
Sage Reference Manual: Algebras

Release 6.3

The Sage Development Team

August 11, 2014

CONTENTS

1	Finite-Dimensional Algebras	1
2	Elements of Finite Algebras	9
3	Ideals of Finite Algebras	13
4	Morphisms Between Finite Algebras	15
5	Free algebras	17
6	Free algebra elements	29
7	Free associative unital algebras, implemented via Singular's letterplace rings	31
8	Weighted homogeneous elements of free algebras, in letterplace implementation.	43
9	Homogeneous ideals of free algebras.	53
10	Finite dimensional free algebra quotients	63
11	Free algebra quotient elements	67
12	Group algebras	69
13	Iwahori-Hecke Algebras	75
14	Hall Algebras	93
15	Shuffle algebras	101
16	The Steenrod algebra	109
17	Steenrod algebra bases	145
18	Miscellaneous functions for the Steenrod algebra and its elements	157
19	Multiplication for elements of the Steenrod algebra	169
20	Indices and Tables	177
	Bibliography	179

FINITE-DIMENSIONAL ALGEBRAS

`class sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra`

Bases: `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 methods requiring associativity assume this without checking
- `category` – (default: `FiniteDimensionalAlgebrasWithBasis(k)`) 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: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1, 0, 0], [0, 1, 0], [0, 0, 0]]), Matrix([[0, 1, 0], [0, 0, 1], [0, 0, 0]])])
sage: B
Finite-dimensional algebra of degree 3 over Rational Field
```

Element

alias of `FiniteDimensionalAlgebraElement`

base_extend (F)

Return self base changed to 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)

TESTS:

```
sage: A = FiniteDimensionalAlgebra(QQ, [Matrix([0])])
sage: a = A(0)
sage: a.parent()
Finite-dimensional algebra of degree 1 over Rational Field
sage: A(1)
Traceback (most recent call last):
...
TypeError: algebra is not unitary

sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,1,0], [0,0,0]]), Matrix([[0,1,0], [0,0,1], [0,0,0]])]
sage: B(17)
17*e0 + 17*e2
```

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]
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]
sage: B.is_commutative()
True

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

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
```

Note: If a finite-dimensional algebra over a field admits a left identity, then this is the unique left identity, and it is also a right identity.

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: B.left_table()
[
  [1 0]  [ 0  1]
  [0 1], [-1  0]
]
```

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]
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
```

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-dim
[0 1]]

sage: B = FiniteDimensionalAlgebra(QQ, [Matrix([[1,0,0], [0,1,0], [0,0,0]]), Matrix([[0,1,0],
sage: B.primary_decomposition()
[Morphism from Finite-dimensional algebra of degree 3 over Rational Field to Finite-dimensio
[0]
[1], Morphism from Finite-dimensional algebra of degree 3 over Rational Field to Finite-dime
[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-d
[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-d
[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],
```

```
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]  
]
```


ELEMENTS OF FINITE ALGEBRAS

`class sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDime`

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], [0,0,1], [0,0,0]])])
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.

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]).inverse()
1/5*e0 - 2/5*e1
```

is_invertible()

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

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], [1,0,0], [0,2,0]])])
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],
[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],
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
```

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],
sage: B(5).vector()
(5, 0, 5)
```


IDEALS OF FINITE ALGEBRAS

class sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_ideal.**FiniteDimens**

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], [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]
```


MORPHISMS BETWEEN FINITE ALGEBRAS

```
class sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism.FiniteDim
```

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], [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.FiniteDim
```

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
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`.

EXAMPLE:

```
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
```

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
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 #...:

```
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
```

TESTS:

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

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.

TESTS:

```
sage: FreeAlgebra.create_key(GF(5), ['x', 'y', 'z'])
(Finite Field of size 5, ('x', 'y', 'z'))
sage: FreeAlgebra.create_key(GF(5), ['x', 'y', 'z'], 3)
(Finite Field of size 5, ('x', 'y', 'z'))
sage: FreeAlgebra.create_key(GF(5), 3, 'xyz')
(Finite Field of size 5, ('x', 'y', 'z'))
sage: FreeAlgebra.create_key(GF(5), ['x', 'y', 'z'], implementation='letterplace')
(Multivariate Polynomial Ring in x, y, z over Finite Field of size 5,)
sage: FreeAlgebra.create_key(GF(5), ['x', 'y', 'z'], 3, implementation='letterplace')
(Multivariate Polynomial Ring in x, y, z over Finite Field of size 5,)
sage: FreeAlgebra.create_key(GF(5), 3, 'xyz', implementation='letterplace')
(Multivariate Polynomial Ring in x, y, z over Finite Field of size 5,)
sage: FreeAlgebra.create_key(GF(5), 3, 'xyz', implementation='letterplace', degrees=[1,2,3])
((1, 2, 3), Multivariate Polynomial Ring in x, y, z, x_ over Finite Field of size 5)
```

