} \dots \mathcal{P}(s_1, s_2, \dots)$. It returns as output a dictionary whose keys are pairs of tuples (e, t) of non-negative integers, and for each tuple the associated value is the coefficient in the product formula.

The Adem relations and admissible sequences

If $p = 2$, then the mod 2 Steenrod algebra is generated by Steenrod squares Sq^a for $a \geq 0$ (equal to the Milnor basis element $Sq(a)$). The *Adem relations* are as follows: if $a < 2b$,

$$Sq^a Sq^b = \sum_{j=0}^{a/2} \binom{b-j-1}{a-2j} Sq^{a+b-j} Sq^j$$

A monomial $Sq^{i_1} Sq^{i_2} \dots Sq^{i_n}$ is called *admissible* if $i_k \geq 2i_{k+1}$ for all k . One can use the Adem relations to show that the admissible monomials span the Steenrod algebra, as a vector space; with more work, one can show that the admissible monomials are also linearly independent. They form the *Serre-Cartan* basis for the Steenrod algebra. To multiply a collection of admissible monomials, concatenate them and see if the result is admissible. If it is, you're done. If not, find the first pair $Sq^a Sq^b$ where it fails to be admissible and apply the Adem relations there. Repeat with the resulting terms. One can prove that this process terminates in a finite number of steps, and therefore gives a procedure for multiplying elements of the Serre-Cartan basis.

At an odd prime p , the Steenrod algebra is generated by the p th power operations \mathcal{P}^a (the same as $\mathcal{P}(a)$ in the Milnor basis) and the Bockstein operation β ($= Q_0$ in the Milnor basis). The odd primary *Adem relations* are as follows: if $a < pb$,

$$\mathcal{P}^a \mathcal{P}^b = \sum_{j=0}^{a/p} (-1)^{a+j} \binom{(b-j)(p-1)-1}{a-pj} \mathcal{P}^{a+b-j} \mathcal{P}^j$$

Also, if $a \leq pb$,

$$\mathcal{P}^a \beta \mathcal{P}^b = \sum_{j=0}^{a/p} (-1)^{a+j} \binom{(b-j)(p-1)}{a-pj} \beta \mathcal{P}^{a+b-j} \mathcal{P}^j + \sum_{j=0}^{a/p} (-1)^{a+j-1} \binom{(b-j)(p-1)-1}{a-pj-1} \mathcal{P}^{a+b-j} \beta \mathcal{P}^j$$

The *admissible* monomials at an odd prime are products of the form

$$\beta^{\epsilon_0} \mathcal{P}^{s_1} \beta^{\epsilon_1} \mathcal{P}^{s_2} \dots \mathcal{P}^{s_n} \beta^{\epsilon_n}$$

where $s_k \geq \epsilon_{k+1} + ps_{k+1}$ for all k . As at the prime 2, these form a basis for the Steenrod algebra.

The main function for this is `make_mono_admissible_()` (and in practice, one should use the cached version, `make_mono_admissible`), which converts a product of Steenrod squares or pth power operations and Bocksteins into a dictionary representing a sum of admissible monomials.

`sage.algebras.steenrod.steenrod_algebra_mult.adem (a, b, c=0, p=2, generic=None)`

The mod p Adem relations

INPUT:

- a, b, c (optional) - nonnegative integers, corresponding to either $P^a P^b$ or (if c present) to $P^a \beta^b P^c$
- p - positive prime number (optional, default 2)
- *generic* - whether to use the generic Steenrod algebra, (default: depends on prime)

OUTPUT:

a dictionary representing the mod p Adem relations applied to $P^a P^b$ or (if c present) to $P^a \beta^b P^c$.

Note: Users should use `adem()` instead of this function (which has a trailing underscore in its name): `adem()` is the cached version of this one, and so will be faster.

The mod p Adem relations for the mod p Steenrod algebra are as follows: if $p = 2$, then if $a < 2b$,

$$\text{Sq}^a \text{Sq}^b = \sum_{j=0}^{a/2} \binom{b-j-1}{a-2j} \text{Sq}^{a+b-j} \text{Sq}^j$$

If p is odd, then if $a < pb$,

$$P^a P^b = \sum_{j=0}^{a/p} (-1)^{a+j} \binom{(b-j)(p-1)-1}{a-pj} P^{a+b-j} P^j$$

Also for p odd, if $a \leq pb$,

$$P^a \beta P^b = \sum_{j=0}^{a/p} (-1)^{a+j} \binom{(b-j)(p-1)}{a-pj} \beta P^{a+b-j} P^j + \sum_{j=0}^{a/p} (-1)^{a+j-1} \binom{(b-j)(p-1)-1}{a-pj-1} P^{a+b-j} \beta P^j$$

EXAMPLES:

If two arguments (a and b) are given, then computations are done mod 2. If $a \geq 2b$, then the dictionary $\{(a,b): 1\}$ is returned. Otherwise, the right side of the mod 2 Adem relation for $\text{Sq}^a \text{Sq}^b$ is returned. For example, since $\text{Sq}^2 \text{Sq}^2 = \text{Sq}^3 \text{Sq}^1$, we have:

```
sage: from sage.algebras.steenrod.steenrod_algebra_mult import adem
sage: adem(2,2) # indirect doctest
{(3, 1): 1}
sage: adem(4,2)
{(4, 2): 1}
sage: adem(4,4)
{(6, 2): 1, (7, 1): 1}
```

If p is given and is odd, then with two inputs a and b , the Adem relation for $P^a P^b$ is computed. With three inputs a, b, c , the Adem relation for $P^a \beta^b P^c$ is computed. In either case, the keys in the output are all tuples of odd length, with (i_1, i_2, \dots, i_m) representing

$$\beta^{i_1} P^{i_2} \beta^{i_3} P^{i_4} \dots \beta^{i_m}$$

For instance:

```
sage: adem(3,1, p=3)
{(0, 3, 0, 1, 0): 1}
sage: adem(3,0,1, p=3)
{(0, 3, 0, 1, 0): 1}
sage: adem(1,0,1, p=7)
{(0, 2, 0): 2}
sage: adem(1,1,1, p=5)
{(0, 2, 1): 1, (1, 2, 0): 1}
sage: adem(1,1,2, p=5)
{(0, 3, 1): 1, (1, 3, 0): 2}
```

`sage.algebras.steenrod.steenrod_algebra_mult.binomial_mod2 (n, k)`

The binomial coefficient $\binom{n}{k}$, computed mod 2.

INPUT:

• n, k - integers

OUTPUT:

n choose k , mod 2

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_mult import binomial_mod2
sage: binomial_mod2(4,2)
0
sage: binomial_mod2(5,4)
1
sage: binomial_mod2(3 * 32768, 32768)
1
sage: binomial_mod2(4 * 32768, 32768)
0
```

`sage.algebras.steenrod.steenrod_algebra_mult.binomial_modp (n, k, p)`

The binomial coefficient $\binom{n}{k}$, computed mod p .

INPUT:

• n, k - integers

• p - prime number

OUTPUT:

n choose k , mod p

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_mult import binomial_modp
sage: binomial_modp(5,2,3)
1
sage: binomial_modp(6,2,11) # 6 choose 2 = 15
4
```

```
sage.algebras.steenrod.steenrod_algebra_mult.make_mono_admissible ( mono,
                                                                    p=2,
                                                                    generic=None)
```

Given a tuple *mono*, view it as a product of Steenrod operations, and return a dictionary giving data equivalent to writing that product as a linear combination of admissible monomials.

When $p = 2$, the sequence (and hence the corresponding monomial) (i_1, i_2, \dots) is admissible if $i_j \geq 2i_{j+1}$ for all j .

When p is odd, the sequence $(e_1, i_1, e_2, i_2, \dots)$ is admissible if $i_j \geq e_{j+1} + pi_{j+1}$ for all j .

INPUT:

- *mono* - a tuple of non-negative integers
- *p* - prime number, optional (default 2)
- *generic* - whether to use the generic Steenrod algebra, (default: depends on prime)

OUTPUT:

Dictionary of terms of the form (tuple: coeff), where ‘tuple’ is an admissible tuple of non-negative integers and ‘coeff’ is its coefficient. This corresponds to a linear combination of admissible monomials. When p is odd, each tuple must have an odd length: it should be of the form $(e_1, i_1, e_2, i_2, \dots, e_k)$ where each e_j is either 0 or 1 and each i_j is a positive integer: this corresponds to the admissible monomial

$$\beta^{e_1} \mathcal{P}^{i_1} \beta^{e_2} \mathcal{P}^{i_2} \dots \mathcal{P}^{i_k} \beta^{e_k}$$

ALGORITHM:

Given (i_1, i_2, i_3, \dots) , apply the Adem relations to the first pair (or triple when p is odd) where the sequence is inadmissible, and then apply this function recursively to each of the resulting tuples $(i_1, \dots, i_{j-1}, NEW, i_{j+2}, \dots)$, keeping track of the coefficients.

Note: Users should use `make_mono_admissible()` instead of this function (which has a trailing underscore in its name): `make_mono_admissible()` is the cached version of this one, and so will be faster.

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_mult import make_mono_
      ↪admissible
sage: make_mono_admissible((12,)) # already admissible, indirect doctest
{(12,): 1}
sage: make_mono_admissible((2,1)) # already admissible
{(2, 1): 1}
sage: make_mono_admissible((2,2))
{(3, 1): 1}
sage: make_mono_admissible((2, 2, 2))
{(5, 1): 1}
sage: make_mono_admissible((0, 2, 0, 1, 0), p=7)
{(0, 3, 0): 3}
```

Test the fix from [trac ticket #13796](#):

```
sage: SteenrodAlgebra(p=2, basis='adem').Q(2) * (Sq(6) * Sq(2)) # indirect doctest
Sq^10 Sq^4 Sq^1 + Sq^10 Sq^5 + Sq^12 Sq^3 + Sq^13 Sq^2
```

```
sage.algebras.steenrod.steenrod_algebra_mult.milnor_multiplication ( r, s)
```

Product of Milnor basis elements *r* and *s* at the prime 2.

INPUT:

- r - tuple of non-negative integers
- s - tuple of non-negative integers

OUTPUT:

Dictionary of terms of the form (tuple: coeff), where ‘tuple’ is a tuple of non-negative integers and ‘coeff’ is 1.

This computes Milnor matrices for the product of $Sq(r)$ and $Sq(s)$, computes their multinomial coefficients, and for each matrix whose coefficient is 1, add $Sq(t)$ to the output, where t is the tuple formed by the diagonals sums from the matrix.

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_mult import milnor_
      ↪multiplication
sage: milnor_multiplication((2,), (1,))
{(0, 1): 1, (3,): 1}
sage: milnor_multiplication((4,), (2,1))
{(0, 3): 1, (2, 0, 1): 1, (6, 1): 1}
sage: milnor_multiplication((2,4), (0,1))
{(2, 0, 0, 1): 1, (2, 5): 1}
```

These examples correspond to the following product computations:

$$\begin{aligned} Sq(2)Sq(1) &= Sq(0, 1) + Sq(3) \\ Sq(4)Sq(2, 1) &= Sq(6, 1) + Sq(0, 3) + Sq(2, 0, 1) \\ Sq(2, 4)Sq(0, 1) &= Sq(2, 5) + Sq(2, 0, 0, 1) \end{aligned}$$

This uses the same algorithm Monks does in his Maple package: see <http://mathweb.scranton.edu/monks/software/Steenrod/steen.html>.

```
sage.algebras.steenrod.steenrod_algebra_mult.milnor_multiplication_odd ( m1,
                                                                           m2,
                                                                           p)
```

Product of Milnor basis elements defined by $m1$ and $m2$ at the odd prime p .

INPUT:

- $m1$ - pair of tuples (e, r) , where e is an increasing tuple of non-negative integers and r is a tuple of non-negative integers
- $m2$ - pair of tuples (f, s) , same format as $m1$
- p - odd prime number

OUTPUT:

Dictionary of terms of the form (tuple: coeff), where ‘tuple’ is a pair of tuples, as for r and s , and ‘coeff’ is an integer mod p .

This computes the product of the Milnor basis elements $Q_{e_1}Q_{e_2}\dots P(r_1, r_2, \dots)$ and $Q_{f_1}Q_{f_2}\dots P(s_1, s_2, \dots)$.

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_mult import milnor_
      ↪multiplication_odd
sage: milnor_multiplication_odd(((0,2),(5,)), ((1,),(1,)), 5)
{((0, 1, 2), (0, 1)): 4, ((0, 1, 2), (6,)): 4}
sage: milnor_multiplication_odd(((0,2,4),()), ((1,3),()), 7)
{((0, 1, 2, 3, 4), ()): 6}
```

```
sage: milnor_multiplication_odd(((0,2,4),()), ((1,5),()), 7)
{((0, 1, 2, 4, 5), ()): 1}
sage: milnor_multiplication_odd(((),(6,)), (((),(2,))), 3)
{(((), (0, 2)): 1, (((), (4, 1)): 1, (((), (8,)): 1}
```

These examples correspond to the following product computations:

$$\begin{aligned}
 p = 5: \quad Q_0 Q_2 \mathcal{P}(5) Q_1 \mathcal{P}(1) &= 4Q_0 Q_1 Q_2 \mathcal{P}(0,1) + 4Q_0 Q_1 Q_2 \mathcal{P}(6) \\
 p = 7: \quad (Q_0 Q_2 Q_4)(Q_1 Q_3) &= 6Q_0 Q_1 Q_2 Q_3 Q_4 \\
 p = 7: \quad (Q_0 Q_2 Q_4)(Q_1 Q_5) &= Q_0 Q_1 Q_2 Q_3 Q_5 \\
 p = 3: \quad \mathcal{P}(6) \mathcal{P}(2) &= \mathcal{P}(0,2) + \mathcal{P}(4,1) + \mathcal{P}(8)
 \end{aligned}$$

The following used to fail until the trailing zeroes were eliminated in `p_mono`:

```
sage: A = SteenrodAlgebra(3)
sage: a = A.P(0,3); b = A.P(12); c = A.Q(1,2)
sage: (a+b)*c == a*c + b*c
True
```

Test that the bug reported in #7212 has been fixed:

```
sage: A.P(36,6)*A.P(27,9,81)
2 P(13,21,83) + P(14,24,82) + P(17,20,83) + P(25,18,83) + P(26,21,82) +
↪ P(36,15,80,1) + P(49,12,83) + 2 P(50,15,82) + 2 P(53,11,83) + 2 P(63,15,81)
```

Associativity once failed because of a sign error:

```
sage: a,b,c = A.Q_exp(0,1), A.P(3), A.Q_exp(1,1)
sage: (a*b)*c == a*(b*c)
True
```

This uses the same algorithm Monks does in his Maple package to iterate through the possible matrices: see <http://mathweb.scranton.edu/monks/software/Steenrod/steen.html>.

`sage.algebras.steenrod.steenrod_algebra_mult.multinomial (list)`
Multinomial coefficient of list, mod 2.

INPUT:

- list - list of integers

OUTPUT:

None if the multinomial coefficient is 0, or sum of list if it is 1

Given the input $[n_1, n_2, n_3, \dots]$, this computes the multinomial coefficient $(n_1 + n_2 + n_3 + \dots)! / (n_1! n_2! n_3! \dots)$, mod 2. The method is roughly this: expand each n_i in binary. If there is a 1 in the same digit for any n_i and n_j with $i \neq j$, then the coefficient is 0; otherwise, it is 1.

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_mult import multinomial
sage: multinomial([1,2,4])
7
sage: multinomial([1,2,5])
sage: multinomial([1,2,12,192,256])
463
```

This function does not compute any factorials, so the following are actually reasonable to do:

```

sage: multinomial([1, 65536])
65537
sage: multinomial([4, 65535])
sage: multinomial([32768, 65536])
98304

```

`sage.algebras.steenrod.steenrod_algebra_mult.multinomial_odd (list, p)`
 Multinomial coefficient of list, mod p.

INPUT:

- list - list of integers
- p - a prime number

OUTPUT:

Associated multinomial coefficient, mod p

Given the input $[n_1, n_2, n_3, \dots]$, this computes the multinomial coefficient $(n_1 + n_2 + n_3 + \dots)! / (n_1! n_2! n_3! \dots)$, mod p . The method is this: expand each n_i in base p : $n_i = \sum_j p^j n_{ij}$. Do the same for the sum of the n_i 's, which we call m : $m = \sum_j p^j m_j$. Then the multinomial coefficient is congruent, mod p , to the product of the multinomial coefficients $m_j! / (n_{1j}! n_{2j}! \dots)$.

Furthermore, any multinomial coefficient $m! / (n_1! n_2! \dots)$ can be computed as a product of binomial coefficients: it equals

$$\binom{n_1}{n_1} \binom{n_1 + n_2}{n_2} \binom{n_1 + n_2 + n_3}{n_3} \dots$$

This is convenient because Sage's binomial function returns integers, not rational numbers (as would be produced just by dividing factorials).

EXAMPLES:

```

sage: from sage.algebras.steenrod.steenrod_algebra_mult import multinomial_odd
sage: multinomial_odd([1, 2, 4], 2)
1
sage: multinomial_odd([1, 2, 4], 7)
0
sage: multinomial_odd([1, 2, 4], 11)
6
sage: multinomial_odd([1, 2, 4], 101)
4
sage: multinomial_odd([1, 2, 4], 107)
105

```

4.21 Weyl Algebras

AUTHORS:

- Travis Scrimshaw (2013-09-06): Initial version

class `sage.algebras.weyl_algebra.DifferentialWeylAlgebra (R, names=None)`
 Bases: `sage.rings.ring.Algebra`, `sage.structure.unique_representation.UniqueRepresentation`

The differential Weyl algebra of a polynomial ring.

Let R be a commutative ring. The (differential) Weyl algebra W is the algebra generated by $x_1, x_2, \dots, x_n, \partial_{x_1}, \partial_{x_2}, \dots, \partial_{x_n}$ subject to the relations: $[x_i, x_j] = 0$, $[\partial_{x_i}, \partial_{x_j}] = 0$, and $\partial_{x_i} x_j = x_j \partial_{x_i} + \delta_{ij}$. Therefore ∂_{x_i} is acting as the partial differential operator on x_i .

The Weyl algebra can also be constructed as an iterated Ore extension of the polynomial ring $R[x_1, x_2, \dots, x_n]$ by adding x_i at each step. It can also be seen as a quantization of the symmetric algebra $Sym(V)$, where V is a finite dimensional vector space over a field of characteristic zero, by using a modified Groenewold-Moyal product in the symmetric algebra.

The Weyl algebra (even for $n = 1$) over a field of characteristic 0 has many interesting properties.

- It's a non-commutative domain.
- It's a simple ring (but not in positive characteristic) that is not a matrix ring over a division ring.
- It has no finite-dimensional representations.
- It's a quotient of the universal enveloping algebra of the Heisenberg algebra \mathfrak{h}_n .

REFERENCES:

- [Wikipedia article Weyl_algebra](#)

INPUT:

- R – a (polynomial) ring
- `names` – (default: `None`) if `None` and R is a polynomial ring, then the variable names correspond to those of R ; otherwise if `names` is specified, then R is the base ring

EXAMPLES:

There are two ways to create a Weyl algebra, the first is from a polynomial ring:

```
sage: R.<x,y,z> = QQ[]
sage: W = DifferentialWeylAlgebra(R); W
Differential Weyl algebra of polynomials in x, y, z over Rational Field
```

We can call `W.inject_variables()` to give the polynomial ring variables, now as elements of W , and the differentials:

```
sage: W.inject_variables()
Defining x, y, z, dx, dy, dz
sage: (dx * dy * dz) * (x^2 * y * z + x * z * dy + 1)
x*z*dx*dy^2*dz + z*dy^2*dz + x^2*y*z*dx*dy*dz + dx*dy*dz
+ x*dx*dy^2 + 2*x*y*z*dy*dz + dy^2 + x^2*z*dx*dz + x^2*y*dx*dy
+ 2*x*z*dz + 2*x*y*dy + x^2*dx + 2*x
```

Or directly by specifying a base ring and variable names:

```
sage: W.<a,b> = DifferentialWeylAlgebra(QQ); W
Differential Weyl algebra of polynomials in a, b over Rational Field
```

Todo

Implement the `graded_algebra()` as a polynomial ring once they are considered to be graded rings (algebras).

Element

alias of *DifferentialWeylAlgebraElement*

algebra_generators ()Return the algebra generators of `self`.**See also:**`variables()`, `differentials()`

EXAMPLES:

```

sage: R.<x,y,z> = QQ[]
sage: W = DifferentialWeylAlgebra(R)
sage: W.algebra_generators()
Finite family {'dz': dz, 'dx': dx, 'dy': dy, 'y': y, 'x': x, 'z': z}

```

basis ()Return a basis of `self`.

EXAMPLES:

```

sage: W.<x,y> = DifferentialWeylAlgebra(QQ)
sage: B = W.basis()
sage: it = iter(B)
sage: [next(it) for i in range(20)]
[1, x, y, dx, dy, x^2, x*y, x*dx, x*dy, y^2, y*dx, y*dy,
 dx^2, dx*dy, dy^2, x^3, x^2*y, x^2*dx, x^2*dy, x*y^2]
sage: dx, dy = W.differentials()
sage: (dx*x).monomials()
[1, x*dx]
sage: B[(x*y).support()[0]]
x*y
sage: sorted((dx*x).monomial_coefficients().items())
[((0, 0), (0, 0)), 1], (((1, 0), (1, 0)), 1)]

```

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

EXAMPLES:

```

sage: W.<a,b> = DifferentialWeylAlgebra(QQ)
sage: W.degree_on_basis( ((1, 3, 2), (0, 1, 3)) )
10

sage: W.<x,y,z> = DifferentialWeylAlgebra(QQ)
sage: dx,dy,dz = W.differentials()
sage: elt = y*dy - (3*x - z)*dx
sage: elt.degree()
2

```

differentials ()Return the differentials of `self`.**See also:**`algebra_generators()`, `variables()`

EXAMPLES:

```

sage: W.<x,y,z> = DifferentialWeylAlgebra(QQ)
sage: W.differentials()
Finite family {'dz': dz, 'dx': dx, 'dy': dy}

```


gen (*i*)Return the *i* -th generator of *self* .**See also:***algebra_generators()***EXAMPLES:**

```

sage: R.<x,y,z> = QQ[]
sage: W = DifferentialWeylAlgebra(R)
sage: [W.gen(i) for i in range(6)]
[x, y, z, dx, dy, dz]

```

ngens ()Return the number of generators of *self* .**EXAMPLES:**

```

sage: R.<x,y,z> = QQ[]
sage: W = DifferentialWeylAlgebra(R)
sage: W.ngens()
6

```

one ()

Return the multiplicative identity element 1.

EXAMPLES:

```

sage: R.<x,y,z> = QQ[]
sage: W = DifferentialWeylAlgebra(R)
sage: W.one()
1

```

polynomial_ring ()Return the associated polynomial ring of *self* .**EXAMPLES:**

```

sage: W.<a,b> = DifferentialWeylAlgebra(QQ)
sage: W.polynomial_ring()
Multivariate Polynomial Ring in a, b over Rational Field

```

```

sage: R.<x,y,z> = QQ[]
sage: W = DifferentialWeylAlgebra(R)
sage: W.polynomial_ring() == R
True

```

variables ()Return the variables of *self* .**See also:***algebra_generators()*, *differentials()***EXAMPLES:**

```

sage: W.<x,y,z> = DifferentialWeylAlgebra(QQ)
sage: W.variables()
Finite family {'y': y, 'x': x, 'z': z}

```

zero ()

Return the additive identity element 0.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: W = DifferentialWeylAlgebra(R)
sage: W.zero()
0
```

class sage.algebras.weyl_algebra. **DifferentialWeylAlgebraElement** (*parent, monomials*)

Bases: sage.structure.element.AlgebraElement

An element in a differential Weyl algebra.

list ()

Return *self* as a list.

This list consists of pairs (m, c) , where m is a pair of tuples indexing a basis element of *self*, and c is the coordinate of *self* corresponding to this basis element. (Only nonzero coordinates are shown.)

EXAMPLES:

```
sage: W.<x,y,z> = DifferentialWeylAlgebra(QQ)
sage: dx,dy,dz = W.differentials()
sage: elt = dy - (3*x - z)*dx
sage: elt.list()
[(((0, 0, 0), (0, 1, 0)), 1),
 (((0, 0, 1), (1, 0, 0)), 1),
 (((1, 0, 0), (1, 0, 0)), -3)]
```

monomial_coefficients (*copy=True*)

Return a dictionary which has the basis keys in the support of *self* as keys and their corresponding coefficients as values.

INPUT:

- *copy* – (default: *True*) if *self* is internally represented by a dictionary *d*, then make a copy of *d*; if *False*, then this can cause undesired behavior by mutating *d*

EXAMPLES:

```
sage: W.<x,y,z> = DifferentialWeylAlgebra(QQ)
sage: dx,dy,dz = W.differentials()
sage: elt = (dy - (3*x - z)*dx)
sage: sorted(elt.monomial_coefficients().items())
[(((0, 0, 0), (0, 1, 0)), 1),
 (((0, 0, 1), (1, 0, 0)), 1),
 (((1, 0, 0), (1, 0, 0)), -3)]
```

support ()

Return the support of *self*.

EXAMPLES:

```
sage: W.<x,y,z> = DifferentialWeylAlgebra(QQ)
sage: dx,dy,dz = W.differentials()
sage: elt = dy - (3*x - z)*dx + 1
sage: elt.support()
[(((0, 0, 0), (0, 1, 0)),
```

```
((1, 0, 0), (1, 0, 0)),
((0, 0, 0), (0, 0, 0)),
((0, 0, 1), (1, 0, 0))]
```

`sage.algebras.weyl_algebra.repr_from_monomials` (*monomials*, *term_repr*,
use_latex=False)

Return a string representation of an element of a free module from the dictionary `monomials`.

INPUT:

- `monomials` – a list of pairs `[m, c]` where `m` is the index and `c` is the coefficient
- `term_repr` – a function which returns a string given an index (can be `repr` or `latex`, for example)
- `use_latex` – (default: `False`) if `True` then the output is in latex format

EXAMPLES:

```
sage: from sage.algebras.weyl_algebra import repr_from_monomials
sage: R.<x,y,z> = QQ[]
sage: d = [(z, 4/7), (y, sqrt(2)), (x, -5)]
sage: repr_from_monomials(d, lambda m: repr(m))
'4/7*z + sqrt(2)*y - 5*x'
sage: a = repr_from_monomials(d, lambda m: latex(m), True); a
\frac{4}{7} z + \sqrt{2} y - 5 x
sage: type(a)
<class 'sage.misc.latex.LatexExpr'>
```

The zero element:

```
sage: repr_from_monomials([], lambda m: repr(m))
'0'
sage: a = repr_from_monomials([], lambda m: latex(m), True); a
0
sage: type(a)
<class 'sage.misc.latex.LatexExpr'>
```

A “unity” element:

```
sage: repr_from_monomials([(1, 1)], lambda m: repr(m))
'1'
sage: a = repr_from_monomials([(1, 1)], lambda m: latex(m), True); a
1
sage: type(a)
<class 'sage.misc.latex.LatexExpr'>
```