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.

TESTS:

```
sage: FreeAlgebra.create_object('4.7.1', (QQ['x', 'y'],))
Free Associative Unital Algebra on 2 generators (x, y) over Rational Field
sage: FreeAlgebra.create_object('4.7.1', (QQ['x', 'y'],)) is FreeAlgebra(QQ, ['x', 'y'])
False
```

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
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

```

TESTS:

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*.

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 - x*y*x^2
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
```

TESTS:

We test some corner cases and alternative inputs:

```
sage: F.lie_polynomial(Word('xy'))
x*y - y*x
sage: F.lie_polynomial('xy')
x*y - y*x
sage: F.lie_polynomial(M.one())
1
sage: F.lie_polynomial(Word([]))
1
sage: F.lie_polynomial('')
1
```

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] + PBW[x^2*y]*PBW[x]
+ 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, 0,0,0,1]), M([0,0,0,1, 0,0,1,0, 0,0,0,1, -1,0,0,0]), M([0,0,0,0, 0,0,0,1, 0,0,1,0, 0,0,0,0])]
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, 0,0,1,0]), M([0,0,0,1, 0,0,1,0, 0,0,0,1, -1,0,0,0]), M([0,0,0,0, 0,0,1,0, 0,0,0,1, 0,0,0,1])]
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] + PBW[y]*PBW[x*y^2]*PBW[x]
+ 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
```

TESTS:

Check that going between the two bases is the identity:

```
sage: F = FreeAlgebra(QQ, 2, 'x,y')
sage: PBW = F.pbw_basis()
sage: M = F.monoid()
sage: L = [j.to_monoid_element() for i in range(6) for j in Words('xy', i)]
sage: all(PBW(F(PBW(m))) == PBW(m) for m in L)
True
sage: all(F(PBW(F(m))) == F(m) for m in L)
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.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

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 - x*y*x + 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
```

TESTS:

Check that we have the correct parent:

```
sage: PBW.expansion(PBW(x*y)).parent() is F
True
sage: PBW.expansion(PBW.one()).parent() is F
True
```

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] + PBW[y]^2*PBW[x*y]*PBW[x]^2 + PBW[y]^3*PBW[x]^3
```

TESTS:

Check that multiplication agrees with the multiplication in the free algebra:

```
sage: F = FreeAlgebra(QQ, 2, 'x,y')
sage: PBW = F.pbw_basis()
sage: x, y = PBW.gens()
sage: F(x*y)
x*y
sage: F(x*y*x)
x*y*x
sage: PBW(F(x)*F(y)*F(x)) == x*y*x
True
```

```
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=range(1, 11)))
True
```


FREE ALGEBRA ELEMENTS

AUTHORS:

- David Kohel (2005-09)

TESTS:

```
sage: R.<x,y> = FreeAlgebra(QQ,2)
sage: x == loads(dumps(x))
True
sage: x*y
x*y
sage: (x*y)^0
1
sage: (x*y)^3
x*y*x*y*x*y
```

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

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] + 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]
```


FREE ASSOCIATIVE UNITAL ALGEBRAS, IMPLEMENTED VIA SINGULAR'S LETTERPLACE RINGS

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
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
```

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 -
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
```

TEST:

```
sage: TestSuite(F).run()
sage: TestSuite(L).run()
sage: loads(dumps(F)) is F
True
```

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
```

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

NOTE:

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

EXAMPLE:

```
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*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.

EXAMPLE:

```
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.

EXAMPLE:

```
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()`.

EXAMPLE:

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*.

EXAMPLE:

```
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.

EXAMPLE:

```
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.

EXAMPLE:

```
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.

TEST:

```

sage: F.<x,y,z> = FreeAlgebra(QQ, implementation='letterplace')
sage: F.is_field()
False

```

ngens()

Return the number of generators.

EXAMPLE:

```

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.

EXAMPLE:

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.

EXAMPLE:

```

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**)

EXAMPLE:

```
sage: groebner = sage.libs.singular.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.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.

5.1.119 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 reduce also with bad ecart (in the local case)
- * 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,
                                                                    interrupt-
                                                                    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**)

EXAMPLE:

```
sage: groebner = sage.libs.singular.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.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.

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

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

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

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

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

`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("tty")'
    resets the terminal.

`system("version")'
    returns the version number of SINGULAR as int.

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

`system("gen")'
    returns the generating element of the multiplicative group of
     $(\mathbb{Z}/p)\backslash\{0\}$  (as int) where p is the characteristic of the
    basering.

`system("nblocks")'

`system("nblocks",' ring_name `)'
    returns the number of blocks of the given ring, or the number

```

of parameters of the current basering, if no second argument is given. The return type is int.

```
`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("--")`  
    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.
```

```
`system("browsers");`  
    returns a string about available help browsers. *Note The  
    online help system:: returns the number of cpus as int (for  
    creating multiple threads/processes).
```

```
`system("pid")`
```

```
`*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  
    ==> // --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
==> // --min-time      "0.5"
==> // --cpus          2
==> // --MPport
==> // --MPhost
==> // --link
==> // --MPrsh
==> // --ticks-per-sec  1
==> // --MPtransp
==> // --MPmode
// 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 IdxFile.
==> // ** Either set environment variable SINGULAR_IDX_FILE to IdxFile,
==> // ** or make sure that IdxFile is at /home/hannes/psico2/doc/singular.id\
    x
==> // ** resource 'x' not found
==> // ** Setting help browser to 'builtin'.

```


WEIGHTED HOMOGENEOUS ELEMENTS OF FREE ALGEBRAS, IN LETTERPLACE IMPLEMENTATION.

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.

EXAMPLE:

```
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.

EXAMPLE:

```
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.

EXAMPLE:

```
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*x_3*x_4*z_5
```

lm()

The leading monomial of this free algebra element.

EXAMPLE:

```
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 $spt = q$.

EXAMPLE:

```
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.

EXAMPLE:

```
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 $\deg(x)$.

EXAMPLE:

```
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.

EXAMPLE:

```
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**)

EXAMPLE:

```
sage: groebner = sage.libs.singular.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.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.

5.1.119 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 reduce also with bad ecart (in the local case)

        * 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**)

EXAMPLE:

```
sage: groebner = sage.libs.singular.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.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.

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

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

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

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

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

`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("tty")'
    resets the terminal.

`system("version")'
    returns the version number of SINGULAR as int.

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

`system("gen")'
    returns the generating element of the multiplicative group of
     $(\mathbb{Z}/p)\backslash\{0\}$  (as int) where p is the characteristic of the
    basering.

`system("nblocks")'

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

```

`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("--")'
    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.

`system("browsers");'
    returns a string about available help browsers. *Note The
    online help system::. returns the number of cpus as int (for
    creating multiple threads/processes).

`system("pid")'

`*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
    ==> // --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
==> // --min-time       "0.5"
==> // --cpus           2
==> // --MPport
==> // --MPhost
==> // --link
==> // --MPrsh
==> // --ticks-per-sec  1
==> // --MPtransp
==> // --MPmode
// 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 IdxFile.
==> // ** Either set environment variable SINGULAR_IDX_FILE to IdxFile,
==> // ** or make sure that IdxFile is at /home/hannes/psico2/doc/singular.idx\
    x
==> // ** resource `x` not found
==> // ** Setting help browser to 'builtin'.
```


HOMOGENEOUS IDEALS OF FREE ALGEBRAS.