```
sage: repr_from_monomials([(1, -1)], lambda m: repr(m))
'-1'
sage: a = repr_from_monomials([(1, -1)], lambda m: latex(m), True); a
-1
sage: type(a)
<class 'sage.misc.latex.LatexExpr'>
```

Leading minus signs are dealt with appropriately:

```
sage: d = [(z, -4/7), (y, -sqrt(2)), (x, -5)]
sage: repr_from_monomials(d, lambda m: repr(m))
'-4/7*z - sqrt(2)*y - 5*x'
sage: a = repr_from_monomials(d, lambda m: latex(m), True); a
```

```

-\frac{4}{7} z - \sqrt{2} y - 5 x
sage: type(a)
<class 'sage.misc.latex.LatexExpr'>

```

Indirect doctests using a class that uses this function:

```

sage: R.<x,y> = QQ[]
sage: A = CliffordAlgebra(QuadraticForm(R, 3, [x,0,-1,3,-4,5]))
sage: a,b,c = A.gens()
sage: a*b*c
e0*e1*e2
sage: b*c
e1*e2
sage: (a*a + 2)
x + 2
sage: c*(a*a + 2)*b
(-x - 2)*e1*e2 - 4*x - 8
sage: latex(c*(a*a + 2)*b)
\left( - x - 2 \right) e_{1} e_{2} - 4 x - 8

```

4.22 Yokonuma-Hecke Algebras

AUTHORS:

- Travis Scrimshaw (2015-11): initial version

class `sage.algebras.yokonuma_hecke_algebra.YokonumaHeckeAlgebra` (*d, n, q, R*)
 Bases: `sage.combinat.free_module.CombinatorialFreeModule`

The Yokonuma-Hecke algebra $Y_{d,n}(q)$.

Let R be a commutative ring and q be a unit in R . The *Yokonuma-Hecke algebra* $Y_{d,n}(q)$ is the associative, unital R -algebra generated by $t_1, t_2, \dots, t_n, g_1, g_2, \dots, g_{n-1}$ and subject to the relations:

- $g_i g_j = g_j g_i$ for all $|i - j| > 1$,
- $g_i g_{i+1} g_i = g_{i+1} g_i g_{i+1}$,
- $t_i t_j = t_j t_i$,
- $t_j g_i = g_i t_{j s_i}$, and
- $t_j^d = 1$,

where s_i is the simple transposition $(i, i + 1)$, along with the quadratic relation

$$g_i^2 = 1 + \frac{(q - q^{-1})}{d} \left(\sum_{s=0}^{d-1} t_i^s t_{i+1}^{-s} \right) g_i.$$

Thus the Yokonuma-Hecke algebra can be considered a quotient of the framed braid group $(\mathbf{Z}/d\mathbf{Z}) \wr B_n$, where B_n is the classical braid group on n strands, by the quadratic relations. Moreover, all of the algebra generators are invertible. In particular, we have

$$g_i^{-1} = g_i - (q - q^{-1})e_i.$$

When we specialize $q = \pm 1$, we obtain the group algebra of the complex reflection group $G(d, 1, n) = (\mathbf{Z}/d\mathbf{Z}) \wr S_n$. Moreover for $d = 1$, the Yokonuma-Hecke algebra is equal to the `:class'Iwahori-Hecke<Iwahori-HeckeAlgebra>` of type A_{n-1} .

INPUT:

- d – the maximum power of t
- n – the number of generators
- q – (optional) an invertible element in a commutative ring; the default is $q \in \mathbb{Q}[q, q^{-1}]$
- R – (optional) a commutative ring containing q ; the default is the parent of q

EXAMPLES:

We construct $Y_{4,3}$ and do some computations:

```
sage: Y = algebras.YokonumaHecke(4, 3)
sage: g1, g2, t1, t2, t3 = Y.algebra_generators()
sage: g1 * g2
g[1,2]
sage: t1 * g1
t1*g[1]
sage: g2 * t2
t3*g[2]
sage: g2 * t3
t2*g[2]
sage: (g2 + t1) * (g1 + t2*t3)
g[2,1] + t2*t3*g[2] + t1*g[1] + t1*t2*t3
sage: g1 * g1
1 - (1/4*q^-1-1/4*q)*g[1] - (1/4*q^-1-1/4*q)*t1*t2^3*g[1]
- (1/4*q^-1-1/4*q)*t1^2*t2^2*g[1] - (1/4*q^-1-1/4*q)*t1^3*t2*g[1]
sage: g2 * g1 * t1
t3*g[2,1]
```

We construct the elements e_i and show that they are idempotents:

```
sage: e1 = Y.e(1); e1
1/4 + 1/4*t1*t2^3 + 1/4*t1^2*t2^2 + 1/4*t1^3*t2
sage: e1 * e1 == e1
True
sage: e2 = Y.e(2); e2
1/4 + 1/4*t2*t3^3 + 1/4*t2^2*t3^2 + 1/4*t2^3*t3
sage: e2 * e2 == e2
True
```

REFERENCES:

- [CL2013]
- [CPdA2014]
- [ERH2015]
- [JPdA15]

class Element (M, x)

Bases: `sage.combinat.free_module.CombinatorialFreeModuleElement`

Create a combinatorial module element. This should never be called directly, but only through the parent combinatorial free module's `__call__()` method.

inverse ()

Return the inverse if `self` is a basis element.

EXAMPLES:

```

sage: Y = algebras.YokonumaHecke(3, 3)
sage: t = prod(Y.t()); t
t1*t2*t3
sage: ~t
t1^2*t2^2*t3^2
sage: [3*(t*g) for g in Y.g()]
[(q^-1+q)*t2*t3^2 + (q^-1+q)*t1*t3^2
 + (q^-1+q)*t1^2*t2^2*t3^2 + 3*t1^2*t2^2*t3^2*g[1],
 (q^-1+q)*t1^2*t3 + (q^-1+q)*t1^2*t2
 + (q^-1+q)*t1^2*t2^2*t3^2 + 3*t1^2*t2^2*t3^2*g[2]]

```

`YokonumaHeckeAlgebra.algebra_generators()`

Return the algebra generators of `self`.

EXAMPLES:

```

sage: Y = algebras.YokonumaHecke(5, 3)
sage: dict(Y.algebra_generators())
{'g1': g[1], 'g2': g[2], 't1': t1, 't2': t2, 't3': t3}

```

`YokonumaHeckeAlgebra.e(i)`

Return the element e_i .

EXAMPLES:

```

sage: Y = algebras.YokonumaHecke(4, 3)
sage: Y.e(1)
1/4 + 1/4*t1*t2^3 + 1/4*t1^2*t2^2 + 1/4*t1^3*t2
sage: Y.e(2)
1/4 + 1/4*t2*t3^3 + 1/4*t2^2*t3^2 + 1/4*t2^3*t3

```

`YokonumaHeckeAlgebra.g(i=None)`

Return the generator(s) g_i .

INPUT:

- `i` – (default: `None`) the generator g_i or if `None`, then the list of all generators g_i

EXAMPLES:

```

sage: Y = algebras.YokonumaHecke(8, 3)
sage: Y.g(1)
g[1]
sage: Y.g()
[g[1], g[2]]

```

`YokonumaHeckeAlgebra.gens()`

Return the generators of `self`.

EXAMPLES:

```

sage: Y = algebras.YokonumaHecke(5, 3)
sage: Y.gens()
(g[1], g[2], t1, t2, t3)

```

`YokonumaHeckeAlgebra.inverse_g(i)`

Return the inverse of the generator g_i .

From the quadratic relation, we have

$$g_i^{-1} = g_i - (q - q^{-1})e_i.$$

EXAMPLES:

```
sage: Y = algebras.YokonumaHecke(2, 4)
sage: [2*Y.inverse_g(i) for i in range(1, 4)]
[(q^-1+q) + 2*g[1] + (q^-1+q)*t1*t2,
 (q^-1+q) + 2*g[2] + (q^-1+q)*t2*t3,
 (q^-1+q) + 2*g[3] + (q^-1+q)*t3*t4]
```

`YokonumaHeckeAlgebra.one_basis()`

Return the index of the basis element of 1.

EXAMPLES:

```
sage: Y = algebras.YokonumaHecke(5, 3)
sage: Y.one_basis()
((0, 0, 0), [1, 2, 3])
```

`YokonumaHeckeAlgebra.product_on_basis(m1, m2)`

Return the product of the basis elements indexed by `m1` and `m2`.

EXAMPLES:

```
sage: Y = algebras.YokonumaHecke(4, 3)
sage: m = ((1, 0, 2), Permutations(3)([2, 1, 3]))
sage: 4 * Y.product_on_basis(m, m)
-(q^-1-q)*t2^2*g[1] + 4*t1*t2 - (q^-1-q)*t1*t2*g[1]
- (q^-1-q)*t1^2*g[1] - (q^-1-q)*t1^3*t2^3*g[1]
```

Check that we apply the permutation correctly on t_i :

```
sage: Y = algebras.YokonumaHecke(4, 3)
sage: g1, g2, t1, t2, t3 = Y.algebra_generators()
sage: g21 = g2 * g1
sage: g21 * t1
t3*g[2, 1]
```

`YokonumaHeckeAlgebra.t(i=None)`

Return the generator(s) t_i .

INPUT:

- `i` – (default: `None`) the generator t_i or if `None`, then the list of all generators t_i

EXAMPLES:

```
sage: Y = algebras.YokonumaHecke(8, 3)
sage: Y.t(2)
t2
sage: Y.t()
[t1, t2, t3]
```


VARIOUS ASSOCIATIVE ALGEBRAS

5.1 Associated Graded Algebras To Filtered Algebras

AUTHORS:

- Travis Scrimshaw (2014-10-08): Initial version

class sage.algebras.associated_graded. **AssociatedGradedAlgebra** (*A*, *category=None*)
 Bases: sage.combinat.free_module.CombinatorialFreeModule

The associated graded algebra/module $\text{gr } A$ of a filtered algebra/module with basis A .

Let A be a filtered module over a commutative ring R . Let $(F_i)_{i \in I}$ be the filtration of A , with I being a totally ordered set. Define

$$G_i = F_i / \sum_{j < i} F_j$$

for every $i \in I$, and then

$$\text{gr } A = \bigoplus_{i \in I} G_i.$$

There are canonical projections $p_i : F_i \rightarrow G_i$ for every $i \in I$. Moreover $\text{gr } A$ is naturally a graded R -module with G_i being the i -th graded component. This graded R -module is known as the *associated graded module* (or, for short, just *graded module*) of A .

Now, assume that A (endowed with the filtration $(F_i)_{i \in I}$) is not just a filtered R -module, but also a filtered R -algebra. Let $u \in G_i$ and $v \in G_j$, and let $u' \in F_i$ and $v' \in F_j$ be lifts of u and v , respectively (so that $u = p_i(u')$ and $v = p_j(v')$). Then, we define a multiplication $*$ on $\text{gr } A$ (not to be mistaken for the multiplication of the original algebra A) by

$$u * v = p_{i+j}(u'v').$$

The *associated graded algebra* (or, for short, just *graded algebra*) of A is the graded algebra $\text{gr } A$ (endowed with this multiplication).

Now, assume that A is a filtered R -algebra with basis. Let $(b_x)_{x \in X}$ be the basis of A , and consider the partition $X = \bigsqcup_{i \in I} X_i$ of the set X , which is part of the data of a filtered algebra with basis. We know (see `FilteredModulesWithBasis`) that A (being a filtered R -module with basis) is canonically (when the basis is considered to be part of the data) isomorphic to $\text{gr } A$ as an R -module. Therefore the k -th graded component G_k can be identified with the span of $(b_x)_{x \in X_k}$, or equivalently the k -th homogeneous component of A . Suppose that $u'v' = \sum_{k \leq i+j} m_k$ where $m_k \in G_k$ (which has been identified with the k -th homogeneous component of A). Then $u * v = m_{i+j}$. We also note that the choice of identification of G_k with the k -th homogeneous component of A depends on the given basis.

The basis $(b_x)_{x \in X}$ of A gives rise to a basis of $\text{gr } A$. This latter basis is still indexed by the elements of X , and consists of the images of the b_x under the R -module isomorphism from A to $\text{gr } A$. It makes $\text{gr } A$ into a graded R -algebra with basis.

In this class, the R -module isomorphism from A to $\text{gr } A$ is implemented as `to_graded_conversion()` and also as the default conversion from A to $\text{gr } A$. Its inverse map is implemented as `from_graded_conversion()`. The projection $p_i : F_i \rightarrow G_i$ is implemented as `projection(i)`.

INPUT:

- A – a filtered module (or algebra) with basis

OUTPUT:

The associated graded module of A , if A is just a filtered R -module. The associated graded algebra of A , if A is a filtered R -algebra.

EXAMPLES:

Associated graded module of a filtered module:

```
sage: A = Modules(QQ).WithBasis().Filtered().example()
sage: grA = A.graded_algebra()
sage: grA.category()
Category of graded modules with basis over Rational Field
sage: x = A.basis()[Partition([3,2,1])]
sage: grA(x)
Bbar[[3, 2, 1]]
```

Associated graded algebra of a filtered algebra:

```
sage: A = Algebras(QQ).WithBasis().Filtered().example()
sage: grA = A.graded_algebra()
sage: grA.category()
Category of graded algebras with basis over Rational Field
sage: x,y,z = [grA.algebra_generators()[s] for s in ['x','y','z']]
sage: x
bar(U['x'])
sage: y * x + z
bar(U['x']*U['y']) + bar(U['z'])
sage: A(y) * A(x) + A(z)
U['x']*U['y']
```

We note that the conversion between A and $\text{gr } A$ is the canonical $\mathbb{Q}\mathbb{Q}$ -module isomorphism stemming from the fact that the underlying $\mathbb{Q}\mathbb{Q}$ -modules of A and $\text{gr } A$ are isomorphic:

```
sage: grA(A.an_element())
bar(U['x']^2*U['y']^2*U['z']^3) + 2*bar(U['x']) + 3*bar(U['y']) + bar(1)
sage: elt = A.an_element() + A.algebra_generators()['x'] + 2
sage: grelt = grA(elt); grelt
bar(U['x']^2*U['y']^2*U['z']^3) + 3*bar(U['x']) + 3*bar(U['y']) + 3*bar(1)
sage: A(grelt) == elt
True
```

Todo

The algebra A must currently be an instance of (a subclass of) `CombinatorialFreeModule`. This should work with any filtered algebra with a basis.

Todo

Implement a version of associated graded algebra for filtered algebras without a distinguished basis.

REFERENCES:

- [Wikipedia article Filtered_algebra#Associated_graded_algebra](#)

algebra_generators ()

Return the algebra generators of `self`.

This assumes that the algebra generators of A provided by its `algebra_generators` method are homogeneous.

EXAMPLES:

```
sage: A = Algebras(QQ).WithBasis().Filtered().example()
sage: grA = A.graded_algebra()
sage: grA.algebra_generators()
Finite family {'y': bar(U['y']), 'x': bar(U['x']), 'z': bar(U['z'])}
```

degree_on_basis (x)

Return the degree of the basis element indexed by x .

EXAMPLES:

```
sage: A = Algebras(QQ).WithBasis().Filtered().example()
sage: grA = A.graded_algebra()
sage: all(A.degree_on_basis(x) == grA.degree_on_basis(x)
....:      for g in grA.algebra_generators() for x in g.support())
True
```

gen (*args, **kws)

Return a generator of `self`.

EXAMPLES:

```
sage: A = Algebras(QQ).WithBasis().Filtered().example()
sage: grA = A.graded_algebra()
sage: grA.gen('x')
bar(U['x'])
```

one_basis ()

Return the basis index of the element 1 of $\text{gr } A$.

This assumes that the unity 1 of A belongs to F_0 .

EXAMPLES:

```
sage: A = Algebras(QQ).WithBasis().Filtered().example()
sage: grA = A.graded_algebra()
sage: grA.one_basis()
1
```

product_on_basis (x, y)

Return the product on basis elements given by the indices x and y .

EXAMPLES:

```

sage: A = Algebras(QQ).WithBasis().Filtered().example()
sage: grA = A.graded_algebra()
sage: G = grA.algebra_generators()
sage: x,y,z = G['x'], G['y'], G['z']
sage: x * y # indirect doctest
bar(U['x']*U['y'])
sage: y * x
bar(U['x']*U['y'])
sage: z * y * x
bar(U['x']*U['y']*U['z'])

```

5.2 Commutative Differential Graded Algebras

An algebra is said to be *graded commutative* if it is endowed with a grading and its multiplication satisfies the Koszul sign convention: $yx = (-1)^{ij}xy$ if x and y are homogeneous of degrees i and j , respectively. Thus the multiplication is anticommutative for odd degree elements, commutative otherwise. *Commutative differential graded algebras* are graded commutative algebras endowed with a graded differential of degree 1. These algebras can be graded over the integers or they can be multi-graded (i.e., graded over a finite rank free abelian group \mathbb{Z}^n); if multi-graded, the total degree is used in the Koszul sign convention, and the differential must have total degree 1.

EXAMPLES:

All of these algebras may be constructed with the function `GradedCommutativeAlgebra()`. For most users, that will be the main function of interest. See its documentation for many more examples.

We start by constructing some graded commutative algebras. Generators have degree 1 by default:

```

sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ)
sage: x.degree()
1
sage: x^2
0
sage: y*x
-x*y
sage: B.<a,b> = GradedCommutativeAlgebra(QQ, degrees = (2,3))
sage: a.degree()
2
sage: b.degree()
3

```

Once we have defined a graded commutative algebra, it is easy to define a differential on it using the `GCAgebra.cdg_algebra()` method:

```

sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(1,1,2))
sage: B = A.cdg_algebra({x: x*y, y: -x*y})
sage: B
Commutative Differential Graded Algebra with generators ('x', 'y', 'z') in degrees_
↪ (1, 1, 2) over Rational Field with differential:
  x --> x*y
  y --> -x*y
  z --> 0
sage: B.cohomology(3)
Free module generated by {[x*z + y*z]} over Rational Field
sage: B.cohomology(4)
Free module generated by {[z^2]} over Rational Field

```

We can also compute algebra generators for the cohomology in a range of degrees, and in this case we compute up to degree 10:

```
sage: B.cohomology_generators(10)
{1: [x + y], 2: [z]}
```

AUTHORS:

- Miguel Marco, John Palmieri (2014-07): initial version

class sage.algebras.commutative_dga. **CohomologyClass** (*x*)

Bases: sage.structure.sage_object.SageObject

A class for representing cohomology classes.

This just has `_repr_` and `_latex_` methods which put brackets around the object's name.

EXAMPLES:

```
sage: from sage.algebras.commutative_dga import CohomologyClass
sage: CohomologyClass(3)
[3]
sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees = (2,3,3,1))
sage: CohomologyClass(x^2+2*y*z)
[2*y*z + x^2]
```

representative ()

Return the representative of `self`.

EXAMPLES:

```
sage: from sage.algebras.commutative_dga import CohomologyClass
sage: x = CohomologyClass(sin)
sage: x.representative() == sin
True
```

class sage.algebras.commutative_dga. **Differential** (*A*, *im_gens*)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.categories.morphism.Morphism

Differential of a commutative graded algebra.

INPUT:

- *A* – algebra where the differential is defined
- *im_gens* – tuple containing the image of each generator

EXAMPLES:

```
sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(1,1,2,3))
sage: B = A.cdg_algebra({x: x*y, y: -x*y, z: t})
sage: B
Commutative Differential Graded Algebra with generators ('x', 'y', 'z', 't') in_
→degrees (1, 1, 2, 3) over Rational Field with differential:
  x --> x*y
  y --> -x*y
  z --> t
  t --> 0
sage: B.differential()(x)
x*y
```

coboundaries (*n*)

The n -th coboundary group of the algebra.

This is a vector space over the base field F , and it is returned as a subspace of the vector space F^d , where the n -th homogeneous component has dimension d .

INPUT:

- n – degree

EXAMPLES:

```
sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(1,1,2))
sage: d = A.differential({z: x*z})
sage: d.coboundaries(2)
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
sage: d.coboundaries(3)
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[0 1]
```

cocycles (*n*)

The n -th cocycle group of the algebra.

This is a vector space over the base field F , and it is returned as a subspace of the vector space F^d , where the n -th homogeneous component has dimension d .

INPUT:

- n – degree

EXAMPLES:

```
sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(1,1,2))
sage: d = A.differential({z: x*z})
sage: d.cocycles(2)
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 0]
```

cohomology (*n*)

The n -th cohomology group of `self`.

This is a vector space over the base ring, defined as the quotient cocycles/coboundaries. The elements of the quotient are lifted to the vector space of cocycles, and this is described in terms of those lifts.

INPUT:

- n – degree

See also:

`cohomology_raw()`

EXAMPLES:

```
sage: A.<a,b,c,d,e> = GradedCommutativeAlgebra(QQ, degrees=(1,1,1,1,1))
sage: d = A.differential({d: a*b, e: b*c})
sage: d.cohomology(2)
Free module generated by {[c*e], [c*d - a*e], [b*e], [b*d], [a*d], [a*c]}
over Rational Field
```

Compare to `cohomology_raw()` :

```
sage: d.cohomology_raw(2)
Vector space quotient V/W of dimension 6 over Rational Field where
V: Vector space of degree 10 and dimension 8 over Rational Field
Basis matrix:
[ 0  1  0  0  0  0  0  0  0  0]
[ 0  0  1  0  0  0 -1  0  0  0]
[ 0  0  0  1  0  0  0  0  0  0]
[ 0  0  0  0  1  0  0  0  0  0]
[ 0  0  0  0  0  1  0  0  0  0]
[ 0  0  0  0  0  0  0  1  0  0]
[ 0  0  0  0  0  0  0  0  1  0]
[ 0  0  0  0  0  0  0  0  0  1]
W: Vector space of degree 10 and dimension 2 over Rational Field
Basis matrix:
[0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1]
```

cohomology_raw (*n*)

The *n* -th cohomology group of self .

This is a vector space over the base ring, and it is returned as the quotient cocycles/coboundaries.

INPUT:

- *n* – degree

See also:

`cohomology()`

EXAMPLES:

```
sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(2,3,2,4))
sage: d = A.differential({t: x*y, x: y, z: y})
sage: d.cohomology_raw(4)
Vector space quotient V/W of dimension 2 over Rational Field where
V: Vector space of degree 4 and dimension 2 over Rational Field
Basis matrix:
[ 1  0  0 -1/2]
[ 0  1 -2  1]
W: Vector space of degree 4 and dimension 0 over Rational Field
Basis matrix:
[]
```

Compare to `cohomology()` :

```
sage: d.cohomology(4)
Free module generated by {[-1/2*x^2 + t], [x^2 - 2*x*z + z^2]} over Rational_
↪Field
```

differential_matrix (*n*)

The matrix that gives the differential in degree *n* .

INPUT:

- *n* – degree

EXAMPLES:

```

sage: A.<x,y,z,t> = GradedCommutativeAlgebra(GF(5), degrees=(2, 3, 2, 4))
sage: d = A.differential({t: x*y, x: y, z: y})
sage: d.differential_matrix(4)
[0 1]
[2 0]
[1 1]
[0 2]
sage: A.inject_variables()
Defining x, y, z, t
sage: d(t)
x*y
sage: d(z^2)
2*y*z
sage: d(x*z)
x*y + y*z
sage: d(x^2)
2*x*y

```

class `sage.algebras.commutative_dga.DifferentialGCAAlgebra (A, differential)`

Bases: `sage.algebras.commutative_dga.GCAAlgebra`

A commutative differential graded algebra.

INPUT:

- `A` – a graded commutative algebra; that is, an instance of `GCAAlgebra`
- `differential` – a differential

As described in the module-level documentation, these are graded algebras for which oddly graded elements anticommute and evenly graded elements commute, and on which there is a graded differential of degree 1.

These algebras should be graded over the integers; multi-graded algebras should be constructed using `DifferentialGCAAlgebra_multigraded` instead.

Note that a natural way to construct these is to use the `GradedCommutativeAlgebra()` function and the `GCAAlgebra.cdg_algebra()` method.

EXAMPLES:

```

sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(3, 2, 2, 3))
sage: A.cdg_algebra({x: y*z})
Commutative Differential Graded Algebra with generators ('x', 'y', 'z', 't') in
→degrees (3, 2, 2, 3) over Rational Field with differential:
x --> y*z
y --> 0
z --> 0
t --> 0

```

Alternatively, starting with `GradedCommutativeAlgebra()` :

```

sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(3, 2, 2, 3))
sage: A.cdg_algebra(differential={x: y*z})
Commutative Differential Graded Algebra with generators ('x', 'y', 'z', 't') in
→degrees (3, 2, 2, 3) over Rational Field with differential:
x --> y*z
y --> 0
z --> 0
t --> 0

```

See the function `GradedCommutativeAlgebra()` for more examples.

class Element (*A, rep*)

Bases: *sage.algebras.commutative_dga.GCAlgebra.Element*

Initialize self.

INPUT:

- *parent* – the graded commutative algebra in which this element lies, viewed as a quotient R/I
- *rep* – a representative of the element in R ; this is used as the internal representation of the element

EXAMPLES:

```
sage: B.<x,y> = GradedCommutativeAlgebra(QQ, degrees=(2, 2))
sage: a = B({(1,1): -3, (2,5): 1/2})
sage: a
1/2*x^2*y^5 - 3*x*y
sage: TestSuite(a).run()

sage: b = x^2*y^3+2
sage: b
x^2*y^3 + 2
```

differential ()

The differential on this element.

EXAMPLES:

```
sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees = (2, 3, 2, 4))
sage: B = A.cdg_algebra({t: x*y, x: y, z: y})
sage: B.inject_variables()
Defining x, y, z, t
sage: x.differential()
y
sage: (-1/2 * x^2 + t).differential()
0
```

is_coboundary ()

Return True if self is a coboundary and False otherwise.

This raises an error if the element is not homogeneous.

EXAMPLES:

```
sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=(1,2,2))
sage: B = A.cdg_algebra(differential={b: a*c})
sage: x,y,z = B.gens()
sage: x.is_coboundary()
False
sage: (x*z).is_coboundary()
True
sage: (x*z+x*y).is_coboundary()
False
sage: (x*z+y**2).is_coboundary()
Traceback (most recent call last):
...
ValueError: This element is not homogeneous
```

is_cohomologous_to (*other*)

Return True if self is cohomologous to other and False otherwise.

INPUT:

- other – another element of this algebra

EXAMPLES:

```
sage: A.<a,b,c,d> = GradedCommutativeAlgebra(QQ, degrees=(1,1,1,1))
sage: B = A.cdg_algebra(differential={a:b*c-c*d})
sage: w, x, y, z = B.gens()
sage: (x*y).is_cohomologous_to(y*z)
True
sage: (x*y).is_cohomologous_to(x*z)
False
sage: (x*y).is_cohomologous_to(x*y)
True
```

Two elements whose difference is not homogeneous are cohomologous if and only if they are both coboundaries:

```
sage: w.is_cohomologous_to(y*z)
False
sage: (x*y-y*z).is_cohomologous_to(x*y*z)
True
sage: (x*y*z).is_cohomologous_to(0) # make sure 0 works
True
```

DifferentialGCAgebra. **coboundaries** (*n*)

The *n*-th coboundary group of the algebra.

This is a vector space over the base field F , and it is returned as a subspace of the vector space F^d , where the *n*-th homogeneous component has dimension d .

INPUT:

- n* – degree

EXAMPLES:

```
sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(1,1,2))
sage: B = A.cdg_algebra(differential={z: x*z})
sage: B.coboundaries(2)
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
sage: B.coboundaries(3)
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[0 1]
sage: B.basis(3)
[y*z, x*z]
```

DifferentialGCAgebra. **cocycles** (*n*)

The *n*-th cocycle group of the algebra.

This is a vector space over the base field F , and it is returned as a subspace of the vector space F^d , where the *n*-th homogeneous component has dimension d .

INPUT:

- n* – degree

EXAMPLES:

```

sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(1,1,2))
sage: B = A.cdg_algebra(differential={z: x*z})
sage: B.cocycles(2)
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 0]
sage: B.basis(2)
[x*y, z]

```

DifferentialGCAAlgebra. **cohomology** (*n*)

The *n*-th cohomology group of self.

This is a vector space over the base ring, defined as the quotient cocycles/coboundaries. The elements of the quotient are lifted to the vector space of cocycles, and this is described in terms of those lifts.

INPUT:

• *n* – degree

EXAMPLES:

```

sage: A.<a,b,c,d,e> = GradedCommutativeAlgebra(QQ, degrees=(1,1,1,1,1))
sage: B = A.cdg_algebra({d: a*b, e: b*c})
sage: B.cohomology(2)
Free module generated by {[c*e], [c*d - a*e], [b*e], [b*d], [a*d], [a*c]}_
↳ over Rational Field

```

Compare to `cohomology_raw()` :

```

sage: B.cohomology_raw(2)
Vector space quotient V/W of dimension 6 over Rational Field where
V: Vector space of degree 10 and dimension 8 over Rational Field
Basis matrix:
[ 0  1  0  0  0  0  0  0  0  0  0]
[ 0  0  1  0  0  0 -1  0  0  0  0]
[ 0  0  0  1  0  0  0  0  0  0  0]
[ 0  0  0  0  1  0  0  0  0  0  0]
[ 0  0  0  0  0  1  0  0  0  0  0]
[ 0  0  0  0  0  0  1  0  0  0  0]
[ 0  0  0  0  0  0  0  1  0  0  0]
[ 0  0  0  0  0  0  0  0  1  0  0]
[ 0  0  0  0  0  0  0  0  0  1  0]
W: Vector space of degree 10 and dimension 2 over Rational Field
Basis matrix:
[0 0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 1]

```

DifferentialGCAAlgebra. **cohomology_generators** (*max_degree*)

Return lifts of algebra generators for cohomology in degrees at most *max_degree*.

INPUT:

• *max_degree* – integer

OUTPUT:

A dictionary keyed by degree, where the corresponding value is a list of cohomology generators in that degree. Actually, the elements are lifts of cohomology generators, which means that they lie in this differential graded algebra. It also means that they are only well-defined up to cohomology, not on the nose.

ALGORITHM:

Use induction on degree, so assume we know what happens in degrees less than n . Compute the cocycles Z in degree n . Form a subspace W of this, spanned by the cocycles generated by the lower degree generators, along with the coboundaries in degree n . Find a basis for the complement of W in Z : these represent cohomology generators.

EXAMPLES:

```
sage: A.<a,x,y> = GradedCommutativeAlgebra(QQ, degrees=(1,2,2))
sage: B = A.cdg_algebra(differential={y: a*x})
sage: B.cohomology_generators(3)
{1: [a], 2: [x], 3: [a*y]}
```

The previous example has infinitely generated cohomology: ay^n is a cohomology generator for each n :

```
sage: B.cohomology_generators(10)
{1: [a], 2: [x], 3: [a*y], 5: [a*y^2], 7: [a*y^3], 9: [a*y^4]}
```

In contrast, the corresponding algebra in characteristic p has finitely generated cohomology:

```
sage: A3.<a,x,y> = GradedCommutativeAlgebra(GF(3), degrees=(1,2,2))
sage: B3 = A3.cdg_algebra(differential={y: a*x})
sage: B3.cohomology_generators(20)
{1: [a], 2: [x], 3: [a*y], 5: [a*y^2], 6: [y^3]}
```

This method works with both singly graded and multi-graded algebras:

```
sage: Cs.<a,b,c,d> = GradedCommutativeAlgebra(GF(2), degrees=(1,2,2,3))
sage: Ds = Cs.cdg_algebra({a:c, b:d})
sage: Ds.cohomology_generators(10)
{2: [a^2], 4: [b^2]}

sage: Cm.<a,b,c,d> = GradedCommutativeAlgebra(GF(2), degrees=((1,0), (1,1), (0,2), (0,3)))
sage: Dm = Cm.cdg_algebra({a:c, b:d})
sage: Dm.cohomology_generators(10)
{2: [a^2], 4: [b^2]}
```

DifferentialGCAAlgebra. **cohomology_raw** (n)

The n -th cohomology group of self.

This is a vector space over the base ring, and it is returned as the quotient cocycles/coboundaries.

INPUT:

- n – degree

EXAMPLES:

```
sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees = (2,3,2,4))
sage: B = A.cdg_algebra({t: x*y, x: y, z: y})
sage: B.cohomology_raw(4)
Vector space quotient V/W of dimension 2 over Rational Field where
V: Vector space of degree 4 and dimension 2 over Rational Field
Basis matrix:
[ 1 0 0 -1/2]
[ 0 1 -2 1]
W: Vector space of degree 4 and dimension 0 over Rational Field
Basis matrix:
[]
```

Compare to `cohomology()` :

```
sage: B.cohomology(4)
Free module generated by  $\{-1/2x^2 + t\}, [x^2 - 2xz + z^2]$  over Rational_
↪Field
```

`DifferentialGCAgebra.differential (x=None)`

The differential of `self`.

This returns a map, and so it may be evaluated on elements of this algebra.

EXAMPLES:

```
sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(2,1,1))
sage: B = A.cdg_algebra({y:y*z, z: y*z})
sage: d = B.differential(); d
Differential of Commutative Differential Graded Algebra with generators ('x',
↪'y', 'z') in degrees (2, 1, 1) over Rational Field
Defn: x --> 0
      y --> y*z
      z --> y*z
sage: d(y)
y*z
```

`DifferentialGCAgebra.graded_commutative_algebra ()`

Return the base graded commutative algebra of `self`.

EXAMPLES:

```
sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(3, 2, 2, 3))
sage: D = A.cdg_algebra({x: y*z})
sage: D.graded_commutative_algebra() == A
True
```

`DifferentialGCAgebra.quotient (I, check=True)`

Create the quotient of this algebra by a two-sided ideal `I`.

INPUT:

- `I` – a two-sided homogeneous ideal of this algebra
- `check` – (default: `True`) if `True`, check whether `I` is generated by homogeneous elements

EXAMPLES:

```
sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(2,1,1))
sage: B = A.cdg_algebra({y:y*z, z: y*z})
sage: B.inject_variables()
Defining x, y, z
sage: I = B.ideal([x*y])
sage: C = B.quotient(I)
sage: (x*y).differential()
x*y*z
sage: C((x*y).differential())
0
sage: C(x*y)
0
```

It is checked that the differential maps the ideal into itself, to make sure that the quotient inherits a differential structure:

```

sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(2,2,1))
sage: B = A.cdg_algebra({z:y})
sage: B.quotient(B.ideal(y*z))
Traceback (most recent call last):
...
ValueError: The differential does not preserve the ideal
sage: B.quotient(B.ideal(z))
Traceback (most recent call last):
...
ValueError: The differential does not preserve the ideal

```

class sage.algebras.commutative_dga. **DifferentialGCAgebra_multigraded** (*A*, *differential*)
Bases: *sage.algebras.commutative_dga.DifferentialGCAgebra*, *sage.algebras.commutative_dga.GCAgebra_multigraded*

A commutative differential multi-graded algebras.

INPUT:

- *A* – a commutative multi-graded algebra
- *differential* – a differential

EXAMPLES:

```

sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1,0), (0, 1), (0,2)))
sage: B = A.cdg_algebra(differential={a: c})
sage: B.basis((1,0))
[a]
sage: B.basis(1, total=True)
[b, a]
sage: B.cohomology((1, 0))
Free module generated by {} over Rational Field
sage: B.cohomology(1, total=True)
Free module generated by {[b]} over Rational Field

```

class **Element** (*A*, *rep*)
Bases: *sage.algebras.commutative_dga.GCAgebra_multigraded.Element*, *sage.algebras.commutative_dga.DifferentialGCAgebra.Element*

Element class of a commutative differential multi-graded algebra.

DifferentialGCAgebra_multigraded.coboundaries (*n*, *total=False*)

The *n*-th coboundary group of the algebra.

This is a vector space over the base field F , and it is returned as a subspace of the vector space F^d , where the *n*-th homogeneous component has dimension *d*.

INPUT:

- *n* – degree
- *total* (default `False`) – if `True`, return the coboundaries in total degree *n*

If *n* is an integer rather than a multi-index, then the total degree is used in that case as well.

EXAMPLES:

```

sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1,0), (0, 1), (0,2)))
sage: B = A.cdg_algebra(differential={a: c})

```

```

sage: B.coboundaries((0,2))
Vector space of degree 1 and dimension 1 over Rational Field
Basis matrix:
[1]
sage: B.coboundaries(2)
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[0 1]

```

DifferentialGCAgebra_multigraded. **cocycles** (*n*, *total=False*)

The *n*-th cocycle group of the algebra.

This is a vector space over the base field F , and it is returned as a subspace of the vector space F^d , where the *n*-th homogeneous component has dimension d .

INPUT:

- *n* – degree
- *total* – (default: False) if True, return the cocycles in total degree *n*

If *n* is an integer rather than a multi-index, then the total degree is used in that case as well.

EXAMPLES:

```

sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1,0), (0, 1), (0,2)))
sage: B = A.cdg_algebra(differential={a: c})
sage: B.cocycles((0,1))
Vector space of degree 1 and dimension 1 over Rational Field
Basis matrix:
[1]
sage: B.cocycles((0,1), total=True)
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 0]

```

DifferentialGCAgebra_multigraded. **cohomology** (*n*, *total=False*)

The *n*-th cohomology group of the algebra.

This is a vector space over the base ring, defined as the quotient cocycles/coboundaries. The elements of the quotient are lifted to the vector space of cocycles, and this is described in terms of those lifts.

Compare to [cohomology_raw\(\)](#).

INPUT:

- *n* – degree
- *total* – (default: False) if True, return the cohomology in total degree *n*

If *n* is an integer rather than a multi-index, then the total degree is used in that case as well.

EXAMPLES:

```

sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1,0), (0, 1), (0,2)))
sage: B = A.cdg_algebra(differential={a: c})
sage: B.cohomology((0,2))
Free module generated by {} over Rational Field

sage: B.cohomology(1)
Free module generated by {[b]} over Rational Field

```

DifferentialGCAgebra_multigraded. **cohomology_raw** (*n*, *total=False*)

The *n*-th cohomology group of the algebra.

This is a vector space over the base ring, and it is returned as the quotient cocycles/coboundaries.

Compare to `cohomology()`.

INPUT:

- *n* – degree
- *total* – (default: `False`) if `True`, return the cohomology in total degree *n*

If *n* is an integer rather than a multi-index, then the total degree is used in that case as well.

EXAMPLES:

```
sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1,0), (0, 1), (0,2)))
sage: B = A.cdg_algebra(differential={a: c})
sage: B.cohomology_raw((0,2))
Vector space quotient V/W of dimension 0 over Rational Field where
V: Vector space of degree 1 and dimension 1 over Rational Field
Basis matrix:
[1]
W: Vector space of degree 1 and dimension 1 over Rational Field
Basis matrix:
[1]

sage: B.cohomology_raw(1)
Vector space quotient V/W of dimension 1 over Rational Field where
V: Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 0]
W: Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
```

class sage.algebras.commutative_dga. **Differential_multigraded** (*A*, *im_gens*)

Bases: `sage.algebras.commutative_dga.Differential`

Differential of a commutative multi-graded algebra.

coboundaries (*n*, *total=False*)

The *n*-th coboundary group of the algebra.

This is a vector space over the base field F , and it is returned as a subspace of the vector space F^d , where the *n*-th homogeneous component has dimension d .

INPUT:

- *n* – degree
- *total* (default `False`) – if `True`, return the coboundaries in total degree *n*

If *n* is an integer rather than a multi-index, then the total degree is used in that case as well.

EXAMPLES:

```
sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1,0), (0, 1), (0,2)))
sage: d = A.differential({a: c})
sage: d.coboundaries((0,2))
Vector space of degree 1 and dimension 1 over Rational Field
Basis matrix:
[1]
```



```

sage: d.coboundaries(2)
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[0 1]

```

cocycles (*n*, *total=False*)

The *n*-th cocycle group of the algebra.

This is a vector space over the base field F , and it is returned as a subspace of the vector space F^d , where the *n*-th homogeneous component has dimension d .

INPUT:

- *n* – degree
- *total* – (default: `False`) if `True`, return the cocycles in total degree *n*

If *n* is an integer rather than a multi-index, then the total degree is used in that case as well.

EXAMPLES:

```

sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1,0), (0, 1), (0,2)))
sage: d = A.differential({a: c})
sage: d.cocycles((0,1))
Vector space of degree 1 and dimension 1 over Rational Field
Basis matrix:
[1]
sage: d.cocycles((0,1), total=True)
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 0]

```

cohomology (*n*, *total=False*)

The *n*-th cohomology group of the algebra.

This is a vector space over the base ring, defined as the quotient cocycles/coboundaries. The elements of the quotient are lifted to the vector space of cocycles, and this is described in terms of those lifts.

INPUT:

- *n* – degree
- *total* – (default: `False`) if `True`, return the cohomology in total degree *n*

If *n* is an integer rather than a multi-index, then the total degree is used in that case as well.

See also:

`cohomology_raw()`

EXAMPLES:

```

sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1,0), (0, 1), (0,2)))
sage: d = A.differential({a: c})
sage: d.cohomology((0,2))
Free module generated by {} over Rational Field

sage: d.cohomology(1)
Free module generated by {[b]} over Rational Field

```

cohomology_raw (*n*, *total=False*)

The *n*-th cohomology group of the algebra.

This is a vector space over the base ring, and it is returned as the quotient cocycles/coboundaries.

INPUT:

- `n` – degree
- `total` – (default: `False`) if `True`, return the cohomology in total degree `n`

If `n` is an integer rather than a multi-index, then the total degree is used in that case as well.

See also:

`cohomology()`

EXAMPLES:

```
sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1,0), (0, 1), (0,2)))
sage: d = A.differential({a: c})
sage: d.cohomology_raw((0,2))
Vector space quotient V/W of dimension 0 over Rational Field where
V: Vector space of degree 1 and dimension 1 over Rational Field
Basis matrix:
[1]
W: Vector space of degree 1 and dimension 1 over Rational Field
Basis matrix:
[1]

sage: d.cohomology_raw(1)
Vector space quotient V/W of dimension 1 over Rational Field where
V: Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 0]
W: Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
```

`differential_matrix_multigraded (n, total=False)`

The matrix that gives the differential in degree `n`.

Todo

Rename this to `differential_matrix` once inheritance, overriding, and cached methods work together better. See [trac ticket #17201](#).

INPUT:

- `n` – degree
- `total` – (default: `False`) if `True`, return the matrix corresponding to total degree `n`

If `n` is an integer rather than a multi-index, then the total degree is used in that case as well.

EXAMPLES:

```
sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1,0), (0, 1), (0,2)))
sage: d = A.differential({a: c})
sage: d.differential_matrix_multigraded((1,0))
[1]
sage: d.differential_matrix_multigraded(1, total=True)
[0 0]
[0 1]
```

```

sage: d.differential_matrix_multigraded((1,0), total=True)
[0 0]
[0 1]
sage: d.differential_matrix_multigraded(1)
[0 0]
[0 1]

```

class `sage.algebras.commutative_dga.GCAlgebra` (*base*, *R=None*, *I=None*, *names=None*, *degrees=None*)

Bases: `sage.structure.unique_representation.UniqueRepresentation`,
`sage.rings.quotient_ring.QuotientRing_nc`

A graded commutative algebra.

INPUT:

- *base* – the base field
- *names* – (optional) names of the generators: a list of strings or a single string with the names separated by commas. If not specified, the generators are named “x0”, “x1”, ...
- *degrees* – (optional) a tuple or list specifying the degrees of the generators; if omitted, each generator is given degree 1, and if both *names* and *degrees* are omitted, an error is raised.
- *R* (optional, default None) – the ring over which the algebra is defined: if this is specified, the algebra is defined to be R/I .
- *I* (optional, default None) – an ideal in R . It should include, among other relations, the squares of the generators of odd degree

As described in the module-level documentation, these are graded algebras for which oddly graded elements anticommute and evenly graded elements commute.

The arguments *R* and *I* are primarily for use by the `quotient()` method.

These algebras should be graded over the integers; multi-graded algebras should be constructed using `GCAlgebra_multigraded` instead.

EXAMPLES:

```

sage: A.<a,b> = GradedCommutativeAlgebra(QQ, degrees = (2,3))
sage: a.degree()
2
sage: B = A.quotient(A.ideal(a**2*b))
sage: B
Graded Commutative Algebra with generators ('a', 'b') in degrees (2, 3) with
↪relations [a^2*b] over Rational Field
sage: A.basis(7)
[a^2*b]
sage: B.basis(7)
[]

```

Note that the function `GradedCommutativeAlgebra()` can also be used to construct these algebras.

class `Element` (*A*, *rep*)

Bases: `sage.rings.quotient_ring_element.QuotientRingElement`

An element of a graded commutative algebra.

basis_coefficients (*total=False*)

Return the coefficients of this homogeneous element with respect to the basis in its degree.

For example, if this is the sum of the 0th and 2nd basis elements, return the list `[1, 0, 1]`.

Raise an error if the element is not homogeneous.

INPUT:

- `total` – boolean (default `False`); this is only used in the multi-graded case, in which case if `True`, it returns the coefficients with respect to the basis for the total degree of this element

OUTPUT:

A list of elements of the base field.

EXAMPLES:

```
sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(1, 2, 2, 3))
sage: A.basis(3)
[t, x*z, x*y]
sage: (t + 3*x*y).basis_coefficients()
[1, 0, 3]
sage: (t + x).basis_coefficients()
Traceback (most recent call last):
...
ValueError: This element is not homogeneous

sage: B.<c,d> = GradedCommutativeAlgebra(QQ, degrees=((2,0), (0,4)))
sage: B.basis(4)
[d, c^2]
sage: (c^2 - 1/2 * d).basis_coefficients(total=True)
[-1/2, 1]
sage: (c^2 - 1/2 * d).basis_coefficients()
Traceback (most recent call last):
...
ValueError: This element is not homogeneous
```

degree (*total=False*)

The degree of this element.

If the element is not homogeneous, this returns the maximum of the degrees of its monomials.

INPUT:

- `total` – ignored, present for compatibility with the multi-graded case

EXAMPLES:

```
sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(2,3,3,1))
sage: e1 = y*z+2*x*t-x^2*y
sage: e1.degree()
7
sage: e1.monomials()
[x^2*y, y*z, x*t]
sage: [i.degree() for i in e1.monomials()]
[7, 6, 3]

sage: A(0).degree()
Traceback (most recent call last):
...
ValueError: The zero element does not have a well-defined degree
```

dict ()

A dictionary that determines the element.

The keys of this dictionary are the tuples of exponents of each monomial, and the values are the corresponding coefficients.

EXAMPLES:

```

sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(1, 2, 2, 3))
sage: dic = (x*y - 5*y*z + 7*x*y^2*z^3*t).dict()
sage: sorted(dic.items())
[((0, 1, 1, 0), -5), ((1, 1, 0, 0), 1), ((1, 2, 3, 1), 7)]

```

is_homogeneous (*total=False*)

Return True if self is homogeneous and False otherwise.

INPUT:

- total – boolean (default False); only used in the multi-graded case, in which case if True, check to see if self is homogeneous with respect to total degree

EXAMPLES:

```

sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(2,3,3,1))
sage: e1 = y*z + 2*x*t - x^2*y
sage: e1.degree()
7
sage: e1.monomials()
[x^2*y, y*z, x*t]
sage: [i.degree() for i in e1.monomials()]
[7, 6, 3]
sage: e1.is_homogeneous()
False
sage: em = x^3 - 5*y*z + 3/2*x*z*t
sage: em.is_homogeneous()
True
sage: em.monomials()
[x^3, y*z, x*z*t]
sage: [i.degree() for i in em.monomials()]
[6, 6, 6]

```

The element 0 is homogeneous, even though it doesn't have a well-defined degree:

```

sage: A(0).is_homogeneous()
True

```

A multi-graded example:

```

sage: B.<c,d> = GradedCommutativeAlgebra(QQ, degrees=((2,0), (0,4)))
sage: (c^2 - 1/2 * d).is_homogeneous()
False
sage: (c^2 - 1/2 * d).is_homogeneous(total=True)
True

```

GCAAlgebra.basis (*n*)

Return a basis of the *n*-th homogeneous component of self.

EXAMPLES:

```

sage: A.<x,y,z,t> = GradedCommutativeAlgebra(QQ, degrees=(1, 2, 2, 3))
sage: A.basis(2)
[z, y]
sage: A.basis(3)
[t, x*z, x*y]
sage: A.basis(4)
[x*t, z^2, y*z, y^2]
sage: A.basis(5)
[z*t, y*t, x*z^2, x*y*z, x*y^2]

```

```
sage: A.basis(6)
[x*z*t, x*y*t, z^3, y*z^2, y^2*z, y^3]
```

GCAlgebra. **cdg_algebra** (*differential*)

Construct a differential graded commutative algebra from *self* by specifying a differential.

INPUT:

- *differential* – a dictionary defining a differential or a map defining a valid differential

The keys of the dictionary are generators of the algebra, and the associated values are their targets under the differential. Any generators which are not specified are assumed to have zero differential. Alternatively, the differential can be defined using the *differential()* method; see below for an example.

See also:

differential()

EXAMPLES:

```
sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=(1,1,1))
sage: B = A.cdg_algebra({a: b*c, b: a*c})
sage: B
Commutative Differential Graded Algebra with generators ('a', 'b', 'c') in
↳degrees (1, 1, 1) over Rational Field with differential:
  a --> b*c
  b --> a*c
  c --> 0
```

Note that *differential* can also be a map:

```
sage: d = A.differential({a: b*c, b: a*c})
sage: d
Differential of Graded Commutative Algebra with generators ('a', 'b', 'c') in
↳degrees (1, 1, 1) over Rational Field
Defn: a --> b*c
      b --> a*c
      c --> 0
sage: A.cdg_algebra(d) is B
True
```

GCAlgebra. **differential** (*diff*)

Construct a differential on *self*.

INPUT:

- *diff* – a dictionary defining a differential

The keys of the dictionary are generators of the algebra, and the associated values are their targets under the differential. Any generators which are not specified are assumed to have zero differential.

EXAMPLES:

```
sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(2,1,1))
sage: A.differential({y:y*z, z: y*z})
Differential of Graded Commutative Algebra with generators ('x', 'y', 'z') in
↳degrees (2, 1, 1) over Rational Field
Defn: x --> 0
      y --> y*z
      z --> y*z
sage: B.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=(1,2,2))
```

```
sage: d = B.differential({b:a*c, c:a*c})
sage: d(b*c)
a*b*c + a*c^2
```

`GCAlgebra.quotient(I, check=True)`

Create the quotient of this algebra by a two-sided ideal I .

INPUT:

- I – a two-sided homogeneous ideal of this algebra
- `check` – (default: `True`) if `True`, check whether I is generated by homogeneous elements

EXAMPLES:

```
sage: A.<x,y,z,t> = GradedCommutativeAlgebra(GF(5), degrees=(2, 3, 2, 4))
sage: I = A.ideal([x*t+y^2, x*z - t])
sage: B = A.quotient(I)
sage: B
Graded Commutative Algebra with generators ('x', 'y', 'z', 't') in degrees
↳ (2, 3, 2, 4) with relations [x*t, x*z - t] over Finite Field of size 5
sage: B(x*t)
0
sage: B(x*z)
t
sage: A.basis(7)
[y*t, y*z^2, x*y*z, x^2*y]
sage: B.basis(7)
[y*t, y*z^2, x^2*y]
```

```
class sage.algebras.commutative_dga.GCAlgebra_multigraded ( base,          degrees,
                                                             names=None, R=None,
                                                             I=None)
```

Bases: `sage.algebras.commutative_dga.GCAlgebra`

A multi-graded commutative algebra.

INPUT:

- `base` – the base field
- `degrees` – a tuple or list specifying the degrees of the generators
- `names` – (optional) names of the generators: a list of strings or a single string with the names separated by commas; if not specified, the generators are named x_0, x_1, \dots
- `R` – (optional) the ring over which the algebra is defined
- `I` – (optional) an ideal in R ; it should include, among other relations, the squares of the generators of odd degree

When defining such an algebra, each entry of `degrees` should be a list, tuple, or element of an additive (free) abelian group. Regardless of how the user specifies the degrees, Sage converts them to group elements.

The arguments `R` and `I` are primarily for use by the `GCAlgebra.quotient()` method.

EXAMPLES:

```
sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1,0), (0,1), (1,1)))
sage: A
Graded Commutative Algebra with generators ('a', 'b', 'c') in degrees ((1, 0),
↳ (0, 1), (1, 1)) over Rational Field
sage: a**2
```

```

0
sage: c.degree(total=True)
2
sage: c**2
c^2
sage: c.degree()
(1, 1)

```

Although the degree of `c` was defined using a Python tuple, it is returned as an element of an additive abelian group, and so it can be manipulated via arithmetic operations:

```

sage: type(c.degree())
<class 'sage.groups.additive_abelian.additive_abelian_group.AdditiveAbelianGroup_
↳fixed_gens_with_category.element_class'>
sage: 2 * c.degree()
(2, 2)
sage: (a*b).degree() == a.degree() + b.degree()
True

```

The `basis()` method and the `Element.degree()` method both accept the boolean keyword `total`. If `True`, use the total degree:

```

sage: A.basis(2, total=True)
[a*b, c]
sage: c.degree(total=True)
2

```

class `Element` (`A, rep`)

Bases: `sage.algebras.commutative_dga.GCAAlgebra.Element`

Initialize self.

INPUT:

- `parent` – the graded commutative algebra in which this element lies, viewed as a quotient R/I
- `rep` – a representative of the element in R ; this is used as the internal representation of the element

EXAMPLES:

```

sage: B.<x,y> = GradedCommutativeAlgebra(QQ, degrees=(2, 2))
sage: a = B({(1,1): -3, (2,5): 1/2})
sage: a
1/2*x^2*y^5 - 3*x*y
sage: TestSuite(a).run()

sage: b = x^2*y^3+2
sage: b
x^2*y^3 + 2

```

degree (`total=False`)

Return the degree of this element.

INPUT:

- `total` – if `True`, return the total degree, an integer; otherwise, return the degree as an element of an additive free abelian group

If not requesting the total degree, raise an error if the element is not homogeneous.

EXAMPLES:


```

sage: A.<a,b,c> = GradedCommutativeAlgebra(GF(2), degrees=((1,0), (0,1),
↪(1,1)))
sage: (a**2*b).degree()
(2, 1)
sage: (a**2*b).degree(total=True)
3
sage: (a**2*b + c).degree()
Traceback (most recent call last):
...
ValueError: This element is not homogeneous
sage: (a**2*b + c).degree(total=True)
3
sage: A(0).degree()
Traceback (most recent call last):
...
ValueError: The zero element does not have a well-defined degree

```

`GCAlgebra_multigraded.basis (n, total=False)`

Basis in degree n .

- n – degree or integer
- `total` (optional, default `False`) – if `True`, return the basis in total degree n .

If n is an integer rather than a multi-index, then the total degree is used in that case as well.

EXAMPLES:

```

sage: A.<a,b,c> = GradedCommutativeAlgebra(GF(2), degrees=((1,0), (0,1),
↪(1,1)))
sage: A.basis((1,1))
[c, a*b]
sage: A.basis(2, total=True)
[c, b^2, a*b, a^2]

```

Since 2 is not a multi-index, we don't need to specify `total=True`:

```

sage: A.basis(2)
[c, b^2, a*b, a^2]

```

If `total==True`, then n can still be a tuple, list, etc., and its total degree is used instead:

```

sage: A.basis((1,1), total=True)
[c, b^2, a*b, a^2]

```

`GCAlgebra_multigraded.cdg_algebra (differential)`

Construct a differential graded commutative algebra from `self` by specifying a differential.

INPUT:

- `differential` – a dictionary defining a differential or a map defining a valid differential

The keys of the dictionary are generators of the algebra, and the associated values are their targets under the differential. Any generators which are not specified are assumed to have zero differential. Alternatively, the differential can be defined using the `differential()` method; see below for an example.

See also:

`differential()`

EXAMPLES:

```

sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1,0), (0, 1), (0,2)))
sage: A.cdg_algebra({a: c})
Commutative Differential Graded Algebra with generators ('a', 'b', 'c') in
↳degrees ((1, 0), (0, 1), (0, 2)) over Rational Field with differential:
  a --> c
  b --> 0
  c --> 0
sage: d = A.differential({a: c})
sage: A.cdg_algebra(d)
Commutative Differential Graded Algebra with generators ('a', 'b', 'c') in
↳degrees ((1, 0), (0, 1), (0, 2)) over Rational Field with differential:
  a --> c
  b --> 0
  c --> 0

```

GCAlgebra_multigraded. **differential** (*diff*)

Construct a differential on self.

INPUT:

- diff – a dictionary defining a differential

The keys of the dictionary are generators of the algebra, and the associated values are their targets under the differential. Any generators which are not specified are assumed to have zero differential.

EXAMPLES:

```

sage: A.<a,b,c> = GradedCommutativeAlgebra(QQ, degrees=((1,0), (0, 1), (0,2)))
sage: A.differential({a: c})
Differential of Graded Commutative Algebra with generators ('a', 'b', 'c') in
↳degrees ((1, 0), (0, 1), (0, 2)) over Rational Field
Defn: a --> c
      b --> 0
      c --> 0

```

GCAlgebra_multigraded. **quotient** (*I, check=True*)

Create the quotient of this algebra by a two-sided ideal *I*.

INPUT:

- I – a two-sided homogeneous ideal of this algebra
- check – (default: True) if True, check whether *I* is generated by homogeneous elements

EXAMPLES:

```

sage: A.<x,y,z,t> = GradedCommutativeAlgebra(GF(5), degrees=(2, 3, 2, 4))
sage: I = A.ideal([x*t+y^2, x*z - t])
sage: B = A.quotient(I)
sage: B
Graded Commutative Algebra with generators ('x', 'y', 'z', 't') in degrees
↳(2, 3, 2, 4) with relations [x*t, x*z - t] over Finite Field of size 5
sage: B(x*t)
0
sage: B(x*z)
t
sage: A.basis(7)
[y*t, y*z^2, x*y*z, x^2*y]
sage: B.basis(7)
[y*t, y*z^2, x^2*y]

```

```
sage.algebras.commutative_dga. GradedCommutativeAlgebra ( ring,          names=None,
                                                         degrees=None,      rela-
                                                         tions=None)
```

A graded commutative algebra.

INPUT:

There are two ways to call this. The first way defines a free graded commutative algebra:

- `ring` – the base field over which to work
- `names` – names of the generators. You may also use Sage's `A.<x,y,...> = ...` syntax to define the names. If no names are specified, the generators are named `x0`, `x1`, ...
- `degrees` – degrees of the generators; if this is omitted, the degree of each generator is 1, and if both `names` and `degrees` are omitted, an error is raised

Once such an algebra has been defined, one can use its associated methods to take a quotient, impose a differential, etc. See the examples below.

The second way takes a graded commutative algebra and imposes relations:

- `ring` – a graded commutative algebra
- `relations` – a list or tuple of elements of `ring`

EXAMPLES:

Defining a graded commutative algebra:

```
sage: GradedCommutativeAlgebra(QQ, 'x, y, z')
Graded Commutative Algebra with generators ('x', 'y', 'z') in degrees (1, 1, 1)
→over Rational Field
sage: GradedCommutativeAlgebra(QQ, degrees=(2, 3, 4))
Graded Commutative Algebra with generators ('x0', 'x1', 'x2') in degrees (2, 3,
→4) over Rational Field
```

As usual in Sage, the `A.<...>` notation defines both the algebra and the generator names:

```
sage: A.<x,y,z> = GradedCommutativeAlgebra(QQ, degrees=(1, 2, 1))
sage: x^2
0
sage: z*x # Odd classes anticommute.
-x*z
sage: z*y # y is central since it is in degree 2.
y*z
sage: (x*y**3*z).degree()
8
sage: A.basis(3) # basis of homogeneous degree 3 elements
[y*z, x*y]
```

Defining a quotient:

```
sage: I = A.ideal(x*y)
sage: AQ = A.quotient(I)
sage: AQ
Graded Commutative Algebra with generators ('x', 'y', 'z') in degrees (1, 2, 1)
→with relations [x*y] over Rational Field
sage: AQ.basis(3)
[y*z]
```

Note that `AQ` has no specified differential. This is reflected in its print representation: `AQ` is described as a “graded commutative algebra” – the word “differential” is missing. Also, it has no default differential:

```
sage: AQ.differential()
Traceback (most recent call last):
...
TypeError: differential() takes exactly 2 arguments (1 given)
```

Now we add a differential to `AQ`:

```
sage: B = AQ.cdg_algebra({y:y*z})
sage: B
Commutative Differential Graded Algebra with generators ('x', 'y', 'z') in
→degrees (1, 2, 1) with relations [x*y] over Rational Field with differential:
x --> 0
y --> y*z
z --> 0
sage: B.differential()
Differential of Commutative Differential Graded Algebra with generators ('x', 'y
→', 'z') in degrees (1, 2, 1) with relations [x*y] over Rational Field
Defn: x --> 0
      y --> y*z
      z --> 0
sage: B.cohomology(1)
Free module generated by {[z], [x]} over Rational Field
sage: B.cohomology(2)
Free module generated by {[x*z]} over Rational Field
```

We compute algebra generators for cohomology in a range of degrees. This cohomology algebra appears to be finitely generated:

```
sage: B.cohomology_generators(15)
{1: [z, x]}
```

We can construct multi-graded rings as well. We work in characteristic 2 for a change, so the algebras here are honestly commutative:

```
sage: C.<a,b,c,d> = GradedCommutativeAlgebra(GF(2), degrees=((1,0), (1,1), (0,2),
→(0,3)))
sage: D = C.cdg_algebra(differential={a:c, b:d})
sage: D
Commutative Differential Graded Algebra with generators ('a', 'b', 'c', 'd') in
→degrees ((1, 0), (1, 1), (0, 2), (0, 3)) over Finite Field of size 2 with
→differential:
a --> c
b --> d
c --> 0
d --> 0
```

We can examine `D` using both total degrees and multidegrees. Use tuples, lists, vectors, or elements of additive abelian groups to specify degrees:

```
sage: D.basis(3) # basis in total degree 3
[d, a*c, a*b, a^3]
sage: D.basis((1,2)) # basis in degree (1,2)
[a*c]
sage: D.basis([1,2])
[a*c]
```

```
sage: D.basis(vector([1,2]))
[a*c]
sage: G = AdditiveAbelianGroup([0,0]); G
Additive abelian group isomorphic to Z + Z
sage: D.basis(G(vector([1,2])))
[a*c]
```

At this point, a , for example, is an element of C . We can redefine it so that it is instead an element of D in several ways, for instance using `gens()` method:

```
sage: a, b, c, d = D.gens()
sage: a.differential()
c
```

Or the `inject_variables()` method:

```
sage: D.inject_variables()
Defining a, b, c, d
sage: (a*b).differential()
b*c + a*d
sage: (a*b*c**2).degree()
(2, 5)
```

Degrees are returned as elements of additive abelian groups:

```
sage: (a*b*c**2).degree() in G
True
sage: (a*b*c**2).degree(total=True) # total degree
7
sage: D.cohomology(4)
Free module generated by {[b^2], [a^4]} over Finite Field of size 2
sage: D.cohomology((2,2))
Free module generated by {[b^2]} over Finite Field of size 2
```

`sage.algebras.commutative_dga.exterior_algebra_basis (n , $degrees$)`
 Basis of an exterior algebra in degree n , where the generators are in degrees $degrees$.

INPUT:

- n - integer
- $degrees$ - iterable of integers

Return list of lists, each list representing exponents for the corresponding generators. (So each list consists of 0's and 1's.)

EXAMPLES:

```
sage: from sage.algebras.commutative_dga import exterior_algebra_basis
sage: exterior_algebra_basis(1, (1,3,1))
[[0, 0, 1], [1, 0, 0]]
sage: exterior_algebra_basis(4, (1,3,1))
[[0, 1, 1], [1, 1, 0]]
sage: exterior_algebra_basis(10, (1,5,1,1))
[]
```

`sage.algebras.commutative_dga.total_degree (deg)`
 Total degree of deg .

INPUT:

- `deg` - an element of a free abelian group.

In fact, `deg` could be an integer, a Python int, a list, a tuple, a vector, etc. This function returns the sum of the components of `deg`.

EXAMPLES:

```
sage: from sage.algebras.commutative_dga import total_degree
sage: total_degree(12)
12
sage: total_degree(range(5))
10
sage: total_degree(vector(range(5)))
10
sage: G = AdditiveAbelianGroup((0,0))
sage: x = G.gen(0); y = G.gen(1)
sage: 3*x+4*y
(3, 4)
sage: total_degree(3*x+4*y)
7
```

5.3 Q-Systems

AUTHORS:

- Travis Scrimshaw (2013-10-08): Initial version

class `sage.algebras.q_system.QSystem` (*base_ring*, *cartan_type*, *level*)
 Bases: `sage.combinat.free_module.CombinatorialFreeModule`

A Q-system.

Let \mathfrak{g} be a tamely-laced symmetrizable Kac-Moody algebra with index set I over a field k . Follow the presentation given in [HKOTY1999], an unrestricted Q-system is a k -algebra in infinitely many variables $Q_m^{(a)}$, where $a \in I$ and $m \in \mathbf{Z}_{>0}$, that satisfies the relations

$$\left(Q_m^{(a)}\right)^2 = Q_{m+1}^{(a)} Q_{m-1}^{(a)} + \prod_{b \sim a} \prod_{k=0}^{-C_{ab}-1} Q_{\left\lfloor \frac{mC_{ba}-k}{C_{ab}} \right\rfloor}^{(b)},$$

with $Q_0^{(a)} := 1$. Q-systems can be considered as T-systems where we forget the spectral parameter u and for \mathfrak{g} of finite type, have a solution given by the characters of Kirillov-Reshetikhin modules (again without the spectral parameter) for an affine Kac-Moody algebra $\widehat{\mathfrak{g}}$ with \mathfrak{g} as its classical subalgebra. See [KNS2011] for more information.

Q-systems have a natural bases given by polynomials of the fundamental representations $Q_1^{(a)}$, for $a \in I$. As such, we consider the Q-system as generated by $\{Q_1^{(a)}\}_{a \in I}$.

There is also a level ℓ restricted Q-system (with unit boundary condition) given by setting $Q_{d_a \ell}^{(a)} = 1$, where d_a are the entries of the symmetrizing matrix for the dual type of \mathfrak{g} .

EXAMPLES:

We begin by constructing a Q-system and doing some basic computations in type A_4 :

```

sage: Q = QSystem(QQ, ['A', 4])
sage: Q.Q(3, 1)
Q^(3) [1]
sage: Q.Q(1, 2)
Q^(1) [1]^2 - Q^(2) [1]
sage: Q.Q(3, 3)
-Q^(1) [1]*Q^(3) [1] + Q^(1) [1]*Q^(4) [1]^2 + Q^(2) [1]^2
- 2*Q^(2) [1]*Q^(3) [1]*Q^(4) [1] + Q^(3) [1]^3
sage: x = Q.Q(1, 1) + Q.Q(2, 1); x
Q^(1) [1] + Q^(2) [1]
sage: x * x
Q^(1) [1]^2 + 2*Q^(1) [1]*Q^(2) [1] + Q^(2) [1]^2

```

Next we do some basic computations in type C_4 :

```

sage: Q = QSystem(QQ, ['C', 4])
sage: Q.Q(4, 1)
Q^(4) [1]
sage: Q.Q(1, 2)
Q^(1) [1]^2 - Q^(2) [1]
sage: Q.Q(3, 3)
Q^(1) [1]*Q^(4) [1]^2 - 2*Q^(2) [1]*Q^(3) [1]*Q^(4) [1] + Q^(3) [1]^3

```

REFERENCES:

•[HKOTY1999]

•[KNS2011]

class Element (M, x)

Bases: `sage.combinat.free_module.CombinatorialFreeModuleElement`

An element of a Q-system.

`QSystem.Q` (a, m)

Return the generator $Q_m^{(a)}$ of `self`.

EXAMPLES:

```

sage: Q = QSystem(QQ, ['A', 8])
sage: Q.Q(2, 1)
Q^(2) [1]
sage: Q.Q(6, 2)
-Q^(5) [1]*Q^(7) [1] + Q^(6) [1]^2
sage: Q.Q(7, 3)
-Q^(5) [1]*Q^(7) [1] + Q^(5) [1]*Q^(8) [1]^2 + Q^(6) [1]^2
- 2*Q^(6) [1]*Q^(7) [1]*Q^(8) [1] + Q^(7) [1]^3
sage: Q.Q(1, 0)
1

```

`QSystem.algebra_generators` ()

Return the algebra generators of `self`.

EXAMPLES:

```

sage: Q = QSystem(QQ, ['A', 4])
sage: Q.algebra_generators()
Finite family {1: Q^(1) [1], 2: Q^(2) [1], 3: Q^(3) [1], 4: Q^(4) [1]}

```

`QSystem.cartan_type()`
Return the Cartan type of `self`.

EXAMPLES:

```
sage: Q = QSystem(QQ, ['A', 4])
sage: Q.cartan_type()
['A', 4]
```

`QSystem.dimension()`
Return the dimension of `self`, which is ∞ .

EXAMPLES:

```
sage: F = QSystem(QQ, ['A', 4])
sage: F.dimension()
+Infinity
```

`QSystem.gens()`
Return the generators of `self`.

EXAMPLES:

```
sage: Q = QSystem(QQ, ['A', 4])
sage: Q.gens()
(Q^(1) [1], Q^(2) [1], Q^(3) [1], Q^(4) [1])
```

`QSystem.index_set()`
Return the index set of `self`.

EXAMPLES:

```
sage: Q = QSystem(QQ, ['A', 4])
sage: Q.index_set()
(1, 2, 3, 4)
```

`QSystem.level()`
Return the restriction level of `self` or `None` if the system is unrestricted.

EXAMPLES:

```
sage: Q = QSystem(QQ, ['A', 4])
sage: Q.level()

sage: Q = QSystem(QQ, ['A', 4], 5)
sage: Q.level()
5
```

`QSystem.one_basis()`
Return the basis element indexing 1.

EXAMPLES:

```
sage: Q = QSystem(QQ, ['A', 4])
sage: Q.one_basis()
1
sage: Q.one_basis().parent() is Q._indices
True
```


`sage.algebras.q_system.is_tamely_laced (ct)`

Check if the Cartan type `ct` is tamely-laced.

A (symmetrizable) Cartan type with index set I is *tamely-laced* if $A_{ij} < -1$ implies $d_i = -A_{ji} = 1$ for all $i, j \in I$, where $(d_i)_{i \in I}$ is the diagonal matrix symmetrizing the Cartan matrix $(A_{ij})_{i, j \in I}$.

EXAMPLES:

```
sage: from sage.algebras.q_system import is_tamely_laced
sage: all(is_tamely_laced(ct)
....:     for ct in CartanType.samples(crystallographic=True, finite=True))
True
sage: for ct in CartanType.samples(crystallographic=True, affine=True):
....:     if not is_tamely_laced(ct):
....:         print(ct)
['A', 1, 1]
['BC', 1, 2]
['BC', 5, 2]
['BC', 1, 2]^*
['BC', 5, 2]^*
sage: cm = CartanMatrix([[2,-1,0,0],[-3,2,-2,-2],[0,-1,2,-1],[0,-1,-1,2]])
sage: is_tamely_laced(cm)
True
```


NON-ASSOCIATIVE ALGEBRAS

6.1 Jordan Algebras

AUTHORS:

- Travis Scrimshaw (2014-04-02): initial version

class sage.algebras.jordan_algebra. **JordanAlgebra**

Bases: sage.structure.parent.Parent, sage.structure.unique_representation.UniqueRepresentation

A Jordan algebra.

A *Jordan algebra* is a magmatic algebra (over a commutative ring R) whose multiplication satisfies the following axioms:

- $xy = yx$, and
- $(xy)(xx) = x(y(xx))$ (the Jordan identity).

See [Ja1971], [Ch2012], and [McC1978], for example.

These axioms imply that a Jordan algebra is power-associative and the following generalization of Jordan's identity holds [Al1947]: $(x^m y)x^n = x^m (y x^n)$ for all $m, n \in \mathbf{Z}_{>0}$.

Let A be an associative algebra over a ring R in which 2 is invertible. We construct a Jordan algebra A^+ with ground set A by defining the multiplication as

$$x \circ y = \frac{xy + yx}{2}.$$

Often the multiplication is written as $x \circ y$ to avoid confusion with the product in the associative algebra A . We note that if A is commutative then this reduces to the usual multiplication in A .

Jordan algebras constructed in this fashion, or their subalgebras, are called *special*. All other Jordan algebras are called *exceptional*.

Jordan algebras can also be constructed from a module M over R with a symmetric bilinear form $(\cdot, \cdot) : M \times M \rightarrow R$. We begin with the module $M^* = R \oplus M$ and define multiplication in M^* by

$$(\alpha + x) \circ (\beta + y) = \underbrace{\alpha\beta + (x, y)}_{\in R} + \underbrace{\beta x + \alpha y}_{\in M}$$

where $\alpha, \beta \in R$ and $x, y \in M$.

INPUT:

Can be either an associative algebra A or a symmetric bilinear form given as a matrix (possibly followed by, or preceded by, a base ring argument)

EXAMPLES:

We let the base algebra A be the free algebra on 3 generators:

```
sage: F.<x,y,z> = FreeAlgebra(QQ)
sage: J = JordanAlgebra(F); J
Jordan algebra of Free Algebra on 3 generators (x, y, z) over Rational Field
sage: a,b,c = map(J, F.gens())
sage: a*b
1/2*x*y + 1/2*y*x
sage: b*a
1/2*x*y + 1/2*y*x
```

Jordan algebras are typically non-associative:

```
sage: (a*b)*c
1/4*x*y*z + 1/4*y*x*z + 1/4*z*x*y + 1/4*z*y*x
sage: a*(b*c)
1/4*x*y*z + 1/4*x*z*y + 1/4*y*z*x + 1/4*z*y*x
```

We check the Jordan identity:

```
sage: (a*b)*(a*a) == a*(b*(a*a))
True
sage: x = a + c
sage: y = b - 2*a
sage: (x*y)*(x*x) == x*(y*(x*x))
True
```

Next we construct a Jordan algebra from a symmetric bilinear form:

```
sage: m = matrix([[ -2, 3], [3, 4]])
sage: J.<a,b,c> = JordanAlgebra(m); J
Jordan algebra over Integer Ring given by the symmetric bilinear form:
[ -2  3]
[  3  4]
sage: a
1 + (0, 0)
sage: b
0 + (1, 0)
sage: x = 3*a - 2*b + c; x
3 + (-2, 1)
```

We again show that Jordan algebras are usually non-associative:

```
sage: (x*b)*b
-6 + (7, 0)
sage: x*(b*b)
-6 + (4, -2)
```

We verify the Jordan identity:

```
sage: y = -a + 4*b - c
sage: (x*y)*(x*x) == x*(y*(x*x))
True
```

The base ring, while normally inferred from the matrix, can also be explicitly specified:

```

sage: J.<a,b,c> = JordanAlgebra(m, QQ); J
Jordan algebra over Rational Field given by the symmetric bilinear form:
[-2  3]
[ 3  4]
sage: J.<a,b,c> = JordanAlgebra(QQ, m); J # either order work
Jordan algebra over Rational Field given by the symmetric bilinear form:
[-2  3]
[ 3  4]

```

REFERENCES:

- [Wikipedia article Jordan_algebra](#)
- [Ja1971]
- [Ch2012]
- [McC1978]
- [Al1947]

```

class sage.algebras.jordan_algebra. JordanAlgebraSymmetricBilinear ( R, form,
                                                                    names=None)

```

Bases: `sage.algebras.jordan_algebra.JordanAlgebra`

A Jordan algebra given by a symmetric bilinear form m .

```

class Element ( parent, s, v)

```

Bases: `sage.structure.element.AlgebraElement`

An element of a Jordan algebra defined by a symmetric bilinear form.

```

bar ( )

```

Return the result of the bar involution of `self`.

The bar involution $\bar{}$ is the R -linear endomorphism of M^* defined by $\bar{1} = 1$ and $\bar{x} = -x$ for $x \in M$.

EXAMPLES:

```

sage: m = matrix([[0,1],[1,1]])
sage: J.<a,b,c> = JordanAlgebra(m)
sage: x = 4*a - b + 3*c
sage: x.bar()
4 + (1, -3)

```

We check that it is an algebra morphism:

```

sage: y = 2*a + 2*b - c
sage: x.bar() * y.bar() == (x*y).bar()
True

```

```

monomial_coefficients ( copy=True)

```

Return a dictionary whose keys are indices of basis elements in the support of `self` and whose values are the corresponding coefficients.

INPUT:

- `copy` – ignored

EXAMPLES:

```

sage: m = matrix([[0,1],[1,1]])
sage: J.<a,b,c> = JordanAlgebra(m)
sage: elt = a + 2*b - c

```

```
sage: elt.monomial_coefficients()
{0: 1, 1: 2, 2: -1}
```

norm ()

Return the norm of self .

The norm of an element $\alpha + x \in M^*$ is given by $n(\alpha + x) = \alpha^2 - (x, x)$.

EXAMPLES:

```
sage: m = matrix([[0,1],[1,1]])
sage: J.<a,b,c> = JordanAlgebra(m)
sage: x = 4*a - b + 3*c; x
4 + (-1, 3)
sage: x.norm()
13
```

trace ()

Return the trace of self .

The trace of an element $\alpha + x \in M^*$ is given by $t(\alpha + x) = 2\alpha$.

EXAMPLES:

```
sage: m = matrix([[0,1],[1,1]])
sage: J.<a,b,c> = JordanAlgebra(m)
sage: x = 4*a - b + 3*c
sage: x.trace()
8
```

JordanAlgebraSymmetricBilinear. **algebra_generators ()**

Return a basis of self .

The basis returned begins with the unity of R and continues with the standard basis of M .

EXAMPLES:

```
sage: m = matrix([[0,1],[1,1]])
sage: J = JordanAlgebra(m)
sage: J.basis()
Family (1 + (0, 0), 0 + (1, 0), 0 + (0, 1))
```

JordanAlgebraSymmetricBilinear. **basis ()**

Return a basis of self .

The basis returned begins with the unity of R and continues with the standard basis of M .

EXAMPLES:

```
sage: m = matrix([[0,1],[1,1]])
sage: J = JordanAlgebra(m)
sage: J.basis()
Family (1 + (0, 0), 0 + (1, 0), 0 + (0, 1))
```

JordanAlgebraSymmetricBilinear. **gens ()**

Return the generators of self .

EXAMPLES:

```
sage: m = matrix([[0,1],[1,1]])
sage: J = JordanAlgebra(m)
sage: J.basis()
Family (1 + (0, 0), 0 + (1, 0), 0 + (0, 1))
```

`JordanAlgebraSymmetricBilinear.one()`

Return the element 1 if it exists.

EXAMPLES:

```
sage: m = matrix([[0,1],[1,1]])
sage: J = JordanAlgebra(m)
sage: J.one()
1 + (0, 0)
```

`JordanAlgebraSymmetricBilinear.zero()`

Return the element 0.

EXAMPLES:

```
sage: m = matrix([[0,1],[1,1]])
sage: J = JordanAlgebra(m)
sage: J.zero()
0 + (0, 0)
```

class `sage.algebras.jordan_algebra.SpecialJordanAlgebra` (*A, names=None*)

Bases: `sage.algebras.jordan_algebra.JordanAlgebra`

A (special) Jordan algebra A^+ from an associative algebra A .

class `Element` (*parent, x*)

Bases: `sage.structure.element.AlgebraElement`

An element of a special Jordan algebra.

monomial_coefficients (*copy=True*)

Return a dictionary whose keys are indices of basis elements in the support of `self` and whose values are the corresponding coefficients.

INPUT:

- `copy` – (default: `True`) if `self` is internally represented by a dictionary `d`, then make a copy of `d`; if `False`, then this can cause undesired behavior by mutating `d`

EXAMPLES:

```
sage: F.<x,y,z> = FreeAlgebra(QQ)
sage: J = JordanAlgebra(F)
sage: a,b,c = map(J, F.gens())
sage: elt = a + 2*b - c
sage: elt.monomial_coefficients()
{x: 1, y: 2, z: -1}
```

`SpecialJordanAlgebra.algebra_generators()`

Return the basis of `self`.

EXAMPLES:

```
sage: F.<x,y,z> = FreeAlgebra(QQ)
sage: J = JordanAlgebra(F)
sage: J.basis()
Lazy family (Term map(i))_{i in Free monoid on 3 generators (x, y, z)}
```

`SpecialJordanAlgebra.basis()`

Return the basis of self.

EXAMPLES:

```
sage: F.<x,y,z> = FreeAlgebra(QQ)
sage: J = JordanAlgebra(F)
sage: J.basis()
Lazy family (Term map(i))_{i in Free monoid on 3 generators (x, y, z)}
```

`SpecialJordanAlgebra.gens()`

Return the generators of self.

EXAMPLES:

```
sage: cat = Algebras(QQ).WithBasis().FiniteDimensional()
sage: C = CombinatorialFreeModule(QQ, ['x','y','z'], category=cat)
sage: J = JordanAlgebra(C)
sage: J.gens()
(B['x'], B['y'], B['z'])

sage: F.<x,y,z> = FreeAlgebra(QQ)
sage: J = JordanAlgebra(F)
sage: J.gens()
Traceback (most recent call last):
...
NotImplementedError: infinite set
```

`SpecialJordanAlgebra.one()`

Return the element 1 if it exists.

EXAMPLES:

```
sage: F.<x,y,z> = FreeAlgebra(QQ)
sage: J = JordanAlgebra(F)
sage: J.one()
1
```

`SpecialJordanAlgebra.zero()`

Return the element 0.

EXAMPLES:

```
sage: F.<x,y,z> = FreeAlgebra(QQ)
sage: J = JordanAlgebra(F)
sage: J.zero()
0
```

6.2 Free Pre-Lie Algebras

AUTHORS:

- Florent Hivert, Frédéric Chapoton (2011)

class `sage.combinat.free_prelie_algebra.FreePreLieAlgebra` (*R*, *names=None*)

Bases: `sage.combinat.free_module.CombinatorialFreeModule`

The free pre-Lie algebra.

Pre-Lie algebras are non-associative algebras, where the product $*$ satisfies

$$(x * y) * z - x * (y * z) = (x * z) * y - x * (z * y).$$

We use here the convention where the associator

$$(x, y, z) := (x * y) * z - x * (y * z)$$

is symmetric in its two rightmost arguments. This is sometimes called a right pre-Lie algebra.

They have appeared in numerical analysis and deformation theory.

The free Pre-Lie algebra on a given set E has an explicit description using rooted trees, just as the free associative algebra can be described using words. The underlying vector space has a basis indexed by finite rooted trees endowed with a map from their vertices to E . In this basis, the product of two (decorated) rooted trees $S * T$ is the sum over vertices of S of the rooted tree obtained by adding one edge from the root of T to the given vertex of S . The root of these trees is taken to be the root of S . The free pre-Lie algebra can also be considered as the free algebra over the PreLie operad.

Warning: The usual binary operator $*$ can be used for the pre-Lie product. Beware that it but must be parenthesized properly, as the pre-Lie product is not associative. By default, a multiple product will be taken with left parentheses.

EXAMPLES:

```
sage: F = algebras.FreePreLie(ZZ, 'xyz')
sage: x, y, z = F.gens()
sage: (x * y) * z
B[x[y[z[]]]] + B[x[y[], z[]]]
sage: (x * y) * z - x * (y * z) == (x * z) * y - x * (z * y)
True
```

The free pre-Lie algebra is non-associative:

```
sage: x * (y * z) == (x * y) * z
False
```

The default product is with left parentheses:

```
sage: x * y * z == (x * y) * z
True
sage: x * y * z * x == ((x * y) * z) * x
True
```

The NAP product as defined in [\[Liv\]](#) is also implemented on the same vector space:

```
sage: N = F.nap_product
sage: N(x*y, z*z)
B[x[y[], z[z[]]]]
```

When there is only one generator, unlabelled trees are used instead:

```
sage: F1 = algebras.FreePreLie(QQ, 'w')
sage: w = F1.gen(0); w
B[[]]
sage: w * w * w * w
B[[[]]] + B[[[]], []] + 3*B[[[]], []] + B[[[]], [], []]
```

REFERENCES:

algebra_generators ()

Return the generators of this algebra.

These are the rooted trees with just one vertex.

EXAMPLES:

```
sage: A = algebras.FreePreLie(ZZ, 'fgh'); A
Free PreLie algebra on 3 generators ['f', 'g', 'h']
over Integer Ring
sage: list(A.algebra_generators())
[B[f[]], B[g[]], B[h[]]]

sage: A = algebras.FreePreLie(QQ, ['x1', 'x2'])
sage: list(A.algebra_generators())
[B[x1[]], B[x2[]]]
```

an_element ()

Return an element of self.

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, 'xy')
sage: A.an_element()
B[x[x[x[x[]]]]] + B[x[x[], x[x[]]]]
```

degree_on_basis (t)

Return the degree of a rooted tree in the free Pre-Lie algebra.

This is the number of vertices.

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, '@')
sage: RT = A.basis().keys()
sage: A.degree_on_basis(RT([RT([])]))
2
```

gen (i)

Return the *i*-th generator of the algebra.

INPUT:

- *i* – an integer

EXAMPLES:

```
sage: F = algebras.FreePreLie(ZZ, 'xyz')
sage: F.gen(0)
B[x[]]

sage: F.gen(4)
Traceback (most recent call last):
...
IndexError: argument i (= 4) must be between 0 and 2
```

gens ()

Return the generators of self (as an algebra).

EXAMPLES:

```
sage: A = algebras.FreePreLie(ZZ, 'fgh')
sage: A.gens()
(B[f[]], B[g[]], B[h[]])
```

nap_product ()
Return the NAP product.

See also:

nap_product_on_basis()

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, '@')
sage: RT = A.basis().keys()
sage: x = A(RT([RT([])])
sage: A.nap_product(x, x)
B[[[]], [[]]]
```

nap_product_on_basis (*x*, *y*)
Return the NAP product of two trees.

This is the grafting of the root of *y* over the root of *x*. The root of the resulting tree is the root of *x*.

See also:

nap_product()

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, '@')
sage: RT = A.basis().keys()
sage: x = RT([RT([])])
sage: A.nap_product_on_basis(x, x)
B[[[]], [[]]]
```

pre_Lie_product ()
Return the pre-Lie product.

See also:

pre_Lie_product_on_basis()

EXAMPLES:

```
sage: A = algebras.FreePreLie(QQ, '@')
sage: RT = A.basis().keys()
sage: x = A(RT([RT([])])
sage: A.pre_Lie_product(x, x)
B[[[]], [[]]] + B[[[]], [[]]]
```

pre_Lie_product_on_basis (*x*, *y*)
Return the pre-Lie product of two trees.

This is the sum over all graftings of the root of *y* over a vertex of *x*. The root of the resulting trees is the root of *x*.

See also:

pre_Lie_product()

EXAMPLES:

```

sage: A = algebras.FreePreLie(QQ, '@')
sage: RT = A.basis().keys()
sage: x = RT([RT([])])
sage: A.product_on_basis(x, x)
B[[[]]] + B[[[]], [[]]]

```

product_on_basis (*x*, *y*)

Return the pre-Lie product of two trees.

This is the sum over all graftings of the root of *y* over a vertex of *x*. The root of the resulting trees is the root of *x*.

See also:

[*pre_Lie_product\(\)*](#)

EXAMPLES:

```

sage: A = algebras.FreePreLie(QQ, '@')
sage: RT = A.basis().keys()
sage: x = RT([RT([])])
sage: A.product_on_basis(x, x)
B[[[]]] + B[[[]], [[]]]

```

some_elements ()

Return some elements of the free pre-Lie algebra.

EXAMPLES:

```

sage: A = algebras.FreePreLie(QQ, '@')
sage: A.some_elements()
[B[[[]], B[[[]]]], B[[[]]] + B[[[]], [[]]], B[[[]]] + B[[[]], [[]]],
↪ B[[[]]]

```

With several generators:

```

sage: A = algebras.FreePreLie(QQ, 'xy')
sage: A.some_elements()
[B[x[]],
 B[x[x[]]],
 B[x[x[x[]]]] + B[x[x[], x[x[]]]],
 B[x[x[x[]]]] + B[x[x[], x[]]],
 B[x[x[y[]]]] + B[x[x[], y[]]]]

```

variable_names ()

Return the names of the variables.

EXAMPLES:

```

sage: R = algebras.FreePreLie(QQ, 'xy')
sage: R.variable_names()
{'x', 'y'}

```

6.3 Shuffle algebras

AUTHORS:

- Frédéric Chapoton (2013-03): Initial version

- Matthieu Deneufchatel (2013-07): Implemented dual PBW basis

class `sage.algebras.shuffle_algebra.DualPBWBasis (R, names)`
 Bases: `sage.combinat.free_module.CombinatorialFreeModule`

The basis dual to the Poincare-Birkhoff-Witt basis of the free algebra.

We recursively define the dual PBW basis as the basis of the shuffle algebra given by

$$S_w = \begin{cases} w & |w| = 1, \\ xS_u & w = xu \text{ and } w \in \text{Lyn}(X), \\ \frac{S_{\ell_{i_1}}^{*\alpha_1} * \dots * S_{\ell_{i_k}}^{*\alpha_k}}{\alpha_1! \dots \alpha_k!} & w = \ell_{i_1}^{\alpha_1} \dots \ell_{i_k}^{\alpha_k} \text{ with } \ell_1 > \dots > \ell_k \in \text{Lyn}(X). \end{cases}$$

where $S * T$ denotes the shuffle product of S and T and $\text{Lyn}(X)$ is the set of Lyndon words in the alphabet X .

The definition may be found in Theorem 5.3 of [Reu1993].

INPUT:

- `R` – ring
- `names` – names of the generators (string or an alphabet)

EXAMPLES:

```
sage: S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
sage: S
The dual Poincare-Birkhoff-Witt basis of Shuffle Algebra on 2 generators ['a', 'b
↪'] over Rational Field
sage: S.one()
S[word: ]
sage: S.one_basis()
word:
sage: T = ShuffleAlgebra(QQ, 'abcd').dual_pbw_basis(); T
The dual Poincare-Birkhoff-Witt basis of Shuffle Algebra on 4 generators ['a', 'b
↪', 'c', 'd'] over Rational Field
sage: T.algebra_generators()
(S[word: a], S[word: b], S[word: c], S[word: d])
```

class `Element (M, x)`

Bases: `sage.combinat.free_module.CombinatorialFreeModuleElement`

An element in the dual PBW basis.

expand ()

Expand `self` in words of the shuffle algebra.

EXAMPLES:

```
sage: S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
sage: f = S('ab') + S('bab')
sage: f.expand()
B[word: ab] + 2*B[word: abb] + B[word: bab]
```

`DualPBWBasis.algebra_generators ()`

Return the algebra generators of `self`.

EXAMPLES:

```
sage: S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
sage: S.algebra_generators()
(S[word: a], S[word: b])
```

DualPBWBasis.**expansion** ()

Return the morphism corresponding to the expansion into words of the shuffle algebra.

EXAMPLES:

```
sage: S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
sage: f = S('ab') + S('aba')
sage: S.expansion(f)
2*B[word: aab] + B[word: ab] + B[word: aba]
```

DualPBWBasis.**expansion_on_basis** (w)

Return the expansion of S_w in words of the shuffle algebra.

INPUT:

• w – a word

EXAMPLES:

```
sage: S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
sage: S.expansion_on_basis(Word())
B[word: ]
sage: S.expansion_on_basis(Word()).parent()
Shuffle Algebra on 2 generators ['a', 'b'] over Rational Field
sage: S.expansion_on_basis(Word('abba'))
2*B[word: aabb] + B[word: abab] + B[word: abba]
sage: S.expansion_on_basis(Word())
B[word: ]
sage: S.expansion_on_basis(Word('abab'))
2*B[word: aabb] + B[word: abab]
```

DualPBWBasis.**gen** (i)

Return the i -th generator of self .

EXAMPLES:

```
sage: S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
sage: S.gen(0)
S[word: a]
sage: S.gen(1)
S[word: b]
```

DualPBWBasis.**gens** ()

Return the algebra generators of self .

EXAMPLES:

```
sage: S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
sage: S.algebra_generators()
(S[word: a], S[word: b])
```

DualPBWBasis.**one_basis** ()

Return the indexing element of the basis element 1.

EXAMPLES:

```
sage: S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
sage: S.one_basis()
word:
```

DualPBWBasis. **product** (u, v)
Return the product of two elements u and v .

EXAMPLES:

```
sage: S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
sage: a,b = S.gens()
sage: S.product(a, b)
S[word: ba]
sage: S.product(b, a)
S[word: ba]
sage: S.product(b^2*a, a*b*a)
36*S[word: bbbaaa]
```

DualPBWBasis. **shuffle_algebra** ()
Return the associated shuffle algebra of self.

EXAMPLES:

```
sage: S = ShuffleAlgebra(QQ, 'ab').dual_pbw_basis()
sage: S.shuffle_algebra()
Shuffle Algebra on 2 generators ['a', 'b'] over Rational Field
```

class sage.algebras.shuffle_algebra. **ShuffleAlgebra** ($R, names$)
Bases: sage.combinat.free_module.CombinatorialFreeModule

The shuffle algebra on some generators over a base ring.

Shuffle algebras are commutative and associative algebras, with a basis indexed by words. The product of two words $w_1 \cdot w_2$ is given by the sum over the shuffle product of w_1 and w_2 .

See also:

For more on shuffle products, see `shuffle_product` and `shuffle()`.

REFERENCES:

- [Wikipedia article Shuffle algebra](#)

INPUT:

- R – ring
- $names$ – generator names (string or an alphabet)

EXAMPLES:

```
sage: F = ShuffleAlgebra(QQ, 'xyz'); F
Shuffle Algebra on 3 generators ['x', 'y', 'z'] over Rational Field

sage: mul(F.gens())
B[word: xyz] + B[word: xzy] + B[word: yxz] + B[word: yzx] + B[word: zxy] + B[word:
↪ zyx]

sage: mul([ F.gen(i) for i in range(2) ]) + mul([ F.gen(i+1) for i in range(2) ])
B[word: xy] + B[word: yx] + B[word: yz] + B[word: zy]

sage: S = ShuffleAlgebra(ZZ, 'abcabc'); S
```

```

Shuffle Algebra on 3 generators ['a', 'b', 'c'] over Integer Ring
sage: S.base_ring()
Integer Ring

sage: G = ShuffleAlgebra(S, 'mn'); G
Shuffle Algebra on 2 generators ['m', 'n'] over Shuffle Algebra on 3 generators [
↪ 'a', 'b', 'c'] over Integer Ring
sage: G.base_ring()
Shuffle Algebra on 3 generators ['a', 'b', 'c'] over Integer Ring

```

Shuffle algebras commute with their base ring:

```

sage: K = ShuffleAlgebra(QQ, 'ab')
sage: a,b = K.gens()
sage: K.is_commutative()
True
sage: L = ShuffleAlgebra(K, 'cd')
sage: c,d = L.gens()
sage: L.is_commutative()
True
sage: s = a*b^2 * c^3; s
(12*B[word:abb]+12*B[word:bab]+12*B[word:bba])*B[word: ccc]
sage: parent(s)
Shuffle Algebra on 2 generators ['c', 'd'] over Shuffle Algebra on 2 generators [
↪ 'a', 'b'] over Rational Field
sage: c^3 * a * b^2
(12*B[word:abb]+12*B[word:bab]+12*B[word:bba])*B[word: ccc]

```

Shuffle algebras are commutative:

```

sage: c^3 * b * a * b == c * a * c * b^2 * c
True

```

We can also manipulate elements in the basis and coerce elements from our base field:

```

sage: F = ShuffleAlgebra(QQ, 'abc')
sage: B = F.basis()
sage: B[Word('bb')] * B[Word('ca')]
B[word: bbca] + B[word: bcab] + B[word: bcba] + B[word: cabb] + B[word: cbab] +
↪ B[word: cbba]
sage: 1 - B[Word('bb')] * B[Word('ca')] / 2
B[word: ] - 1/2*B[word: bbca] - 1/2*B[word: bcab] - 1/2*B[word: bcba] - 1/
↪ 2*B[word: cabb] - 1/2*B[word: cbab] - 1/2*B[word: cbba]

```

algebra_generators ()

Return the generators of this algebra.

EXAMPLES:

```

sage: A = ShuffleAlgebra(ZZ, 'fgh'); A
Shuffle Algebra on 3 generators ['f', 'g', 'h'] over Integer Ring
sage: A.algebra_generators()
Family (B[word: f], B[word: g], B[word: h])

sage: A = ShuffleAlgebra(QQ, ['x1', 'x2'])
sage: A.algebra_generators()
Family (B[word: x1], B[word: x2])

```


coproduct (S)

Return the coproduct of the series S .

EXAMPLES:

```
sage: F = ShuffleAlgebra(QQ, 'ab')
sage: S = F.an_element(); S
B[word: ] + 2*B[word: a] + 3*B[word: b] + B[word: bab]
sage: F.coproduct(S)
B[word: ] # B[word: ] + 2*B[word: ] # B[word: a]
+ 3*B[word: ] # B[word: b] + B[word: ] # B[word: bab]
+ 2*B[word: a] # B[word: ] + B[word: a] # B[word: bb]
+ B[word: ab] # B[word: b] + 3*B[word: b] # B[word: ]
+ B[word: b] # B[word: ab] + B[word: b] # B[word: ba]
+ B[word: ba] # B[word: b] + B[word: bab] # B[word: ]
+ B[word: bb] # B[word: a]
sage: F.coproduct(F.one())
B[word: ] # B[word: ]
```

coproduct_on_basis (w)

Return the coproduct of the element of the basis indexed by the word w .

INPUT:

•w – a word

EXAMPLES:

```
sage: F = ShuffleAlgebra(QQ, 'ab')
sage: F.coproduct_on_basis(Word('a'))
B[word: ] # B[word: a] + B[word: a] # B[word: ]
sage: F.coproduct_on_basis(Word('aba'))
B[word: ] # B[word: aba] + B[word: a] # B[word: ab] + B[word: a] # B[word: ba]
+ B[word: aa] # B[word: b] + B[word: ab] # B[word: a] + B[word: aba] #
↪ B[word: ]
+ B[word: b] # B[word: aa] + B[word: ba] # B[word: a]
sage: F.coproduct_on_basis(Word())
B[word: ] # B[word: ]
```

counit (S)

Return the counit of S .

EXAMPLES:

```
sage: F = ShuffleAlgebra(QQ, 'ab')
sage: S = F.an_element(); S
B[word: ] + 2*B[word: a] + 3*B[word: b] + B[word: bab]
sage: F.counit(S)
1
```

dual_pbw_basis ()

Return the dual PBW of self .

EXAMPLES:

```
sage: A = ShuffleAlgebra(QQ, 'ab')
sage: A.dual_pbw_basis()
The dual Poincare-Birkhoff-Witt basis of Shuffle Algebra on 2 generators ['a
↪ ', 'b'] over Rational Field
```

gen (*i*)The *i* -th generator of the algebra.

INPUT:

- *i* – an integer

EXAMPLES:

```
sage: F = ShuffleAlgebra(ZZ, 'xyz')
sage: F.gen(0)
B[word: x]

sage: F.gen(4)
Traceback (most recent call last):
...
IndexError: argument i (= 4) must be between 0 and 2
```

gens ()

Return the generators of this algebra.

EXAMPLES:

```
sage: A = ShuffleAlgebra(ZZ, 'fgh'); A
Shuffle Algebra on 3 generators ['f', 'g', 'h'] over Integer Ring
sage: A.algebra_generators()
Family (B[word: f], B[word: g], B[word: h])

sage: A = ShuffleAlgebra(QQ, ['x1', 'x2'])
sage: A.algebra_generators()
Family (B[word: x1], B[word: x2])
```

is_commutative ()

Return True as the shuffle algebra is commutative.

EXAMPLES:

```
sage: R = ShuffleAlgebra(QQ, 'x')
sage: R.is_commutative()
True
sage: R = ShuffleAlgebra(QQ, 'xy')
sage: R.is_commutative()
True
```

one_basis ()Return the empty word, which index of 1 of this algebra, as per `AlgebrasWithBasis.ParentMethods.one_basis()`.

EXAMPLES:

```
sage: A = ShuffleAlgebra(QQ, 'a')
sage: A.one_basis()
word:
sage: A.one()
B[word: ]
```

product_on_basis (*w1*, *w2*)Return the product of basis elements *w1* and *w2*, as per `AlgebrasWithBasis.ParentMethods.product_on_basis()`.

INPUT:

• w_1, w_2 – Basis elements

EXAMPLES:

```
sage: A = ShuffleAlgebra(QQ, 'abc')
sage: W = A.basis().keys()
sage: A.product_on_basis(W("acb"), W("cba"))
B[word: acbacb] + B[word: acbcab] + 2*B[word: acbcha] + 2*B[word: accbab] +
↪ 4*B[word: accbba] + B[word: cabacb] + B[word: cabcab] + B[word: cabcha] +
↪ B[word: cacbab] + 2*B[word: cacbba] + 2*B[word: cbaacb] + B[word: cbacab] +
↪ B[word: cbacba]

sage: (a,b,c) = A.algebra_generators()
sage: a * (1-b)^2 * c
2*B[word: abbc] - 2*B[word: abcb] + 2*B[word: abcb] + B[word: ac] - 2*B[word:
↪ acb] + 2*B[word: acbb] + 2*B[word: babc] - 2*B[word: bac] + 2*B[word: bacb]
↪ + 2*B[word: bbac] + 2*B[word: bbca] - 2*B[word: bca] + 2*B[word: bcab] +
↪ 2*B[word: bcba] + B[word: ca] - 2*B[word: cab] + 2*B[word: cabb] - 2*B[word:
↪ cba] + 2*B[word: cbab] + 2*B[word: cbba]
```

to_dual_pbw_element (w)

Return the element w of self expressed in the dual PBW basis.

INPUT:

• w – an element of the shuffle algebra

EXAMPLES:

```
sage: A = ShuffleAlgebra(QQ, 'ab')
sage: f = 2 * A(Word()) + A(Word('ab')); f
2*B[word: ] + B[word: ab]
sage: A.to_dual_pbw_element(f)
2*S[word: ] + S[word: ab]
sage: A.to_dual_pbw_element(A.one())
S[word: ]
sage: S = A.dual_pbw_basis()
sage: elt = S.expansion_on_basis(Word('abba')); elt
2*B[word: aabb] + B[word: abab] + B[word: abba]
sage: A.to_dual_pbw_element(elt)
S[word: abba]
sage: A.to_dual_pbw_element(2*A(Word('aabb')) + A(Word('abab')))
S[word: abab]
sage: S.expansion(S('abab'))
2*B[word: aabb] + B[word: abab]
```

variable_names ()

Return the names of the variables.

EXAMPLES:

```
sage: R = ShuffleAlgebra(QQ, 'xy')
sage: R.variable_names()
{'x', 'y'}
```

6.4 Free Zinbiel Algebras

AUTHORS:

- Travis Scrimshaw (2015-09): initial version

class sage.algebras.free_zinbiel_algebra. **FreeZinbielAlgebra** (*R*, *n*, *names*)
 Bases: sage.combinat.free_module.CombinatorialFreeModule

The free Zinbiel algebra on n generators.

Let R be a ring. A *Zinbiel algebra* is a non-associative algebra with multiplication \circ that satisfies

$$a \circ (b \circ c) = a \circ (b \circ c) + a \circ (c \circ b).$$

Zinbiel algebras were first introduced by Loday (see [Lod1995] and [LV2012]) as the Koszul dual to Leibniz algebras (hence the name coined by Lemaire).

Zinbiel algebras are divided power algebras, in that for

$$x^{\circ n} = (x \circ (x \circ \cdots \circ (x \circ x) \cdots))$$

we have

$$x^{\circ m} \circ x^{\circ n} = \binom{n+m-1}{m} x^{n+m}$$

and

$$\underbrace{((x \circ \cdots \circ x \circ (x \circ x) \cdots))}_{n+1 \text{ times}} = n! x^n.$$

Note: This implies that Zinbiel algebras are not power associative.

To every Zinbiel algebra, we can construct a corresponding commutative associative algebra by using the symmetrized product:

$$a * b = a \circ b + b \circ a.$$

The free Zinbiel algebra on n generators is isomorphic as R -modules to the reduced tensor algebra $\bar{T}(R^n)$ with the product

$$(x_0 x_1 \cdots x_p) \circ (x_{p+1} x_{p+2} \cdots x_{p+q}) = \sum_{\sigma \in S_{p,q}} x_0 (x_{\sigma(1)} x_{\sigma(2)} \cdots x_{\sigma(p+q)}),$$

where $S_{p,q}$ is the set of (p, q) -shuffles.

The free Zinbiel algebra is free as a divided power algebra. Moreover, the corresponding commutative algebra is isomorphic to the (non-unital) shuffle algebra.

INPUT:

- *R* – a ring
- *n* – (optional) the number of generators
- *names* – the generator names

Warning: Currently the basis is indexed by all words over the variables, including the empty word. This is a slight abuse as it is suppose to be the indexed by all non-empty words.

EXAMPLES:

We create the free Zinbiel algebra and check the defining relation:

```
sage: Z.<x,y,z> = algebras.FreeZinbiel(QQ)
sage: (x*y)*z
Z[xyz] + Z[xzy]
sage: x*(y*z) + x*(z*y)
Z[xyz] + Z[xzy]
```

We see that the Zinbiel algebra is not associative, nor even power associative:

```
sage: x*(y*z)
Z[xyz]
sage: x*(x*x)
Z[xxx]
sage: (x*x)*x
2*Z[xxx]
```

We verify that it is a divided powers algebra:

```
sage: (x*(x*x)) * (x*(x*(x*x)))
15*Z[xxxxxxxx]
sage: binomial(3+4-1,4)
15
sage: (x*(x*(x*x))) * (x*(x*x))
20*Z[xxxxxxxx]
sage: binomial(3+4-1,3)
20
sage: ((x*x)*x)*x
6*Z[xxxx]
sage: (((x*x)*x)*x)*x
24*Z[xxxxx]
```

REFERENCES:

- [Wikipedia article Zinbiel_algebra](#)
- [Lod1995]
- [LV2012]

algebra_generators ()

Return the algebra generators of `self`.

EXAMPLES:

```
sage: Z.<x,y,z> = algebras.FreeZinbiel(QQ)
sage: list(Z.algebra_generators())
[Z[x], Z[y], Z[z]]
```

gens ()

Return the generators of `self`.

EXAMPLES:

```
sage: Z.<x,y,z> = algebras.FreeZinbiel(QQ)
sage: Z.gens()
(Z[x], Z[y], Z[z])
```

product_on_basis (x, y)

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

EXAMPLES:

```
sage: Z.<x,y,z> = algebras.FreeZinbiel(QQ)
sage: (x*y)*z # indirect doctest
Z[xyz] + Z[xzy]
```

INDICES AND TABLES

- Index
- Module Index
- Search Page

BIBLIOGRAPHY

- [Naz96] Maxim Nazarov, Young's Orthogonal Form for Brauer's Centralizer Algebra. *Journal of Algebra* 182 (1996), 664–693.
- [GL1996] J.J. Graham and G.I. Lehrer, Cellular algebras. *Inventiones mathematicae* 123 (1996), 1–34.
- [HR2005] Tom Halverson and Arun Ram, *Partition algebras*. *European Journal of Combinatorics* **26** (2005), 869–921.
- [GR1989] C. Reutenauer, A. M. Garsia. *A decomposition of Solomon's descent algebra*. *Adv. Math.* **77** (1989). <http://www.lacim.uqam.ca/~christo/Publi%C3%A9s/1989/Decomposition%20Solomon.pdf>
- [Atkinson] M. D. Atkinson. *Solomon's descent algebra revisited*. *Bull. London Math. Soc.* 24 (1992) 545-551. <http://www.cs.otago.ac.nz/staffpriv/mike/Papers/Descent/DescAlgRevisited.pdf>
- [MR-Desc] C. Malvenuto, C. Reutenauer, *Duality between quasi-symmetric functions and the Solomon descent algebra*, *Journal of Algebra* 177 (1995), no. 3, 967-982. <http://www.lacim.uqam.ca/~christo/Publi%C3%A9s/1995/Duality.pdf>
- [Schocker2004] Manfred Schocker, *The descent algebra of the symmetric group*. *Fields Inst. Comm.* 40 (2004), pp. 145-161. <http://www.mathematik.uni-bielefeld.de/~ringel/schocker-neu.ps>
- [Solomon67] Louis Solomon. *The Burnside Algebra of a Finite Group*. *Journal of Combinatorial Theory*, **2**, 1967. doi:10.1016/S0021-9800(67)80064-4.
- [Greene73] Curtis Greene. *On the Möbius algebra of a partially ordered set*. *Advances in Mathematics*, **10**, 1973. doi:10.1016/0001-8708(73)90106-0.
- [Etienne98] Gwihen Etienne. *On the Möbius algebra of geometric lattices*. *European Journal of Combinatorics*, **19**, 1998. doi:10.1006/eujc.1998.0227.
- [ChLi] F. Chapoton and M. Livernet, *Pre-Lie algebras and the rooted trees operad*, *International Math. Research Notices* (2001) no 8, pages 395-408.
- [Liv] M. Livernet, *A rigidity theorem for pre-Lie algebras*, *J. Pure Appl. Algebra* 207 (2006), no 1, pages 1-18.

a

[sage.algebras.affine_nil_temperley_lieb](#), 73
[sage.algebras.associated_graded](#), 309
[sage.algebras.catalog](#), 1
[sage.algebras.clifford_algebra](#), 95
[sage.algebras.cluster_algebra](#), 119
[sage.algebras.commutative_dga](#), 312
[sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra](#), 59
[sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element](#), 66
[sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_ideal](#), 69
[sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism](#), 70
[sage.algebras.free_algebra](#), 3
[sage.algebras.free_algebra_element](#), 14
[sage.algebras.free_algebra_quotient](#), 53
[sage.algebras.free_algebra_quotient_element](#), 56
[sage.algebras.free_zinbiel_algebra](#), 359
[sage.algebras.group_algebra](#), 180
[sage.algebras.hall_algebra](#), 149
[sage.algebras.iwahori_hecke_algebra](#), 156
[sage.algebras.jordan_algebra](#), 343
[sage.algebras.letterplace.free_algebra_element_letterplace](#), 28
[sage.algebras.letterplace.free_algebra_letterplace](#), 14
[sage.algebras.letterplace.letterplace_ideal](#), 40
[sage.algebras.nil_coxeter_algebra](#), 190
[sage.algebras.orlik_solomon](#), 191
[sage.algebras.q_system](#), 338
[sage.algebras.quatalg.quaternion_algebra](#), 204
[sage.algebras.rational_cherednik_algebra](#), 229
[sage.algebras.schur_algebra](#), 232
[sage.algebras.shuffle_algebra](#), 352
[sage.algebras.steenrod.steenrod_algebra](#), 235
[sage.algebras.steenrod.steenrod_algebra_bases](#), 270
[sage.algebras.steenrod.steenrod_algebra_misc](#), 280
[sage.algebras.steenrod.steenrod_algebra_mult](#), 291
[sage.algebras.weyl_algebra](#), 298

`sage.algebras.yokonuma_hecke_algebra`, [304](#)

C

`sage.combinat.descent_algebra`, [140](#)

`sage.combinat.diagram_algebras`, [75](#)

`sage.combinat.free_prelie_algebra`, [348](#)

`sage.combinat.partition_algebra`, [194](#)

`sage.combinat.posets.incidence_algebras`, [174](#)

`sage.combinat.posets.moebius_algebra`, [185](#)

A

a_realization() (sage.combinat.descent_algebra.DescentAlgebra method), 147
 a_realization() (sage.combinat.posets.moebius_algebra.MoebiusAlgebra method), 187
 a_realization() (sage.combinat.posets.moebius_algebra.QuantumMoebiusAlgebra method), 189
 AA() (in module sage.algebras.steenrod.steenrod_algebra), 241
 AbstractPartitionDiagram (class in sage.combinat.diagram_algebras), 75
 AbstractPartitionDiagram.options() (in module sage.combinat.diagram_algebras), 77
 AbstractPartitionDiagrams (class in sage.combinat.diagram_algebras), 77
 additive_order() (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), 248
 adem() (in module sage.algebras.steenrod.steenrod_algebra_mult), 293
 AffineNilTemperleyLiebTypeA (class in sage.algebras.affine_nil_temperley_lieb), 73
 algebra_generator() (sage.algebras.affine_nil_temperley_lieb.AffineNilTemperleyLiebTypeA method), 73
 algebra_generators() (sage.algebras.affine_nil_temperley_lieb.AffineNilTemperleyLiebTypeA method), 73
 algebra_generators() (sage.algebras.associated_graded.AssociatedGradedAlgebra method), 311
 algebra_generators() (sage.algebras.clifford_algebra.CliffordAlgebra method), 96
 algebra_generators() (sage.algebras.free_algebra.FreeAlgebra_generic method), 7
 algebra_generators() (sage.algebras.free_algebra.PBWBasisOfFreeAlgebra method), 12
 algebra_generators() (sage.algebras.free_zinbiel_algebra.FreeZinbielAlgebra method), 361
 algebra_generators() (sage.algebras.group_algebra.GroupAlgebra method), 182
 algebra_generators() (sage.algebras.jordan_algebra.JordanAlgebraSymmetricBilinear method), 346
 algebra_generators() (sage.algebras.jordan_algebra.SpecialJordanAlgebra method), 347
 algebra_generators() (sage.algebras.orlik_solomon.OrlikSolomonAlgebra method), 192
 algebra_generators() (sage.algebras.q_system.QSystem method), 339
 algebra_generators() (sage.algebras.rational_cherednik_algebra.RationalCherednikAlgebra method), 230
 algebra_generators() (sage.algebras.shuffle_algebra.DualPBWBasis method), 353
 algebra_generators() (sage.algebras.shuffle_algebra.ShuffleAlgebra method), 356
 algebra_generators() (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 256
 algebra_generators() (sage.algebras.weyl_algebra.DifferentialWeylAlgebra method), 299
 algebra_generators() (sage.algebras.yokonuma_hecke_algebra.YokonumaHeckeAlgebra method), 306
 algebra_generators() (sage.combinat.free_prelie_algebra.FreePreLieAlgebra method), 350
 ambient() (sage.algebras.cluster_algebra.ClusterAlgebra method), 126
 ambient() (sage.combinat.diagram_algebras.SubPartitionAlgebra method), 89
 an_element() (sage.algebras.rational_cherednik_algebra.RationalCherednikAlgebra method), 230
 an_element() (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 256
 an_element() (sage.combinat.free_prelie_algebra.FreePreLieAlgebra method), 350
 antiderivation() (sage.algebras.clifford_algebra.ExteriorAlgebra.Element method), 107

`antipode_on_basis()` (sage.algebras.clifford_algebra.ExteriorAlgebra method), 110
`antipode_on_basis()` (sage.algebras.hall_algebra.HallAlgebra method), 151
`antipode_on_basis()` (sage.algebras.hall_algebra.HallAlgebraMonomials method), 154
`antipode_on_basis()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 257
`arnonA_long_mono_to_string()` (in module sage.algebras.steenrod.steenrod_algebra_misc), 280
`arnonA_mono_to_string()` (in module sage.algebras.steenrod.steenrod_algebra_misc), 281
`arnonC_basis()` (in module sage.algebras.steenrod.steenrod_algebra_bases), 271
`AssociatedGradedAlgebra` (class in sage.algebras.associated_graded), 309
`atomic_basis()` (in module sage.algebras.steenrod.steenrod_algebra_bases), 271
`atomic_basis_odd()` (in module sage.algebras.steenrod.steenrod_algebra_bases), 272

B

`b_matrix()` (sage.algebras.cluster_algebra.ClusterAlgebra method), 126
`b_matrix()` (sage.algebras.cluster_algebra.ClusterAlgebraSeed method), 136
`bar()` (sage.algebras.jordan_algebra.JordanAlgebraSymmetricBilinear.Element method), 345
`bar_on_basis()` (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T method), 166
`base_diagram()` (sage.combinat.diagram_algebras.AbstractPartitionDiagram method), 76
`base_extend()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 60
`basis()` (sage.algebras.commutative_dga.GCAlgebra method), 329
`basis()` (sage.algebras.commutative_dga.GCAlgebra_multigraded method), 333
`basis()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 60
`basis()` (sage.algebras.jordan_algebra.JordanAlgebraSymmetricBilinear method), 346
`basis()` (sage.algebras.jordan_algebra.SpecialJordanAlgebra method), 347
`basis()` (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract method), 211
`basis()` (sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational method), 215
`basis()` (sage.algebras.quatalg.quaternion_algebra.QuaternionOrder method), 222
`basis()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 257
`basis()` (sage.algebras.weyl_algebra.DifferentialWeylAlgebra method), 300
`basis_coefficients()` (sage.algebras.commutative_dga.GCAlgebra.Element method), 327
`basis_for_quaternion_lattice()` (in module sage.algebras.quatalg.quaternion_algebra), 226
`basis_matrix()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_ideal.FiniteDimensionalAlgebraIdeal method), 70
`basis_matrix()` (sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational method), 215
`basis_name()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 258
`basis_name()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), 249
`BasisAbstract` (class in sage.combinat.posets.moebius_algebra), 185
`bijection_on_free_nodes()` (sage.combinat.diagram_algebras.BrauerDiagram method), 79
`binomial_mod2()` (in module sage.algebras.steenrod.steenrod_algebra_mult), 294
`binomial_modp()` (in module sage.algebras.steenrod.steenrod_algebra_mult), 294
`boundary()` (sage.algebras.clifford_algebra.ExteriorAlgebra method), 110
`brauer_diagrams()` (in module sage.combinat.diagram_algebras), 91
`BrauerAlgebra` (class in sage.combinat.diagram_algebras), 78
`BrauerDiagram` (class in sage.combinat.diagram_algebras), 79
`BrauerDiagrams` (class in sage.combinat.diagram_algebras), 81
`BrauerDiagrams.options()` (in module sage.combinat.diagram_algebras), 82

C

`c_matrix()` (sage.algebras.cluster_algebra.ClusterAlgebraSeed method), 136

`c_vector()` (sage.algebras.cluster_algebra.ClusterAlgebraSeed method), 136
`c_vectors()` (sage.algebras.cluster_algebra.ClusterAlgebraSeed method), 137
`cardinality()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 60
`cardinality()` (sage.combinat.diagram_algebras.BrauerDiagrams method), 82
`cardinality()` (sage.combinat.diagram_algebras.PartitionDiagrams method), 87
`cardinality()` (sage.combinat.diagram_algebras.PlanarDiagrams method), 88
`cardinality()` (sage.combinat.diagram_algebras.TemperleyLiebDiagrams method), 91
`cardinality()` (sage.combinat.partition_algebra.SetPartitionsBk_k method), 198
`cardinality()` (sage.combinat.partition_algebra.SetPartitionsBkhalf_k method), 198
`cardinality()` (sage.combinat.partition_algebra.SetPartitionsIk_k method), 199
`cardinality()` (sage.combinat.partition_algebra.SetPartitionsIkhalf_k method), 199
`cardinality()` (sage.combinat.partition_algebra.SetPartitionsPk_k method), 200
`cardinality()` (sage.combinat.partition_algebra.SetPartitionsPkhalf_k method), 200
`cardinality()` (sage.combinat.partition_algebra.SetPartitionsPRk_k method), 199
`cardinality()` (sage.combinat.partition_algebra.SetPartitionsPRkhalf_k method), 199
`cardinality()` (sage.combinat.partition_algebra.SetPartitionsRk_k method), 200
`cardinality()` (sage.combinat.partition_algebra.SetPartitionsRkhalf_k method), 200
`cardinality()` (sage.combinat.partition_algebra.SetPartitionsSk_k method), 201
`cardinality()` (sage.combinat.partition_algebra.SetPartitionsSkhalf_k method), 201
`cardinality()` (sage.combinat.partition_algebra.SetPartitionsTk_k method), 201
`cardinality()` (sage.combinat.partition_algebra.SetPartitionsTkhalf_k method), 201
`cartan_type()` (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra method), 170
`cartan_type()` (sage.algebras.q_system.QSystem method), 339
`cdg_algebra()` (sage.algebras.commutative_dga.GCAlgebra method), 330
`cdg_algebra()` (sage.algebras.commutative_dga.GCAlgebra_multigraded method), 333
`center_basis()` (sage.algebras.clifford_algebra.CliffordAlgebra method), 96
`chain_complex()` (sage.algebras.clifford_algebra.ExteriorAlgebraBoundary method), 116
`chain_complex()` (sage.algebras.clifford_algebra.ExteriorAlgebraCoboundary method), 118
`change_basis()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), 249
`characteristic_polynomial()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebra method), 67
`check()` (sage.combinat.diagram_algebras.AbstractPartitionDiagram method), 76
`check()` (sage.combinat.diagram_algebras.BrauerDiagram method), 80
`check()` (sage.combinat.partition_algebra.SetPartitionsXkElement method), 202
`clear_computed_data()` (sage.algebras.cluster_algebra.ClusterAlgebra method), 127
`clifford_conjugate()` (sage.algebras.clifford_algebra.CliffordAlgebraElement method), 103
`CliffordAlgebra` (class in sage.algebras.clifford_algebra), 95
`CliffordAlgebraElement` (class in sage.algebras.clifford_algebra), 103
`cluster_fan()` (sage.algebras.cluster_algebra.ClusterAlgebra method), 127
`cluster_variable()` (sage.algebras.cluster_algebra.ClusterAlgebra method), 127
`cluster_variable()` (sage.algebras.cluster_algebra.ClusterAlgebraSeed method), 137
`cluster_variables()` (sage.algebras.cluster_algebra.ClusterAlgebra method), 127
`cluster_variables()` (sage.algebras.cluster_algebra.ClusterAlgebraSeed method), 137
`cluster_variables_so_far()` (sage.algebras.cluster_algebra.ClusterAlgebra method), 128
`ClusterAlgebra` (class in sage.algebras.cluster_algebra), 124
`ClusterAlgebraElement` (class in sage.algebras.cluster_algebra), 135
`ClusterAlgebraSeed` (class in sage.algebras.cluster_algebra), 135
`coboundaries()` (sage.algebras.commutative_dga.Differential method), 313
`coboundaries()` (sage.algebras.commutative_dga.Differential_multigraded method), 324

`coboundaries()` (sage.algebras.commutative_dga.DifferentialGCAlgebra method), 318
`coboundaries()` (sage.algebras.commutative_dga.DifferentialGCAlgebra_multigraded method), 322
`coboundary()` (sage.algebras.clifford_algebra.ExteriorAlgebra method), 110
`cocycles()` (sage.algebras.commutative_dga.Differential method), 314
`cocycles()` (sage.algebras.commutative_dga.Differential_multigraded method), 325
`cocycles()` (sage.algebras.commutative_dga.DifferentialGCAlgebra method), 318
`cocycles()` (sage.algebras.commutative_dga.DifferentialGCAlgebra_multigraded method), 323
`coefficient()` (sage.algebras.cluster_algebra.ClusterAlgebra method), 128
`coefficient_names()` (sage.algebras.cluster_algebra.ClusterAlgebra method), 128
`coefficients()` (sage.algebras.cluster_algebra.ClusterAlgebra method), 128
`cohomology()` (sage.algebras.commutative_dga.Differential method), 314
`cohomology()` (sage.algebras.commutative_dga.Differential_multigraded method), 325
`cohomology()` (sage.algebras.commutative_dga.DifferentialGCAlgebra method), 319
`cohomology()` (sage.algebras.commutative_dga.DifferentialGCAlgebra_multigraded method), 323
`cohomology_generators()` (sage.algebras.commutative_dga.DifferentialGCAlgebra method), 319
`cohomology_raw()` (sage.algebras.commutative_dga.Differential method), 315
`cohomology_raw()` (sage.algebras.commutative_dga.Differential_multigraded method), 325
`cohomology_raw()` (sage.algebras.commutative_dga.DifferentialGCAlgebra method), 320
`cohomology_raw()` (sage.algebras.commutative_dga.DifferentialGCAlgebra_multigraded method), 323
`CohomologyClass` (class in sage.algebras.commutative_dga), 313
`comm_long_mono_to_string()` (in module sage.algebras.steenrod.steenrod_algebra_misc), 281
`comm_mono_to_string()` (in module sage.algebras.steenrod.steenrod_algebra_misc), 282
`commutative_ring()` (sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace method), 16
`compose()` (sage.combinat.diagram_algebras.AbstractPartitionDiagram method), 76
`conjugate()` (sage.algebras.clifford_algebra.CliffordAlgebraElement method), 103
`conjugate()` (sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational method), 215
`constant_coefficient()` (sage.algebras.clifford_algebra.ExteriorAlgebra.Element method), 108
`construction()` (sage.algebras.group_algebra.GroupAlgebra method), 182
`contains_seed()` (sage.algebras.cluster_algebra.ClusterAlgebra method), 128
`convert_from_milnor_matrix()` (in module sage.algebras.steenrod.steenrod_algebra_bases), 273
`convert_perm()` (in module sage.algebras.steenrod.steenrod_algebra_misc), 282
`convert_to_milnor_matrix()` (in module sage.algebras.steenrod.steenrod_algebra_bases), 274
`coproduct()` (sage.algebras.shuffle_algebra.ShuffleAlgebra method), 356
`coproduct()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 259
`coproduct()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), 249
`coproduct_on_basis()` (sage.algebras.clifford_algebra.ExteriorAlgebra method), 110
`coproduct_on_basis()` (sage.algebras.hall_algebra.HallAlgebra method), 152
`coproduct_on_basis()` (sage.algebras.hall_algebra.HallAlgebraMonomials method), 155
`coproduct_on_basis()` (sage.algebras.shuffle_algebra.ShuffleAlgebra method), 357
`coproduct_on_basis()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 259
`counit()` (sage.algebras.clifford_algebra.ExteriorAlgebra method), 111
`counit()` (sage.algebras.hall_algebra.HallAlgebra method), 152
`counit()` (sage.algebras.hall_algebra.HallAlgebraMonomials method), 155
`counit()` (sage.algebras.shuffle_algebra.ShuffleAlgebra method), 357
`counit_on_basis()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 260
`coxeter_group()` (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra method), 170
`create_key()` (sage.algebras.free_algebra.FreeAlgebraFactory method), 6
`create_key()` (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebraFactory method), 206
`create_object()` (sage.algebras.free_algebra.FreeAlgebraFactory method), 6
`create_object()` (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebraFactory method), 206

create_set_partition_function() (in module sage.combinat.partition_algebra), 202
 current_ring() (sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace method), 16
 current_seed() (sage.algebras.cluster_algebra.ClusterAlgebra method), 129
 cyclic_right_subideals() (sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational method), 216

D

d_vector() (sage.algebras.cluster_algebra.ClusterAlgebraElement method), 135
 deformed_euler() (sage.algebras.rational_cherednik_algebra.RationalCherednikAlgebra method), 230
 degbound() (sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace method), 17
 degree() (sage.algebras.commutative_dga.GCAlgebra.Element method), 328
 degree() (sage.algebras.commutative_dga.GCAlgebra_multigraded.Element method), 332
 degree() (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 60
 degree() (sage.algebras.letterplace.free_algebra_element_letterplace.FreeAlgebraElement_letterplace method), 28
 degree() (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), 250
 degree_negation() (sage.algebras.clifford_algebra.CliffordAlgebraElement method), 104
 degree_on_basis() (sage.algebras.associated_graded.AssociatedGradedAlgebra method), 311
 degree_on_basis() (sage.algebras.clifford_algebra.CliffordAlgebra method), 97
 degree_on_basis() (sage.algebras.clifford_algebra.ExteriorAlgebra method), 111
 degree_on_basis() (sage.algebras.orlik_solomon.OrlikSolomonAlgebra method), 193
 degree_on_basis() (sage.algebras.rational_cherednik_algebra.RationalCherednikAlgebra method), 230
 degree_on_basis() (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 260
 degree_on_basis() (sage.algebras.weyl_algebra.DifferentialWeylAlgebra method), 300
 degree_on_basis() (sage.combinat.free_prelie_algebra.FreePreLieAlgebra method), 350
 delta() (sage.combinat.posets.incidence_algebras.IncidenceAlgebra method), 175
 delta() (sage.combinat.posets.incidence_algebras.ReducedIncidenceAlgebra method), 178
 depth() (sage.algebras.cluster_algebra.ClusterAlgebraSeed method), 137
 DescentAlgebra (class in sage.combinat.descent_algebra), 140
 DescentAlgebra.B (class in sage.combinat.descent_algebra), 141
 DescentAlgebra.D (class in sage.combinat.descent_algebra), 144
 DescentAlgebra.I (class in sage.combinat.descent_algebra), 145
 DescentAlgebraBases (class in sage.combinat.descent_algebra), 147
 DescentAlgebraBases.ElementMethods (class in sage.combinat.descent_algebra), 147
 DescentAlgebraBases.ParentMethods (class in sage.combinat.descent_algebra), 148
 diagram() (sage.combinat.diagram_algebras.AbstractPartitionDiagram method), 76
 diagram() (sage.combinat.diagram_algebras.DiagramAlgebra.Element method), 83
 DiagramAlgebra (class in sage.combinat.diagram_algebras), 83
 DiagramAlgebra.Element (class in sage.combinat.diagram_algebras), 83
 diagrams() (sage.combinat.diagram_algebras.DiagramAlgebra.Element method), 83
 dict() (sage.algebras.commutative_dga.GCAlgebra.Element method), 328
 Differential (class in sage.algebras.commutative_dga), 313
 differential() (sage.algebras.commutative_dga.DifferentialGCAlgebra method), 321
 differential() (sage.algebras.commutative_dga.DifferentialGCAlgebra.Element method), 317
 differential() (sage.algebras.commutative_dga.GCAlgebra method), 330
 differential() (sage.algebras.commutative_dga.GCAlgebra_multigraded method), 334
 differential_matrix() (sage.algebras.commutative_dga.Differential method), 315
 differential_matrix_multigraded() (sage.algebras.commutative_dga.Differential_multigraded method), 326
 Differential_multigraded (class in sage.algebras.commutative_dga), 324
 DifferentialGCAlgebra (class in sage.algebras.commutative_dga), 316
 DifferentialGCAlgebra.Element (class in sage.algebras.commutative_dga), 316

DifferentialGCAAlgebra_multigraded (class in sage.algebras.commutative_dga), 322
DifferentialGCAAlgebra_multigraded.Element (class in sage.algebras.commutative_dga), 322
differentials() (sage.algebras.weyl_algebra.DifferentialWeylAlgebra method), 300
DifferentialWeylAlgebra (class in sage.algebras.weyl_algebra), 298
DifferentialWeylAlgebraElement (class in sage.algebras.weyl_algebra), 302
dimension() (sage.algebras.clifford_algebra.CliffordAlgebra method), 97
dimension() (sage.algebras.free_algebra_quotient.FreeAlgebraQuotient method), 54
dimension() (sage.algebras.q_system.QSystem method), 340
dimension() (sage.algebras.schur_algebra.SchurAlgebra method), 233
dimension() (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 261
discriminant() (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab method), 206
discriminant() (sage.algebras.quatalg.quaternion_algebra.QuaternionOrder method), 223
dual_pbw_basis() (sage.algebras.shuffle_algebra.ShuffleAlgebra method), 357
DualPBWBasis (class in sage.algebras.shuffle_algebra), 353
DualPBWBasis.Element (class in sage.algebras.shuffle_algebra), 353

E

e() (sage.algebras.yokonuma_hecke_algebra.YokonumaHeckeAlgebra method), 306
Element (sage.algebras.clifford_algebra.CliffordAlgebra attribute), 96
Element (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra attribute), 59
Element (sage.algebras.free_algebra.FreeAlgebra_generic attribute), 7
Element (sage.algebras.free_algebra_quotient.FreeAlgebraQuotient attribute), 54
Element (sage.algebras.weyl_algebra.DifferentialWeylAlgebra attribute), 299
Element (sage.combinat.diagram_algebras.AbstractPartitionDiagrams attribute), 78
Element (sage.combinat.diagram_algebras.BrauerDiagrams attribute), 82
Element (sage.combinat.partition_algebra.SetPartitionsAk_k attribute), 197
Element (sage.combinat.partition_algebra.SetPartitionsAkhalf_k attribute), 197
excess() (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), 251
expand() (sage.algebras.free_algebra.PBWBasisOfFreeAlgebra.Element method), 12
expand() (sage.algebras.shuffle_algebra.DualPBWBasis.Element method), 353
expansion() (sage.algebras.free_algebra.PBWBasisOfFreeAlgebra method), 12
expansion() (sage.algebras.shuffle_algebra.DualPBWBasis method), 354
expansion_on_basis() (sage.algebras.shuffle_algebra.DualPBWBasis method), 354
explore_to_depth() (sage.algebras.cluster_algebra.ClusterAlgebra method), 129
exterior_algebra_basis() (in module sage.algebras.commutative_dga), 337
ExteriorAlgebra (class in sage.algebras.clifford_algebra), 106
ExteriorAlgebra.Element (class in sage.algebras.clifford_algebra), 107
ExteriorAlgebraBoundary (class in sage.algebras.clifford_algebra), 114
ExteriorAlgebraCoboundary (class in sage.algebras.clifford_algebra), 116
ExteriorAlgebraDifferential (class in sage.algebras.clifford_algebra), 119

F

F_polynomial() (sage.algebras.cluster_algebra.ClusterAlgebra method), 125
F_polynomial() (sage.algebras.cluster_algebra.ClusterAlgebraSeed method), 136
F_polynomial() (sage.algebras.cluster_algebra.PrincipalClusterAlgebraElement method), 139
F_polynomials() (sage.algebras.cluster_algebra.ClusterAlgebra method), 126
F_polynomials() (sage.algebras.cluster_algebra.ClusterAlgebraSeed method), 136
F_polynomials_so_far() (sage.algebras.cluster_algebra.ClusterAlgebra method), 126
find_g_vector() (sage.algebras.cluster_algebra.ClusterAlgebra method), 129

[FiniteDimensionalAlgebra](#) (class in `sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra`), [59](#)
[FiniteDimensionalAlgebraElement](#) (class in `sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element`), [66](#)
[FiniteDimensionalAlgebraHomset](#) (class in `sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism`), [70](#)
[FiniteDimensionalAlgebraIdeal](#) (class in `sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_ideal`), [69](#)