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
```

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 -
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,
                                                                    coerce=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 genera
```

```

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
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*

```

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*

```

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
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
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*x*x*x*z*y*y + x*x*x*x*z*y*y*x - 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 +
x*z*y*y*x*x*x + 2*x*x*z*y*y*y*x - 2*x*x*z*y*y*z - x*z*y*z*x*x -
x*z*y*z*y + y*x*z*x*x*x*x*x - 4*y*x*x*x*x*x - 4*y*x*x*x*x*x +
4*y*x*x*z*y*x*x*x + 3*y*x*x*z*y*y*x - 4*y*x*x*z*y*z + y*y*x*x*x*x*x +
y*y*x*x*x*x*z*x - 3*y*y*x*x*x*x*x - y*y*x*x*x*z*y +
5*y*y*x*x*x*x*x + 4*y*y*x*x*z*y*x - 4*y*y*y*x*x*x*z +
4*y*y*y*y*x*x*x + 3*y*y*y*y*z + 4*y*y*y*y*z*x*x + 6*y*y*y*z*y +
y*y*z*x*x*x*x + y*y*z*x*x*z + 7*y*y*z*y*x*x + 7*y*y*z*y*y -
7*y*y*z*z*x - y*z*x*x*x*x - y*z*x*x*x*x + 3*y*z*x*x*x*x +
y*z*x*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*x - z*y*x*z*x*x*x -
z*y*x*x*z*y + z*y*y*x*x*x*x*x - 3*z*y*y*x*x*z + 3*z*y*y*y*x*x*x +
z*y*y*y*y - 3*z*y*y*z*x*x - z*y*z*x*x*x*x - 2*z*y*z*y*x*x +
2*z*y*z*z - z*z*x*x*x*x*x*x + 4*z*z*x*x*x*z + 4*z*z*x*x*z*x -
4*z*z*y*x*x*x*x - 3*z*z*y*y*x*x + 4*z*z*y*z + 4*z*z*z*x*x*x +
2*z*z*z*y,
x*x*x*x*x*x*z + x*x*x*x*x*z*x + x*x*x*z*x*x*x + x*x*z*x*x*x*x + x*z*x*x*x*x*x +
y*x*x*z*y - y*y*x*x*z + y*z*z + z*x*x*x*x*x*x - z*z*y,
x*x*x*x*x*x*x - y*x*x*z - 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 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*x)
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,
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 -

```

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
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,
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

```

```

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**)

EXAMPLE:

```

sage: groebner = sage.libs.singular.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.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.

5.1.119 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 reduce also with bad ecart (in the local case)
- * 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.letterplace_ideal.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**)

EXAMPLE:

```
sage: groebner = sage.libs.singular.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.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.

5.1.141 system

***Syntax:**

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

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

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

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

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

``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("tty")``
resets the terminal.

``system("version")``
returns the version number of SINGULAR as int.

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

``system("gen")``
returns the generating element of the multiplicative group of
 $(\mathbb{Z}/p)\backslash\{0\}$ (as int) where p is the characteristic of the
basing.

``system("nblocks")``

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

``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("-")``
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.

`system("browsers");'
    returns a string about available help browsers. *Note The
    online help system::. returns the number of cpus as int (for
    creating multiple threads/processes).

`system("pid")'

`*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
    ==> // --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
    ==> // --min-time       "0.5"
    ==> // --cpus           2
    ==> // --MPport
    ==> // --MPhost
    ==> // --link

```

```
==> // --MPrsh
==> // --ticks-per-sec    1
==> // --MPtransp
==> // --MPmode
// 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 IdxFile.
==> // ** Either set environment variable SINGULAR_IDX_FILE to IdxFile,
==> // ** or make sure that IdxFile is at /home/hannes/psico2/doc/singular.id\
    x
==> // ** resource `x` not found
==> // ** Setting help browser to 'builtin'.
```

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](#).

TESTS:

```
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: H2 == loads(dumps(H2))
True
sage: i == loads(dumps(i))
True
```

```
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]),
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 + 170141183
```

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 + 170141183
```

TEST:

```
sage: TestSuite(H2).run()
```

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
```

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
```

TESTS:

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

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
```


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
() + 2*(2,4) + 3*(1,2)(3,4) + (1,2,3,4)
sage: a * a
(1,2)(3,4) + (1,2,3,4) + 3*(1,3) + (1,3)(2,4) + 6*(1,4,3,2) + 2*(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*(3,4) + 3*(2,3) + (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
```

```
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 a
```

```
sage: a = kD4.an_element(); a
() + 2*(2,4) + 3*(1,2)(3,4) + (1,2,3,4)
sage: a.antipode()
() + 2*(2,4) + 3*(1,2)(3,4) + (1,4,3,2)
sage: a.coproduct()
() # () + 2*(2,4) # (2,4) + 3*(1,2)(3,4) # (1,2)(3,4) + (1,2,3,4) # (1,2,3,4)
```

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

Create the given group algebra.

INPUT:

- `group`, a group
- `base_ring` (optional, default \mathbf{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
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
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 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)

```

TESTS:

```

sage: A = GroupAlgebra(GL(3, GF(7)))
sage: A.has_coerce_map_from(GL(3, GF(7)))
True
sage: G = SymmetricGroup(5)
sage: x, y = G.gens()
sage: A = GroupAlgebra(G)

```

```
sage: A( A(x) )
(1, 2, 3, 4, 5)
```

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 Rat
sage: A.algebra_generators()
Finite family {(1,2,3): (1,2,3), (1,3): (1,3)}
```

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 Rat
sage: A.algebra_generators()
Finite family {(1,2,3): (1,2,3), (1,3): (1,3)}
```

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.

EXAMPLE:

```
sage: GroupAlgebra(DihedralGroup(6), QQ).random_element()
-1/95*(2, 6)(3, 5) - 1/2*(1, 3)(4, 6)
sage: GroupAlgebra(SU(2, 13), QQ).random_element(1)
1/2*[      11      a + 6]
[2*a + 12      11]
```

class sage.algebras.group_algebra_new.**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_new import GroupAlgebraFunctor
sage: F = GroupAlgebraFunctor(KleinFourGroup())
sage: loads(dumps(F)) == F
True
sage: GroupAlgebra(SU(2, GF(4, 'a')), IntegerModRing(12)).category()
Category of 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_new import GroupAlgebraFunctor
sage: GroupAlgebraFunctor(CyclicPermutationGroup(17)).group() == CyclicPermutationGroup(17)
True
```

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`

Returns 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 [I64] 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], [HKP], [J87] 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])
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)

```

REFERENCES:

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
```

TESTS:

We check the defining properties of the bases:

```
sage: R.<v> = LaurentPolynomialRing(QQ, 'v')
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: W = H.coxeter_group()
sage: T = H.T()
sage: C = H.C()
sage: Cp = H.Cp()
sage: T(Cp[1])
```

```

(v^-1)*T[1] + (v^-1)
sage: T(C[1])
(v^-1)*T[1] - v
sage: C(Cp[1])
C[1] + (v^-1+v)
sage: Cp(C[1])
Cp[1] - (v^-1+v)
sage: all(C[x] == C[x].bar() for x in W) # long time
True
sage: all(Cp[x] == Cp[x].bar() for x in W) # long time
True
sage: all(T(C[x]).bar() == T(C[x]) for x in W) # long time
True
sage: all(T(Cp[x]).bar() == T(Cp[x]) for x in W) # long time
True
sage: KL = KazhdanLusztigPolynomial(W, v)
sage: term = lambda x,y: (-1)^y.length() * v^(-2*y.length()) * KL.P(y, x).substitute(v=v^-2)*T[y]
sage: all(T(C[x]) == (-v)^x.length()*sum(term(x,y) for y in W) for x in W) # long time
True
sage: all(T(Cp[x]) == v^-x.length()*sum(KL.P(y,x).substitute(v=v^2)*T[y] for y in W) for x in W)
True

```

We check conversion between the bases for type B_2 as well as some of the defining properties:

```

sage: H = IwahoriHeckeAlgebra(['B', 2], v**2)
sage: W = H.coxeter_group()
sage: T = H.T()
sage: C = H.C()
sage: Cp = H.Cp()
sage: all(T[x] == T(C(T[x])) for x in W) # long time
True
sage: all(T[x] == T(Cp(T[x])) for x in W) # long time
True
sage: all(C[x] == C(T(C[x])) for x in W) # long time
True
sage: all(C[x] == C(Cp(C[x])) for x in W) # long time
True
sage: all(Cp[x] == Cp(T(Cp[x])) for x in W) # long time
True
sage: all(Cp[x] == Cp(C(Cp[x])) for x in W) # long time
True
sage: all(T(C[x]).bar() == T(C[x]) for x in W) # long time
True
sage: all(T(Cp[x]).bar() == T(Cp[x]) for x in W) # long time
True
sage: KL = KazhdanLusztigPolynomial(W, v)
sage: term = lambda x,y: (-1)^y.length() * v^(-2*y.length()) * KL.P(y, x).substitute(v=v^-2)*T[y]
sage: all(T(C[x]) == (-v)^x.length()*sum(term(x,y) for y in W) for x in W) # long time
True
sage: all(T(Cp[x]) == v^-x.length()*sum(KL.P(y,x).substitute(v=v^2)*T[y] for y in W) for x in W)
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 C (*IHAlgebra*, *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 $\deg P_{v,w}(q) \leq \frac{1}{2}(\ell(w) - \ell(v) - 1)$.

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 then the corresponding Kazhdan-Lusztig basis element is:

$$C_s = (-q_1 q_2)^{1/2} (1 - (-q_1 q_2)^{-1/2} T_s).$$

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 $\alpha(T_i) = -(q_1 q_2)^{-1/2} T_i$.

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]
```