[FiniteDimensionalAlgebraMorphism](#) (class in `sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism`), [70](#)
[free_algebra\(\)](#) (`sage.algebras.free_algebra.PBWBasisOfFreeAlgebra` method), [12](#)
[free_algebra\(\)](#) (`sage.algebras.free_algebra_quotient.FreeAlgebraQuotient` method), [54](#)
[free_module\(\)](#) (`sage.algebras.clifford_algebra.CliffordAlgebra` method), [98](#)
[free_module\(\)](#) (`sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational` method), [217](#)
[free_module\(\)](#) (`sage.algebras.quatalg.quaternion_algebra.QuaternionOrder` method), [223](#)
[FreeAlgebra_generic](#) (class in `sage.algebras.free_algebra`), [6](#)
[FreeAlgebra_letterplace](#) (class in `sage.algebras.letterplace.free_algebra_letterplace`), [16](#)
[FreeAlgebraElement](#) (class in `sage.algebras.free_algebra_element`), [14](#)
[FreeAlgebraElement_letterplace](#) (class in `sage.algebras.letterplace.free_algebra_element_letterplace`), [28](#)
[FreeAlgebraFactory](#) (class in `sage.algebras.free_algebra`), [4](#)
[FreeAlgebraQuotient](#) (class in `sage.algebras.free_algebra_quotient`), [53](#)
[FreeAlgebraQuotientElement](#) (class in `sage.algebras.free_algebra_quotient_element`), [56](#)
[FreePreLieAlgebra](#) (class in `sage.combinat.free_prelie_algebra`), [348](#)
[FreeZinbielAlgebra](#) (class in `sage.algebras.free_zinbiel_algebra`), [360](#)
[from_base_ring\(\)](#) (`sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra` method), [60](#)
[from_involution_permutation_triple\(\)](#) (`sage.combinat.diagram_algebras.BrauerDiagrams` method), [82](#)

G

[g\(\)](#) (`sage.algebras.yokonuma_hecke_algebra.YokonumaHeckeAlgebra` method), [306](#)
[g_algebra\(\)](#) (`sage.algebras.free_algebra.FreeAlgebra_generic` method), [7](#)
[g_matrix\(\)](#) (`sage.algebras.cluster_algebra.ClusterAlgebraSeed` method), [138](#)
[g_vector\(\)](#) (`sage.algebras.cluster_algebra.ClusterAlgebraSeed` method), [138](#)
[g_vector\(\)](#) (`sage.algebras.cluster_algebra.PrincipalClusterAlgebraElement` method), [140](#)
[g_vectors\(\)](#) (`sage.algebras.cluster_algebra.ClusterAlgebra` method), [130](#)
[g_vectors\(\)](#) (`sage.algebras.cluster_algebra.ClusterAlgebraSeed` method), [138](#)
[g_vectors_so_far\(\)](#) (`sage.algebras.cluster_algebra.ClusterAlgebra` method), [130](#)
[GCAAlgebra](#) (class in `sage.algebras.commutative_dga`), [327](#)
[GCAAlgebra.Element](#) (class in `sage.algebras.commutative_dga`), [327](#)
[GCAAlgebra_multigraded](#) (class in `sage.algebras.commutative_dga`), [331](#)
[GCAAlgebra_multigraded.Element](#) (class in `sage.algebras.commutative_dga`), [332](#)
[gen\(\)](#) (`sage.algebras.associated_graded.AssociatedGradedAlgebra` method), [311](#)
[gen\(\)](#) (`sage.algebras.clifford_algebra.CliffordAlgebra` method), [98](#)
[gen\(\)](#) (`sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra` method), [60](#)
[gen\(\)](#) (`sage.algebras.free_algebra.FreeAlgebra_generic` method), [7](#)
[gen\(\)](#) (`sage.algebras.free_algebra.PBWBasisOfFreeAlgebra` method), [12](#)
[gen\(\)](#) (`sage.algebras.free_algebra_quotient.FreeAlgebraQuotient` method), [54](#)
[gen\(\)](#) (`sage.algebras.group_algebra.GroupAlgebra` method), [183](#)
[gen\(\)](#) (`sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace` method), [17](#)
[gen\(\)](#) (`sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab` method), [207](#)
[gen\(\)](#) (`sage.algebras.quatalg.quaternion_algebra.QuaternionOrder` method), [223](#)

`gen()` (sage.algebras.shuffle_algebra.DualPBWBasis method), 354
`gen()` (sage.algebras.shuffle_algebra.ShuffleAlgebra method), 357
`gen()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 261
`gen()` (sage.algebras.weyl_algebra.DifferentialWeylAlgebra method), 300
`gen()` (sage.combinat.free_prelie_algebra.FreePreLieAlgebra method), 350
`generator_degrees()` (sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace method), 17
`gens()` (sage.algebras.clifford_algebra.CliffordAlgebra method), 98
`gens()` (sage.algebras.cluster_algebra.ClusterAlgebra method), 130
`gens()` (sage.algebras.free_algebra.FreeAlgebra_generic method), 8
`gens()` (sage.algebras.free_algebra.PBWBasisOffFreeAlgebra method), 13
`gens()` (sage.algebras.free_zinbiel_algebra.FreeZinbielAlgebra method), 361
`gens()` (sage.algebras.group_algebra.GroupAlgebra method), 183
`gens()` (sage.algebras.jordan_algebra.JordanAlgebraSymmetricBilinear method), 346
`gens()` (sage.algebras.jordan_algebra.SpecialJordanAlgebra method), 348
`gens()` (sage.algebras.q_system.QSystem method), 340
`gens()` (sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational method), 217
`gens()` (sage.algebras.quatalg.quaternion_algebra.QuaternionOrder method), 224
`gens()` (sage.algebras.shuffle_algebra.DualPBWBasis method), 354
`gens()` (sage.algebras.shuffle_algebra.ShuffleAlgebra method), 358
`gens()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 262
`gens()` (sage.algebras.yokonuma_hecke_algebra.YokonumaHeckeAlgebra method), 306
`gens()` (sage.combinat.free_prelie_algebra.FreePreLieAlgebra method), 350
`get_basis_name()` (in module sage.algebras.steenrod.steenrod_algebra_misc), 283
`GL_irreducible_character()` (in module sage.algebras.schur_algebra), 232
`global_options()` (sage.combinat.diagram_algebras.AbstractPartitionDiagram method), 76
`global_options()` (sage.combinat.diagram_algebras.BrauerDiagrams method), 82
`goldman_involution_on_basis()` (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.A method), 161
`goldman_involution_on_basis()` (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.B method), 162
`goldman_involution_on_basis()` (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T method), 166
`graded_algebra()` (sage.algebras.clifford_algebra.CliffordAlgebra method), 98
`graded_commutative_algebra()` (sage.algebras.commutative_dga.DifferentialGCAAlgebra method), 321
`GradedCommutativeAlgebra()` (in module sage.algebras.commutative_dga), 334
`gram_matrix()` (sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational method), 218
`greedy_element()` (sage.algebras.cluster_algebra.ClusterAlgebra method), 130
`groebner_basis()` (sage.algebras.letterplace.letterplace_ideal.LetterplaceIdeal method), 42
`group()` (sage.algebras.group_algebra.GroupAlgebra method), 183
`group()` (sage.algebras.group_algebra.GroupAlgebraFunctor method), 185
`GroupAlgebra` (class in sage.algebras.group_algebra), 181
`GroupAlgebraFunctor` (class in sage.algebras.group_algebra), 185

H

`HallAlgebra` (class in sage.algebras.hall_algebra), 149
`HallAlgebra.Element` (class in sage.algebras.hall_algebra), 151
`HallAlgebraMonomials` (class in sage.algebras.hall_algebra), 153
`HallAlgebraMonomials.Element` (class in sage.algebras.hall_algebra), 154
`hamilton_quatalg()` (in module sage.algebras.free_algebra_quotient), 55
`has_no_braid_relation()` (sage.algebras.affine_nil_temperley_lieb.AffineNilTemperleyLiebTypeA method), 74
`hash_involution_on_basis()` (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.C method), 163
`hash_involution_on_basis()` (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.Cp method), 165
`hash_involution_on_basis()` (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T method), 167

[hodge_dual\(\)](#) (sage.algebras.clifford_algebra.ExteriorAlgebra.Element method), 108
[homogeneous_component\(\)](#) (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 263
[homogeneous_components\(\)](#) (sage.algebras.cluster_algebra.PrincipalClusterAlgebraElement method), 140
[homogeneous_generator_noncommutative_variables\(\)](#) (sage.algebras.nil_coxeter_algebra.NilCoxeterAlgebra method), 190
[homogeneous_noncommutative_variables\(\)](#) (sage.algebras.nil_coxeter_algebra.NilCoxeterAlgebra method), 191
[homology\(\)](#) (sage.algebras.clifford_algebra.ExteriorAlgebraDifferential method), 119

I

[ideal\(\)](#) (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 61
[ideal\(\)](#) (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab method), 207
[ideal_diagrams\(\)](#) (in module sage.combinat.diagram_algebras), 91
[ideal_monoid\(\)](#) (sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace method), 17
[IdealDiagrams](#) (class in sage.combinat.diagram_algebras), 84
[idempotent\(\)](#) (sage.combinat.descent_algebra.DescentAlgebra.I method), 146
[identity\(\)](#) (in module sage.combinat.partition_algebra), 202
[identity_set_partition\(\)](#) (in module sage.combinat.diagram_algebras), 91
[IncidenceAlgebra](#) (class in sage.combinat.posets.incidence_algebras), 174
[IncidenceAlgebra.Element](#) (class in sage.combinat.posets.incidence_algebras), 174
[index_cmp\(\)](#) (in module sage.algebras.iwahori_hecke_algebra), 173
[index_set\(\)](#) (sage.algebras.affine_nil_temperley_lieb.AffineNilTemperleyLiebTypeA method), 74
[index_set\(\)](#) (sage.algebras.q_system.QSystem method), 340
[initial_cluster_variable\(\)](#) (sage.algebras.cluster_algebra.ClusterAlgebra method), 131
[initial_cluster_variable_names\(\)](#) (sage.algebras.cluster_algebra.ClusterAlgebra method), 131
[initial_cluster_variables\(\)](#) (sage.algebras.cluster_algebra.ClusterAlgebra method), 131
[initial_seed\(\)](#) (sage.algebras.cluster_algebra.ClusterAlgebra method), 131
[inner_product_matrix\(\)](#) (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab method), 207
[inner_product_matrix\(\)](#) (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract method), 212
[interior_product\(\)](#) (sage.algebras.clifford_algebra.ExteriorAlgebra.Element method), 108
[interior_product_on_basis\(\)](#) (sage.algebras.clifford_algebra.ExteriorAlgebra method), 111
[intersection\(\)](#) (sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational method), 218
[intersection\(\)](#) (sage.algebras.quatalg.quaternion_algebra.QuaternionOrder method), 224
[intersection_of_row_modules_over_ZZ\(\)](#) (in module sage.algebras.quatalg.quaternion_algebra), 227
[invariants\(\)](#) (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab method), 208
[inverse\(\)](#) (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement method), 67
[inverse\(\)](#) (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T.Element method), 165
[inverse\(\)](#) (sage.algebras.yokonuma_hecke_algebra.YokonumaHeckeAlgebra.Element method), 305
[inverse_g\(\)](#) (sage.algebras.yokonuma_hecke_algebra.YokonumaHeckeAlgebra method), 306
[inverse_generator\(\)](#) (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T method), 167
[inverse_generators\(\)](#) (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T method), 168
[inverse_image\(\)](#) (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism.FiniteDimensionalAlgebraMorphism method), 71
[involution_permutation_triple\(\)](#) (sage.combinat.diagram_algebras.BrauerDiagram method), 80
[is_associative\(\)](#) (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 61
[is_coboundary\(\)](#) (sage.algebras.commutative_dga.DifferentialGCAAlgebra.Element method), 317
[is_cohomologous_to\(\)](#) (sage.algebras.commutative_dga.DifferentialGCAAlgebra.Element method), 317
[is_commutative\(\)](#) (sage.algebras.clifford_algebra.CliffordAlgebra method), 98

`is_commutative()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 61

`is_commutative()` (sage.algebras.free_algebra.FreeAlgebra_generic method), 8

`is_commutative()` (sage.algebras.group_algebra.GroupAlgebra method), 183

`is_commutative()` (sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace method), 18

`is_commutative()` (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract method), 212

`is_commutative()` (sage.algebras.shuffle_algebra.ShuffleAlgebra method), 358

`is_commutative()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 264

`is_commutative()` (sage.combinat.descent_algebra.DescentAlgebraBases.ParentMethods method), 148

`is_decomposable()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), 251

`is_division_algebra()` (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract method), 212

`is_division_algebra()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 264

`is_elementary_symmetric()` (sage.combinat.diagram_algebras.BrauerDiagram method), 80

`is_equivalent()` (sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational method), 218

`is_exact()` (sage.algebras.group_algebra.GroupAlgebra method), 183

`is_exact()` (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract method), 212

`is_field()` (sage.algebras.free_algebra.FreeAlgebra_generic method), 8

`is_field()` (sage.algebras.group_algebra.GroupAlgebra method), 184

`is_field()` (sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace method), 18

`is_field()` (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract method), 212

`is_field()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 265

`is_field()` (sage.combinat.descent_algebra.DescentAlgebraBases.ParentMethods method), 148

`is_finite()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 61

`is_finite()` (sage.algebras.group_algebra.GroupAlgebra method), 184

`is_finite()` (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract method), 213

`is_finite()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 265

`is_FreeAlgebra()` (in module sage.algebras.free_algebra), 13

`is_FreeAlgebraQuotientElement()` (in module sage.algebras.free_algebra_quotient_element), 56

`is_generic()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 265

`is_homogeneous()` (sage.algebras.cluster_algebra.PrincipalClusterAlgebraElement method), 140

`is_homogeneous()` (sage.algebras.commutative_dga.GCAlgebra.Element method), 329

`is_homogeneous()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), 252

`is_integral_domain()` (sage.algebras.group_algebra.GroupAlgebra method), 184

`is_integral_domain()` (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract method), 213

`is_integral_domain()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 265

`is_invertible()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement method), 67

`is_matrix_ring()` (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract method), 213

`is_nilpotent()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement method), 68

`is_nilpotent()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), 252

`is_noetherian()` (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract method), 213

`is_noetherian()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 265

`is_planar()` (in module sage.combinat.diagram_algebras), 91

`is_planar()` (in module sage.combinat.partition_algebra), 202

`is_QuaternionAlgebra()` (in module sage.algebras.quatalg.quaternion_algebra), 227

`is_tamely_laced()` (in module sage.algebras.q_system), 340

`is_unit()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), 252

`is_unit()` (sage.combinat.posets.incidence_algebras.IncidenceAlgebra.Element method), 175

[is_unit\(\)](#) (sage.combinat.posets.incidence_algebras.ReducedIncidenceAlgebra.Element method), 177
[is_unitary\(\)](#) (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 62
[is_valid_profile\(\)](#) (in module sage.algebras.steenrod.steenrod_algebra_misc), 284
[is_zero\(\)](#) (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 63
[is_zerodivisor\(\)](#) (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement method), 68
[IwahoriHeckeAlgebra](#) (class in sage.algebras.iwahori_hecke_algebra), 156
[IwahoriHeckeAlgebra.A](#) (class in sage.algebras.iwahori_hecke_algebra), 160
[IwahoriHeckeAlgebra.B](#) (class in sage.algebras.iwahori_hecke_algebra), 161
[IwahoriHeckeAlgebra.C](#) (class in sage.algebras.iwahori_hecke_algebra), 162
[IwahoriHeckeAlgebra.Cp](#) (class in sage.algebras.iwahori_hecke_algebra), 164
[IwahoriHeckeAlgebra.T](#) (class in sage.algebras.iwahori_hecke_algebra), 165
[IwahoriHeckeAlgebra.T.Element](#) (class in sage.algebras.iwahori_hecke_algebra), 165
[IwahoriHeckeAlgebra_nonstandard](#) (class in sage.algebras.iwahori_hecke_algebra), 170
[IwahoriHeckeAlgebra_nonstandard.C](#) (class in sage.algebras.iwahori_hecke_algebra), 171
[IwahoriHeckeAlgebra_nonstandard.Cp](#) (class in sage.algebras.iwahori_hecke_algebra), 171
[IwahoriHeckeAlgebra_nonstandard.T](#) (class in sage.algebras.iwahori_hecke_algebra), 172

J

[JordanAlgebra](#) (class in sage.algebras.jordan_algebra), 343
[JordanAlgebraSymmetricBilinear](#) (class in sage.algebras.jordan_algebra), 345
[JordanAlgebraSymmetricBilinear.Element](#) (class in sage.algebras.jordan_algebra), 345
[jucys_murphy\(\)](#) (sage.combinat.diagram_algebras.BrauerAlgebra method), 79

K

[k_schur_noncommutative_variables\(\)](#) (sage.algebras.nil_coxeter_algebra.NilCoxeterAlgebra method), 191

L

[lattice\(\)](#) (sage.combinat.posets.moebius_algebra.MoebiusAlgebra method), 187
[lattice\(\)](#) (sage.combinat.posets.moebius_algebra.QuantumMoebiusAlgebra method), 190
[lc\(\)](#) (sage.algebras.letterplace.free_algebra_element_letterplace.FreeAlgebraElement_letterplace method), 28
[left_ideal\(\)](#) (sage.algebras.quatalg.quaternion_algebra.QuaternionOrder method), 224
[left_matrix\(\)](#) (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement method), 68
[left_order\(\)](#) (sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational method), 218
[left_table\(\)](#) (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 63
[letterplace_polynomial\(\)](#) (sage.algebras.letterplace.free_algebra_element_letterplace.FreeAlgebraElement_letterplace method), 29
[LetterplaceIdeal](#) (class in sage.algebras.letterplace.letterplace_ideal), 41
[level\(\)](#) (sage.algebras.q_system.QSystem method), 340
[lie_polynomial\(\)](#) (sage.algebras.free_algebra.FreeAlgebra_generic method), 8
[lift\(\)](#) (sage.algebras.cluster_algebra.ClusterAlgebra method), 131
[lift\(\)](#) (sage.combinat.diagram_algebras.SubPartitionAlgebra method), 89
[lift\(\)](#) (sage.combinat.posets.incidence_algebras.ReducedIncidenceAlgebra method), 178
[lift\(\)](#) (sage.combinat.posets.incidence_algebras.ReducedIncidenceAlgebra.Element method), 177
[lift_isometry\(\)](#) (sage.algebras.clifford_algebra.CliffordAlgebra method), 99
[lift_module_morphism\(\)](#) (sage.algebras.clifford_algebra.CliffordAlgebra method), 99

`lift_morphism()` (sage.algebras.clifford_algebra.ExteriorAlgebra method), 112
`lifted_bilinear_form()` (sage.algebras.clifford_algebra.ExteriorAlgebra method), 113
`list()` (sage.algebras.clifford_algebra.CliffordAlgebraElement method), 104
`list()` (sage.algebras.weyl_algebra.DifferentialWeylAlgebraElement method), 302
`lm()` (sage.algebras.letterplace.free_algebra_element_letterplace.FreeAlgebraElement_letterplace method), 29
`lm_divides()` (sage.algebras.letterplace.free_algebra_element_letterplace.FreeAlgebraElement_letterplace method), 29
`lower_bound()` (sage.algebras.cluster_algebra.ClusterAlgebra method), 131
`lt()` (sage.algebras.letterplace.free_algebra_element_letterplace.FreeAlgebraElement_letterplace method), 30

M

`make_mono_admissible()` (in module sage.algebras.steenrod.steenrod_algebra_mult), 294
`matrix()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement method), 68
`matrix()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism.FiniteDimensionalAlgebraMorphism method), 72
`matrix_action()` (sage.algebras.free_algebra_quotient.FreeAlgebraQuotient method), 54
`maximal_ideal()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 63
`maximal_ideals()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 64
`maximal_order()` (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab method), 208
`maxord_solve_aux_eq()` (in module sage.algebras.quatalg.quaternion_algebra), 227
`may_weight()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), 252
`milnor()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 266
`milnor()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), 253
`milnor_basis()` (in module sage.algebras.steenrod.steenrod_algebra_bases), 275
`milnor_mono_to_string()` (in module sage.algebras.steenrod.steenrod_algebra_misc), 285
`milnor_multiplication()` (in module sage.algebras.steenrod.steenrod_algebra_mult), 295
`milnor_multiplication_odd()` (in module sage.algebras.steenrod.steenrod_algebra_mult), 296
`minimal_polynomial()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement method), 68
`mobius()` (sage.combinat.posets.incidence_algebras.IncidenceAlgebra method), 176
`mobius()` (sage.combinat.posets.incidence_algebras.ReducedIncidenceAlgebra method), 178
`modp_splitting_data()` (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab method), 209
`modp_splitting_map()` (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab method), 210
`module()` (sage.algebras.free_algebra_quotient.FreeAlgebraQuotient method), 55
`moebius()` (sage.combinat.posets.incidence_algebras.IncidenceAlgebra method), 176
`moebius()` (sage.combinat.posets.incidence_algebras.ReducedIncidenceAlgebra method), 178
`MoebiusAlgebra` (class in sage.combinat.posets.moebius_algebra), 186
`MoebiusAlgebra.E` (class in sage.combinat.posets.moebius_algebra), 186
`MoebiusAlgebra.I` (class in sage.combinat.posets.moebius_algebra), 186
`MoebiusAlgebraBases` (class in sage.combinat.posets.moebius_algebra), 187
`MoebiusAlgebraBases.ElementMethods` (class in sage.combinat.posets.moebius_algebra), 187
`MoebiusAlgebraBases.ParentMethods` (class in sage.combinat.posets.moebius_algebra), 187
`monoid()` (sage.algebras.free_algebra.FreeAlgebra_generic method), 9
`monoid()` (sage.algebras.free_algebra_quotient.FreeAlgebraQuotient method), 55
`monomial_basis()` (sage.algebras.free_algebra_quotient.FreeAlgebraQuotient method), 55
`monomial_basis()` (sage.algebras.hall_algebra.HallAlgebra method), 152
`monomial_coefficients()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement method), 68

method), 69
 monomial_coefficients() (sage.algebras.jordan_algebra.JordanAlgebraSymmetricBilinear.Element method), 345
 monomial_coefficients() (sage.algebras.jordan_algebra.SpecialJordanAlgebra.Element method), 347
 monomial_coefficients() (sage.algebras.weyl_algebra.DifferentialWeylAlgebraElement method), 302
 multinomial() (in module sage.algebras.steenrod.steenrod_algebra_mult), 297
 multinomial_odd() (in module sage.algebras.steenrod.steenrod_algebra_mult), 298
 multiply_by_conjugate() (sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational method), 219
 mutate() (sage.algebras.cluster_algebra.ClusterAlgebraSeed method), 138
 mutate_initial() (sage.algebras.cluster_algebra.ClusterAlgebra method), 132

N

nap_product() (sage.combinat.free_prelie_algebra.FreePreLieAlgebra method), 351
 nap_product_on_basis() (sage.combinat.free_prelie_algebra.FreePreLieAlgebra method), 351
 ngens() (sage.algebras.clifford_algebra.CliffordAlgebra method), 101
 ngens() (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 64
 ngens() (sage.algebras.free_algebra.FreeAlgebra_generic method), 9
 ngens() (sage.algebras.free_algebra_quotient.FreeAlgebraQuotient method), 55
 ngens() (sage.algebras.group_algebra.GroupAlgebra method), 184
 ngens() (sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace method), 18
 ngens() (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract method), 213
 ngens() (sage.algebras.quatalg.quaternion_algebra.QuaternionOrder method), 224
 ngens() (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 266
 ngens() (sage.algebras.weyl_algebra.DifferentialWeylAlgebra method), 301
 NilCoxeterAlgebra (class in sage.algebras.nil_coxeter_algebra), 190
 norm() (sage.algebras.jordan_algebra.JordanAlgebraSymmetricBilinear.Element method), 346
 norm() (sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational method), 219
 normal_form() (sage.algebras.letterplace.free_algebra_element_letterplace.FreeAlgebraElement_letterplace method), 30
 normalize_basis_at_p() (in module sage.algebras.quatalg.quaternion_algebra), 228
 normalize_profile() (in module sage.algebras.steenrod.steenrod_algebra_misc), 286
 normalized_laurent_polynomial() (in module sage.algebras.iwahori_hecke_algebra), 173

O

one() (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 64
 one() (sage.algebras.jordan_algebra.JordanAlgebraSymmetricBilinear method), 347
 one() (sage.algebras.jordan_algebra.SpecialJordanAlgebra method), 348
 one() (sage.algebras.schur_algebra.SchurAlgebra method), 233
 one() (sage.algebras.weyl_algebra.DifferentialWeylAlgebra method), 301
 one() (sage.combinat.descent_algebra.DescentAlgebra.I method), 146
 one() (sage.combinat.posets.incidence_algebras.IncidenceAlgebra method), 176
 one() (sage.combinat.posets.moebius_algebra.MoebiusAlgebra.E method), 186
 one() (sage.combinat.posets.moebius_algebra.MoebiusAlgebra.I method), 186
 one() (sage.combinat.posets.moebius_algebra.MoebiusAlgebraBases.ParentMethods method), 187
 one() (sage.combinat.posets.moebius_algebra.QuantumMoebiusAlgebra.E method), 189
 one_basis() (sage.algebras.affine_nil_temperley_lieb.AffineNilTemperleyLiebTypeA method), 74
 one_basis() (sage.algebras.associated_graded.AssociatedGradedAlgebra method), 311
 one_basis() (sage.algebras.clifford_algebra.CliffordAlgebra method), 101
 one_basis() (sage.algebras.free_algebra.FreeAlgebra_generic method), 9
 one_basis() (sage.algebras.free_algebra.PBWBasisOfFreeAlgebra method), 13

`one_basis()` (sage.algebras.hall_algebra.HallAlgebra method), 152
`one_basis()` (sage.algebras.hall_algebra.HallAlgebraMonomials method), 155
`one_basis()` (sage.algebras.orlik_solomon.OrlikSolomonAlgebra method), 193
`one_basis()` (sage.algebras.q_system.QSystem method), 340
`one_basis()` (sage.algebras.rational_cherednik_algebra.RationalCherednikAlgebra method), 230
`one_basis()` (sage.algebras.shuffle_algebra.DualPBWBasis method), 354