TESTS:

```
sage: R.<v> = LaurentPolynomialRing(QQ, 'v')
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: W = H.coxeter_group()
sage: T = H.T()
sage: C = H.C()
sage: Cp = H.Cp()
sage: all(C(T(C[x])) == C[x] for x in W) # long time
True
sage: all(C(Cp(C[x])) == C[x] for x in W) # long time
True
```

Check the defining property between C and C' :

```
sage: T(C[1])
(v^-1)*T[1] - v
sage: -T(Cp[1]).hash_involution()
(v^-1)*T[1] - v
sage: T(Cp[1] + Cp[2]).hash_involution()
-(v^-1)*T[1] - (v^-1)*T[2] + 2*v
sage: -T(C[1] + C[2])
-(v^-1)*T[1] - (v^-1)*T[2] + 2*v
sage: Cp(-C[1].hash_involution())
Cp[1]
sage: Cp(-C[1,2,3].hash_involution())
Cp[1,2,3]
sage: Cp(C[1,2,1,3].hash_involution())
Cp[1,2,3,1]
sage: all(C((-1)**x.length()*Cp[x].hash_involution()) == C[x] for x in W) # long time
True
```

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:

$$\begin{aligned}\overline{C'_w} &= C'_w \\ C'_w &= q^{-\ell(w)/2} \sum_{v \leq w} P_{v,w}(q) T_v\end{aligned}$$

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 then 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]
```

TESTS:

```
sage: R.<v> = LaurentPolynomialRing(QQ, 'v')
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: W = H.coxeter_group()
sage: T = H.T()
sage: C = H.C()
sage: Cp = H.Cp()
sage: all(Cp(T(Cp[x]))) == Cp[x] for x in W) # long time
True
sage: all(Cp(C(Cp[x]))) == Cp[x] for x in W) # long time
True
```

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 [I64].

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

```

TESTS:

```

sage: R.<v> = LaurentPolynomialRing(QQ, 'v')
sage: H = IwahoriHeckeAlgebra('A3', v**2)
sage: W = H.coxeter_group()
sage: T = H.T()
sage: C = H.C()
sage: Cp = H.Cp()
sage: all(T(C(T[x])) == T[x] for x in W) # long time
True
sage: all(T(Cp(T[x])) == T[x] for x in W) # long time
True

```

We check a property of the bar involution and R -polynomials:

```

sage: KL = KazhdanLusztigPolynomial(W, v)
sage: all(T[x].bar() == sum(v^(-2*y.length()) * KL.R(y, x).substitute(v=v^-2) * T[y] for y in T)
True

```

class Element (*M, x*)

Bases: `sage.combinat.free_module.CombinatorialFreeModuleElement`

A class for elements of an Iwahori-Hecke algebra in the T basis.

TESTS:

```

sage: R.<q> = QQ[]
sage: H = IwahoriHeckeAlgebra("B3", q).T()
sage: T1, T2, T3 = H.algebra_generators()
sage: T1+2*T2*T3
2*T[2, 3] + T[1]
sage: T1*T1
(q-1)*T[1] + q

sage: R.<q1,q2> = QQ[]
sage: H = IwahoriHeckeAlgebra("A2", q1, q2=q2).T(prefix="x")
sage: sum(H.algebra_generators())^2
x[1, 2] + x[2, 1] + (q1+q2)*x[1] + (q1+q2)*x[2] + (-2*q1*q2)

sage: H = IwahoriHeckeAlgebra("A2", q1, q2=q2).T(prefix="t")
sage: t1, t2 = H.algebra_generators()
sage: (t1-t2)^3
(q1^2-q1*q2+q2^2)*t[1] + (-q1^2+q1*q2-q2^2)*t[2]

sage: R.<q> = QQ[]
sage: H = IwahoriHeckeAlgebra("G2", q).T()
sage: [T1, T2] = H.algebra_generators()
sage: T1*T2*T1*T2*T1*T2 == T2*T1*T2*T1*T2*T1
True
sage: T1*T2*T1 == T2*T1*T2
False

sage: H = IwahoriHeckeAlgebra("A2", 1).T()
sage: [T1, T2] = H.algebra_generators()
sage: T1+T2
T[1] + T[2]

sage: -(T1+T2)
-T[1] - T[2]
sage: 1-T1
-T[1] + 1

sage: T1.parent()
Iwahori-Hecke algebra of type A2 in 1,-1 over Integer Ring 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

```

TESTS:

We check some alternative forms of input for inverting an element:

```

sage: R.<q> = LaurentPolynomialRing(QQ)
sage: H = IwahoriHeckeAlgebra("A2", q).T()
sage: T1,T2 = H.algebra_generators()
sage: ~ (T1*T2)
(q^-2)*T[2,1] + (q^-2-q^-1)*T[1] + (q^-2-q^-1)*T[2] + (q^-2-2*q^-1+1)
sage: (T1*T2)^(-1)
(q^-2)*T[2,1] + (q^-2-q^-1)*T[1] + (q^-2-q^-1)*T[2] + (q^-2-2*q^-1+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.**hash_involution_on_basis**(w)

Return the hash involution on the basis element $\text{self}[w]$.

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 $T_w \mapsto -1^{\ell(w)}(q_1 q_2)^{-\ell(w)} T_w$, 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
```

```
sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebraT(W, q1, q2=-1,
base_ring=None, prefix='T')
```

TESTS:

```
sage: H = IwahoriHeckeAlgebraT("A2", 1)
doctest...: DeprecationWarning: this class is deprecated. Use IwahoriHeckeAlgebra().T instead
See http://trac.sagemath.org/14261 for details.
```

```
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 $\mathbf{Z}[u, u^{-1}, v, v^{-1}]$ in two variables and has quadratic relations:

$$(T_r - u)(T_r + v^2/u) = 0.$$

The point of these relations is that the product of the two parameters is v^2 which is a square in $\mathbf{Z}[u, u^{-1}, v, v^{-1}]$. Consequently, the Kazhdan-Lusztig bases are defined for this algebra.

More generally, if we have a Iwahori-Hecke algebra with two parameters which has quadratic relations of the form:

$$(T_r - q_1)(T_r - q_2) = 0$$

where $-q_1q_2$ is a square then the Kazhdan-Lusztig bases are well-defined for this algebra. Moreover, these bases be computed by specialization from the generic Iwahori-Hecke algebra using the specialization which sends $u \mapsto q_1$ and $v \mapsto \sqrt{-q_1q_2}$, so that $v^2/u \mapsto -q_2$.

For example, if $q_1 = q = Q^2$ and $q_2 = -1$ then $u \mapsto q$ and $v \mapsto \sqrt{q} = Q$; this is the standard presentation of the Iwahori-Hecke algebra with $(T_r - q)(T_r + 1) = 0$. On the other hand, when $q_1 = q$ and $q_2 = -q^{-1}$ then $u \mapsto q$ and $v \mapsto 1$. This is the normalized presentation with $(T_r - v)(T_r + v^{-1}) = 0$.

Warning: This class uses non-standard parameters for the Iwahori-Hecke algebra and are related to the standard parameters by an outer automorphism that is non-trivial on the T -basis.

```
class C (IHAlgebra, prefix=None)
```

```
Bases: 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 $(-q_1q_1)^{\ell(w)}T_w$ and C_w to $(-1)^{\ell(w)}C'_w$. Therefore, if

$$T_w = \sum_v a_{vw} C'_v$$

then

$$T_w = (-q_1 q_2)^{\ell(w)} \left(\sum_v a_{vw} C'_v \right)^{\#} = \sum_v (-1)^{\ell(v)} \overline{a_{vw}} C_v$$

Note that we cannot just apply `hash_involution()` here because this involution always returns the answer with respect to the same basis.