`one_basis()` (sage.algebras.shuffle_algebra.ShuffleAlgebra method), 358
`one_basis()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 266
`one_basis()` (sage.algebras.yokonuma_hecke_algebra.YokonumaHeckeAlgebra method), 307
`one_basis()` (sage.combinat.descent_algebra.DescentAlgebra.B method), 142
`one_basis()` (sage.combinat.descent_algebra.DescentAlgebra.D method), 144
`one_basis()` (sage.combinat.descent_algebra.DescentAlgebra.I method), 146
`one_basis()` (sage.combinat.diagram_algebras.DiagramAlgebra method), 84
`one_basis()` (sage.combinat.diagram_algebras.PropagatingIdeal method), 89
`one_basis()` (sage.combinat.posets.incidence_algebras.ReducedIncidenceAlgebra method), 179
`order()` (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract method), 214
`order()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 267
`order()` (sage.combinat.diagram_algebras.DiagramAlgebra method), 84
OrlikSolomonAlgebra (class in sage.algebras.orlik_solomon), 191

P

`P()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 254
`pair_to_graph()` (in module sage.combinat.diagram_algebras), 92
`pair_to_graph()` (in module sage.combinat.partition_algebra), 202
`parent()` (sage.algebras.cluster_algebra.ClusterAlgebraSeed method), 139
`partition_diagrams()` (in module sage.combinat.diagram_algebras), 92
PartitionAlgebra (class in sage.combinat.diagram_algebras), 84
PartitionAlgebra_ak (class in sage.combinat.partition_algebra), 195
PartitionAlgebra_bk (class in sage.combinat.partition_algebra), 195
PartitionAlgebra_generic (class in sage.combinat.partition_algebra), 195
PartitionAlgebra_pk (class in sage.combinat.partition_algebra), 196
PartitionAlgebra_prk (class in sage.combinat.partition_algebra), 196
PartitionAlgebra_rk (class in sage.combinat.partition_algebra), 196
PartitionAlgebra_sk (class in sage.combinat.partition_algebra), 196
PartitionAlgebra_tk (class in sage.combinat.partition_algebra), 196
PartitionAlgebraElement_ak (class in sage.combinat.partition_algebra), 194
PartitionAlgebraElement_bk (class in sage.combinat.partition_algebra), 194
PartitionAlgebraElement_generic (class in sage.combinat.partition_algebra), 195
PartitionAlgebraElement_pk (class in sage.combinat.partition_algebra), 195
PartitionAlgebraElement_prk (class in sage.combinat.partition_algebra), 195
PartitionAlgebraElement_rk (class in sage.combinat.partition_algebra), 195
PartitionAlgebraElement_sk (class in sage.combinat.partition_algebra), 195
PartitionAlgebraElement_tk (class in sage.combinat.partition_algebra), 195
PartitionDiagrams (class in sage.combinat.diagram_algebras), 87
`path_from_initial_seed()` (sage.algebras.cluster_algebra.ClusterAlgebraSeed method), 139
`pbw_basis()` (sage.algebras.free_algebra.FreeAlgebra_generic method), 9
`pbw_element()` (sage.algebras.free_algebra.FreeAlgebra_generic method), 9
PBWBasisOfFreeAlgebra (class in sage.algebras.free_algebra), 11
PBWBasisOfFreeAlgebra.Element (class in sage.algebras.free_algebra), 11
`perm()` (sage.combinat.diagram_algebras.BrauerDiagram method), 81

`planar_diagrams()` (in module `sage.combinat.diagram_algebras`), 93
`PlanarAlgebra` (class in `sage.combinat.diagram_algebras`), 87
`PlanarDiagrams` (class in `sage.combinat.diagram_algebras`), 88
`poincare_birkhoff_witt_basis()` (`sage.algebras.free_algebra.FreeAlgebra_generic` method), 9
`poly_reduce()` (in module `sage.algebras.letterplace.free_algebra_element_letterplace`), 31
`poly_reduce()` (in module `sage.algebras.letterplace.free_algebra_letterplace`), 19
`poly_reduce()` (in module `sage.algebras.letterplace.letterplace_ideal`), 44
`polynomial_ring()` (`sage.algebras.weyl_algebra.DifferentialWeylAlgebra` method), 301
`poset()` (`sage.combinat.posets.incidence_algebras.IncidenceAlgebra` method), 176
`poset()` (`sage.combinat.posets.incidence_algebras.ReducedIncidenceAlgebra` method), 179
`pre_Lie_product()` (`sage.combinat.free_prelie_algebra.FreePreLieAlgebra` method), 351
`pre_Lie_product_on_basis()` (`sage.combinat.free_prelie_algebra.FreePreLieAlgebra` method), 351
`primary_decomposition()` (`sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra` method), 65
`prime()` (`sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic` method), 267
`prime()` (`sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element` method), 253
`PrincipalClusterAlgebraElement` (class in `sage.algebras.cluster_algebra`), 139
`product()` (`sage.algebras.free_algebra.PBWBasisOffFreeAlgebra` method), 13
`product()` (`sage.algebras.shuffle_algebra.DualPBWBasis` method), 355
`product_by_generator()` (`sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T` method), 168
`product_by_generator_on_basis()` (`sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T` method), 168
`product_on_basis()` (`sage.algebras.affine_nil_temperley_lieb.AffineNilTemperleyLiebTypeA` method), 74
`product_on_basis()` (`sage.algebras.associated_graded.AssociatedGradedAlgebra` method), 311
`product_on_basis()` (`sage.algebras.free_algebra.FreeAlgebra_generic` method), 10
`product_on_basis()` (`sage.algebras.free_zinbiel_algebra.FreeZinbielAlgebra` method), 361
`product_on_basis()` (`sage.algebras.hall_algebra.HallAlgebra` method), 153
`product_on_basis()` (`sage.algebras.hall_algebra.HallAlgebraMonomials` method), 156
`product_on_basis()` (`sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T` method), 169
`product_on_basis()` (`sage.algebras.orlik_solomon.OrlikSolomonAlgebra` method), 193
`product_on_basis()` (`sage.algebras.rational_cherednik_algebra.RationalCherednikAlgebra` method), 230
`product_on_basis()` (`sage.algebras.schur_algebra.SchurAlgebra` method), 233
`product_on_basis()` (`sage.algebras.shuffle_algebra.ShuffleAlgebra` method), 358
`product_on_basis()` (`sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic` method), 267
`product_on_basis()` (`sage.algebras.yokonuma_hecke_algebra.YokonumaHeckeAlgebra` method), 307
`product_on_basis()` (`sage.combinat.descent_algebra.DescentAlgebra.B` method), 142
`product_on_basis()` (`sage.combinat.descent_algebra.DescentAlgebra.D` method), 144
`product_on_basis()` (`sage.combinat.descent_algebra.DescentAlgebra.I` method), 146
`product_on_basis()` (`sage.combinat.diagram_algebras.DiagramAlgebra` method), 84
`product_on_basis()` (`sage.combinat.free_prelie_algebra.FreePreLieAlgebra` method), 352
`product_on_basis()` (`sage.combinat.posets.incidence_algebras.IncidenceAlgebra` method), 176
`product_on_basis()` (`sage.combinat.posets.moebius_algebra.MoebiusAlgebra.E` method), 186
`product_on_basis()` (`sage.combinat.posets.moebius_algebra.MoebiusAlgebra.I` method), 187
`product_on_basis()` (`sage.combinat.posets.moebius_algebra.MoebiusAlgebraBases.ParentMethods` method), 188
`product_on_basis()` (`sage.combinat.posets.moebius_algebra.QuantumMoebiusAlgebra.E` method), 189
`profile()` (`sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic` method), 268
`propagating_number()` (in module `sage.combinat.diagram_algebras`), 93
`propagating_number()` (in module `sage.combinat.partition_algebra`), 203
`propagating_number()` (`sage.combinat.diagram_algebras.AbstractPartitionDiagram` method), 77
`PropagatingIdeal` (class in `sage.combinat.diagram_algebras`), 88
`PropagatingIdeal.Element` (class in `sage.combinat.diagram_algebras`), 89

pseudoscalar() (sage.algebras.clifford_algebra.CliffordAlgebra method), 101
pst() (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 269
pst_mono_to_string() (in module sage.algebras.steenrod.steenrod_algebra_misc), 288

Q

Q() (sage.algebras.q_system.QSystem method), 339
Q() (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 255
q1() (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra method), 170
q2() (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra method), 170
Q_exp() (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 255
QSystem (class in sage.algebras.q_system), 338
QSystem.Element (class in sage.algebras.q_system), 339
quadratic_form() (sage.algebras.clifford_algebra.CliffordAlgebra method), 102
quadratic_form() (sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational method), 220
quadratic_form() (sage.algebras.quatalg.quaternion_algebra.QuaternionOrder method), 225
QuantumMoebiusAlgebra (class in sage.combinat.posets.moebius_algebra), 188
QuantumMoebiusAlgebra.C (class in sage.combinat.posets.moebius_algebra), 188
QuantumMoebiusAlgebra.E (class in sage.combinat.posets.moebius_algebra), 188
QuantumMoebiusAlgebra.KL (class in sage.combinat.posets.moebius_algebra), 189
quaternion_algebra() (sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational method), 220
quaternion_algebra() (sage.algebras.quatalg.quaternion_algebra.QuaternionOrder method), 225
quaternion_order() (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab method), 211
quaternion_order() (sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational method), 220
QuaternionAlgebra_ab (class in sage.algebras.quatalg.quaternion_algebra), 206
QuaternionAlgebra_abstract (class in sage.algebras.quatalg.quaternion_algebra), 211
QuaternionAlgebraFactory (class in sage.algebras.quatalg.quaternion_algebra), 204
QuaternionFractionalIdeal (class in sage.algebras.quatalg.quaternion_algebra), 214
QuaternionFractionalIdeal_rational (class in sage.algebras.quatalg.quaternion_algebra), 215
QuaternionOrder (class in sage.algebras.quatalg.quaternion_algebra), 222
quo() (sage.algebras.free_algebra.FreeAlgebra_generic method), 10
quotient() (sage.algebras.commutative_dga.DifferentialGCAAlgebra method), 321
quotient() (sage.algebras.commutative_dga.GCAAlgebra method), 331
quotient() (sage.algebras.commutative_dga.GCAAlgebra_multigraded method), 334
quotient() (sage.algebras.free_algebra.FreeAlgebra_generic method), 10
quotient_map() (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 65

R

ramified_primes() (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_ab method), 211
random_element() (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 66
random_element() (sage.algebras.group_algebra.GroupAlgebra method), 185
random_element() (sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract method), 214
random_element() (sage.algebras.quatalg.quaternion_algebra.QuaternionOrder method), 225
rank() (sage.algebras.cluster_algebra.ClusterAlgebra method), 132
rank() (sage.algebras.free_algebra_quotient.FreeAlgebraQuotient method), 55
RationalCherednikAlgebra (class in sage.algebras.rational_cherednik_algebra), 229
reduce() (sage.algebras.letterplace.free_algebra_element_letterplace.FreeAlgebraElement_letterplace method), 30
reduce() (sage.algebras.letterplace.letterplace_ideal.LetterplaceIdeal method), 43
reduced_subalgebra() (sage.combinat.posets.incidence_algebras.IncidenceAlgebra method), 176

[ReducedIncidenceAlgebra](#) (class in `sage.combinat.posets.incidence_algebras`), 177
[ReducedIncidenceAlgebra.Element](#) (class in `sage.combinat.posets.incidence_algebras`), 177
[reflection\(\)](#) (`sage.algebras.clifford_algebra.CliffordAlgebraElement` method), 104
[repr_from_monomials\(\)](#) (in module `sage.algebras.weyl_algebra`), 303
[representative\(\)](#) (`sage.algebras.commutative_dga.CohomologyClass` method), 313
[reset_current_seed\(\)](#) (`sage.algebras.cluster_algebra.ClusterAlgebra` method), 132
[reset_exploring_iterator\(\)](#) (`sage.algebras.cluster_algebra.ClusterAlgebra` method), 133
[restricted_partitions\(\)](#) (in module `sage.algebras.steenrod.steenrod_algebra_bases`), 276
[retract\(\)](#) (`sage.algebras.cluster_algebra.ClusterAlgebra` method), 133
[retract\(\)](#) (`sage.combinat.diagram_algebras.SubPartitionAlgebra` method), 89
[right_ideal\(\)](#) (`sage.algebras.quatalg.quaternion_algebra.QuaternionOrder` method), 225
[right_order\(\)](#) (`sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational` method), 220
[ring\(\)](#) (`sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational` method), 221

S

[sage.algebras.affine_nil_temperley_lieb](#) (module), 73
[sage.algebras.associated_graded](#) (module), 309
[sage.algebras.catalog](#) (module), 1
[sage.algebras.clifford_algebra](#) (module), 95
[sage.algebras.cluster_algebra](#) (module), 119
[sage.algebras.commutative_dga](#) (module), 312
[sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra](#) (module), 59
[sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element](#) (module), 66
[sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_ideal](#) (module), 69
[sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism](#) (module), 70
[sage.algebras.free_algebra](#) (module), 3
[sage.algebras.free_algebra_element](#) (module), 14
[sage.algebras.free_algebra_quotient](#) (module), 53
[sage.algebras.free_algebra_quotient_element](#) (module), 56
[sage.algebras.free_zinbiel_algebra](#) (module), 359
[sage.algebras.group_algebra](#) (module), 180
[sage.algebras.hall_algebra](#) (module), 149
[sage.algebras.iwahori_hecke_algebra](#) (module), 156
[sage.algebras.jordan_algebra](#) (module), 343
[sage.algebras.letterplace.free_algebra_element_letterplace](#) (module), 28
[sage.algebras.letterplace.free_algebra_letterplace](#) (module), 14
[sage.algebras.letterplace.letterplace_ideal](#) (module), 40
[sage.algebras.nil_coxeter_algebra](#) (module), 190
[sage.algebras.orlik_solomon](#) (module), 191
[sage.algebras.q_system](#) (module), 338
[sage.algebras.quatalg.quaternion_algebra](#) (module), 204
[sage.algebras.rational_cherednik_algebra](#) (module), 229
[sage.algebras.schur_algebra](#) (module), 232
[sage.algebras.shuffle_algebra](#) (module), 352
[sage.algebras.steenrod.steenrod_algebra](#) (module), 235
[sage.algebras.steenrod.steenrod_algebra_bases](#) (module), 270
[sage.algebras.steenrod.steenrod_algebra_misc](#) (module), 280
[sage.algebras.steenrod.steenrod_algebra_mult](#) (module), 291
[sage.algebras.weyl_algebra](#) (module), 298
[sage.algebras.yokonuma_hecke_algebra](#) (module), 304

`sage.combinat.descent_algebra` (module), 140
`sage.combinat.diagram_algebras` (module), 75
`sage.combinat.free_prelie_algebra` (module), 348
`sage.combinat.partition_algebra` (module), 194
`sage.combinat.posets.incidence_algebras` (module), 174
`sage.combinat.posets.moebius_algebra` (module), 185
`scalar()` (`sage.algebras.clifford_algebra.ExteriorAlgebra.Element` method), 109
`scalar()` (`sage.algebras.hall_algebra.HallAlgebra.Element` method), 151
`scalar()` (`sage.algebras.hall_algebra.HallAlgebraMonomials.Element` method), 154
`scalars()` (`sage.algebras.cluster_algebra.ClusterAlgebra` method), 133
`scale()` (`sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational` method), 221
`schur_representative_from_index()` (in module `sage.algebras.schur_algebra`), 235
`schur_representative_indices()` (in module `sage.algebras.schur_algebra`), 235
`SchurAlgebra` (class in `sage.algebras.schur_algebra`), 232
`SchurTensorModule` (class in `sage.algebras.schur_algebra`), 234
`SchurTensorModule.Element` (class in `sage.algebras.schur_algebra`), 235
`seeds()` (`sage.algebras.cluster_algebra.ClusterAlgebra` method), 133
`serre_cartan_basis()` (in module `sage.algebras.steenrod.steenrod_algebra_bases`), 277
`serre_cartan_mono_to_string()` (in module `sage.algebras.steenrod.steenrod_algebra_misc`), 289
`set_current_seed()` (`sage.algebras.cluster_algebra.ClusterAlgebra` method), 134
`set_degbound()` (`sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace` method), 18
`set_partition_composition()` (in module `sage.combinat.diagram_algebras`), 93
`set_partition_composition()` (in module `sage.combinat.partition_algebra`), 203
`set_partitions()` (`sage.combinat.diagram_algebras.DiagramAlgebra` method), 84
`SetPartitionsAk()` (in module `sage.combinat.partition_algebra`), 196
`SetPartitionsAk_k` (class in `sage.combinat.partition_algebra`), 197
`SetPartitionsAkhalf_k` (class in `sage.combinat.partition_algebra`), 197
`SetPartitionsBk()` (in module `sage.combinat.partition_algebra`), 197
`SetPartitionsBk_k` (class in `sage.combinat.partition_algebra`), 198
`SetPartitionsBkhalf_k` (class in `sage.combinat.partition_algebra`), 198
`SetPartitionsIk()` (in module `sage.combinat.partition_algebra`), 198
`SetPartitionsIk_k` (class in `sage.combinat.partition_algebra`), 199
`SetPartitionsIkhalf_k` (class in `sage.combinat.partition_algebra`), 199
`SetPartitionsPk()` (in module `sage.combinat.partition_algebra`), 199
`SetPartitionsPk_k` (class in `sage.combinat.partition_algebra`), 199
`SetPartitionsPkhalf_k` (class in `sage.combinat.partition_algebra`), 200
`SetPartitionsPRk()` (in module `sage.combinat.partition_algebra`), 199
`SetPartitionsPRk_k` (class in `sage.combinat.partition_algebra`), 199
`SetPartitionsPRkhalf_k` (class in `sage.combinat.partition_algebra`), 199
`SetPartitionsRk()` (in module `sage.combinat.partition_algebra`), 200
`SetPartitionsRk_k` (class in `sage.combinat.partition_algebra`), 200
`SetPartitionsRkhalf_k` (class in `sage.combinat.partition_algebra`), 200
`SetPartitionsSk()` (in module `sage.combinat.partition_algebra`), 200
`SetPartitionsSk_k` (class in `sage.combinat.partition_algebra`), 201
`SetPartitionsSkhalf_k` (class in `sage.combinat.partition_algebra`), 201
`SetPartitionsTk()` (in module `sage.combinat.partition_algebra`), 201
`SetPartitionsTk_k` (class in `sage.combinat.partition_algebra`), 201
`SetPartitionsTkhalf_k` (class in `sage.combinat.partition_algebra`), 201
`SetPartitionsXkElement` (class in `sage.combinat.partition_algebra`), 201
`shuffle_algebra()` (`sage.algebras.shuffle_algebra.DualPBWBasis` method), 355

ShuffleAlgebra (class in sage.algebras.shuffle_algebra), 355
 singular_system() (in module sage.algebras.letterplace.free_algebra_element_letterplace), 34
 singular_system() (in module sage.algebras.letterplace.free_algebra_letterplace), 21
 singular_system() (in module sage.algebras.letterplace.letterplace_ideal), 46
 some_elements() (sage.algebras.rational_cherednik_algebra.RationalCherednikAlgebra method), 231
 some_elements() (sage.combinat.free_prelie_algebra.FreePreLieAlgebra method), 352
 some_elements() (sage.combinat.posets.incidence_algebras.IncidenceAlgebra method), 177
 some_elements() (sage.combinat.posets.incidence_algebras.ReducedIncidenceAlgebra method), 179
 SpecialJordanAlgebra (class in sage.algebras.jordan_algebra), 347
 SpecialJordanAlgebra.Element (class in sage.algebras.jordan_algebra), 347
 Sq() (in module sage.algebras.steenrod.steenrod_algebra), 242
 Sq() (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_mod_two method), 269
 steenrod_algebra_basis() (in module sage.algebras.steenrod.steenrod_algebra_bases), 277
 steenrod_basis_error_check() (in module sage.algebras.steenrod.steenrod_algebra_bases), 279
 SteenrodAlgebra() (in module sage.algebras.steenrod.steenrod_algebra), 242
 SteenrodAlgebra_generic (class in sage.algebras.steenrod.steenrod_algebra), 248
 SteenrodAlgebra_generic.Element (class in sage.algebras.steenrod.steenrod_algebra), 248
 SteenrodAlgebra_mod_two (class in sage.algebras.steenrod.steenrod_algebra), 269
 SubPartitionAlgebra (class in sage.combinat.diagram_algebras), 89
 subset_image() (sage.algebras.orlik_solomon.OrlikSolomonAlgebra method), 193
 super_categories() (sage.combinat.descent_algebra.DescentAlgebraBases method), 149
 super_categories() (sage.combinat.posets.moebius_algebra.MoebiusAlgebraBases method), 188
 supercenter_basis() (sage.algebras.clifford_algebra.CliffordAlgebra method), 102
 supercommutator() (sage.algebras.clifford_algebra.CliffordAlgebraElement method), 105
 support() (sage.algebras.clifford_algebra.CliffordAlgebraElement method), 105
 support() (sage.algebras.weyl_algebra.DifferentialWeylAlgebraElement method), 302
 symmetric_diagrams() (sage.combinat.diagram_algebras.BrauerDiagrams method), 83

T

t() (sage.algebras.yokonuma_hecke_algebra.YokonumaHeckeAlgebra method), 307
 table() (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 66
 temperley_lieb_diagrams() (in module sage.combinat.diagram_algebras), 93
 TemperleyLiebAlgebra (class in sage.combinat.diagram_algebras), 90
 TemperleyLiebDiagrams (class in sage.combinat.diagram_algebras), 90
 term_order_of_block() (sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace method), 19
 ternary_quadratic_form() (sage.algebras.quatalg.quaternion_algebra.QuaternionOrder method), 226
 theta_basis_element() (sage.algebras.cluster_algebra.ClusterAlgebra method), 134
 theta_series() (sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational method), 221
 theta_series_vector() (sage.algebras.quatalg.quaternion_algebra.QuaternionFractionalIdeal_rational method), 222
 to_B_basis() (sage.combinat.descent_algebra.DescentAlgebra.D method), 144
 to_B_basis() (sage.combinat.descent_algebra.DescentAlgebra.I method), 147
 to_Brauer_partition() (in module sage.combinat.diagram_algebras), 94
 to_C_basis() (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T method), 169
 to_C_basis() (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_nonstandard.T method), 172
 to_Cp_basis() (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T method), 169
 to_Cp_basis() (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_nonstandard.T method), 173
 to_D_basis() (sage.combinat.descent_algebra.DescentAlgebra.B method), 142
 to_dual_pbw_element() (sage.algebras.shuffle_algebra.ShuffleAlgebra method), 359
 to_graph() (in module sage.combinat.diagram_algebras), 94

`to_graph()` (in module `sage.combinat.partition_algebra`), 203
`to_I_basis()` (`sage.combinat.descent_algebra.DescentAlgebra.B` method), 143
`to_matrix()` (`sage.combinat.posets.incidence_algebras.IncidenceAlgebra.Element` method), 175
`to_matrix()` (`sage.combinat.posets.incidence_algebras.ReducedIncidenceAlgebra.Element` method), 178
`to_nsym()` (`sage.combinat.descent_algebra.DescentAlgebra.B` method), 143
`to_pbw_basis()` (`sage.algebras.free_algebra_element.FreeAlgebraElement` method), 14
`to_set_partition()` (in module `sage.combinat.diagram_algebras`), 94
`to_set_partition()` (in module `sage.combinat.partition_algebra`), 204
`to_symmetric_group_algebra()` (`sage.combinat.descent_algebra.DescentAlgebraBases.ElementMethods` method), 147
`to_symmetric_group_algebra()` (`sage.combinat.descent_algebra.DescentAlgebraBases.ParentMethods` method), 148
`to_symmetric_group_algebra_on_basis()` (`sage.combinat.descent_algebra.DescentAlgebra.D` method), 145
`to_symmetric_group_algebra_on_basis()` (`sage.combinat.descent_algebra.DescentAlgebraBases.ParentMethods` method), 148
`to_T_basis()` (`sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.A` method), 161
`to_T_basis()` (`sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.B` method), 162
`to_T_basis()` (`sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_nonstandard.C` method), 171
`to_T_basis()` (`sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_nonstandard.Cp` method), 172
`top_class()` (`sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic` method), 269
`total_degree()` (in module `sage.algebras.commutative_dga`), 337
`trace()` (`sage.algebras.jordan_algebra.JordanAlgebraSymmetricBilinear.Element` method), 346
`transpose()` (`sage.algebras.clifford_algebra.CliffordAlgebraElement` method), 106
`transpose_cmp()` (in module `sage.algebras.hall_algebra`), 156
`trivial_idempotent()` (`sage.algebras.rational_cherednik_algebra.RationalCherednikAlgebra` method), 231

U

`unit_ideal()` (`sage.algebras.quatalg.quaternion_algebra.QuaternionOrder` method), 226
`unpickle_QuaternionAlgebra_v0()` (in module `sage.algebras.quatalg.quaternion_algebra`), 229
`upper_bound()` (`sage.algebras.cluster_algebra.ClusterAlgebra` method), 135
`upper_cluster_algebra()` (`sage.algebras.cluster_algebra.ClusterAlgebra` method), 135

V

`variable_names()` (`sage.algebras.shuffle_algebra.ShuffleAlgebra` method), 359
`variable_names()` (`sage.combinat.free_prelie_algebra.FreePreLieAlgebra` method), 352
`variables()` (`sage.algebras.free_algebra_element.FreeAlgebraElement` method), 14
`variables()` (`sage.algebras.weyl_algebra.DifferentialWeylAlgebra` method), 301
`vector()` (`sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement` method), 69
`vector()` (`sage.algebras.free_algebra_quotient_element.FreeAlgebraQuotientElement` method), 56
`vector_space()` (`sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_ideal.FiniteDimensionalAlgebraIdeal` method), 70
`vector_space()` (`sage.algebras.quatalg.quaternion_algebra.QuaternionAlgebra_abstract` method), 214
`volume_form()` (`sage.algebras.clifford_algebra.ExteriorAlgebra` method), 114

W

`wall_height()` (`sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element` method), 254
`wall_long_mono_to_string()` (in module `sage.algebras.steenrod.steenrod_algebra_misc`), 289
`wall_mono_to_string()` (in module `sage.algebras.steenrod.steenrod_algebra_misc`), 290
`weyl_group()` (`sage.algebras.affine_nil_temperley_lieb.AffineNilTemperleyLiebTypeA` method), 75
`wood_mono_to_string()` (in module `sage.algebras.steenrod.steenrod_algebra_misc`), 290

X

`xi_degrees()` (in module `sage.algebras.steenrod.steenrod_algebra_bases`), [279](#)

Y

`YokonumaHeckeAlgebra` (class in `sage.algebras.yokonuma_hecke_algebra`), [304](#)

`YokonumaHeckeAlgebra.Element` (class in `sage.algebras.yokonuma_hecke_algebra`), [305](#)

Z

`zero()` (`sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism.FiniteDimensionalAlgebraHomset` method), [70](#)

`zero()` (`sage.algebras.jordan_algebra.JordanAlgebraSymmetricBilinear` method), [347](#)

`zero()` (`sage.algebras.jordan_algebra.SpecialJordanAlgebra` method), [348](#)

`zero()` (`sage.algebras.weyl_algebra.DifferentialWeylAlgebra` method), [301](#)

`zeta()` (`sage.combinat.posets.incidence_algebras.IncidenceAlgebra` method), [177](#)

`zeta()` (`sage.combinat.posets.incidence_algebras.ReducedIncidenceAlgebra` method), [179](#)