EXAMPLES:

```
sage: H = sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_nonstandard("A2")
sage: s1,s2 = H.coxeter_group().simple_reflections()
sage: T = H.T()
sage: C = H.C()
sage: T.to_C_basis(s1)
v*T[1] + u
sage: C(T(s1))
v*C[1] + u
sage: C(T( C[1] ))
C[1]
sage: C(T(s1*s2)+T(s1)+T(s2)+1)
v^2*C[1,2] + (u*v+v)*C[1] + (u*v+v)*C[2] + (u^2+2*u+1)
sage: C(T(s1*s2*s1))
v^3*C[1,2,1] + u*v^2*C[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)`

Returns 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)
```


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 [Schiffmann], 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]

```

TESTS:

The coefficients are actually Laurent polynomials in general, so we don't have to work over the fraction field of $\mathbb{Z}[q]$. This didn't work before [trac ticket #15345](#):

```

sage: R.<q> = LaurentPolynomialRing(ZZ)
sage: H = HallAlgebra(R, q)
sage: I = H.monomial_basis()
sage: hi = H(I[2,1]); hi
H[2, 1] + (1+q+q^2)*H[1, 1, 1]
sage: hi.parent() is H

```

```

True
sage: h22 = H[2]*H[2]; h22
H[4] - (1-q)*H[3, 1] + (q+q^2)*H[2, 2]
sage: h22.parent() is H
True
sage: e = SymmetricFunctions(R).e()
sage: e(H[1, 1, 1])
(q^-3)*e[3]

```

REFERENCES:

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.

TESTS:

```

sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True

```

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 $\lambda = (l^1, 2^{l^2}, \dots, m^{l^m})$.

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 [Schiffmann] 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 $\mathbb{Z}[q]$ for our initial example:

```
sage: R = PolynomialRing(ZZ, 'q').fraction_field()
sage: q = R.gen()
sage: H = HallAlgebra(R, q)
sage: I = H.monomial_basis()
```

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 $\mathbb{Z}[q, q^{-1}]$:

```
sage: R.<q> = LaurentPolynomialRing(ZZ)
sage: H = HallAlgebra(R, q)
sage: I = H.monomial_basis()
sage: all(H(I(H[p])) == H[p] for i in range(6) for p in Partitions(i)) # long time
True
sage: all(I(H(I[p])) == I[p] for i in range(6) for p in Partitions(i)) # long time
True
```

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.

TESTS:

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']; f
B['a'] + 3*B['c']
sage: f == loads(dumps(f))
True
```

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
```


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 [Reuten1993].

INPUT:

- *R* – ring
- *names* – names of the generators (string or an alphabet)

REFERENCES:

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
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
sage: T.algebra_generators()
(S[word: a], S[word: b], S[word: c], S[word: d])
```

TESTS:

We check conversion between the bases:

```
sage: A = ShuffleAlgebra(QQ, 'ab')
sage: S = A.dual_pbw_basis()
sage: W = Words('ab', 5)
sage: all(S(A(S(w))) == S(w) for w in W)
True
sage: all(A(S(A(w))) == A(w) for w in W)
True
```

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]
```

TESTS:

Check that multiplication agrees with the multiplication in the shuffle algebra:

```
sage: A = ShuffleAlgebra(QQ, 'ab')
sage: S = A.dual_pbw_basis()
sage: a, b = S.gens()
sage: A(a*b)
B[word: ab] + B[word: ba]
sage: A(a*b*a)
2*B[word: aab] + 2*B[word: aba] + 2*B[word: baa]
sage: S(A(a)*A(b)*A(a)) == a*b*a
True
```

`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']
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
```

```
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])
```

coproduct(S)

Return the coproduct of the series S.

EXAMPLES:

```
sage: F = ShuffleAlgebra(QQ, 'ab')
sage: S = F.an_element(); S
2*B[word: ] + 2*B[word: a] + 3*B[word: b]
sage: F.coproduct(S)
2*B[word: ] # B[word: ] + 2*B[word: ] # B[word: a] + 3*B[word: ] # B[word: b]
+ 2*B[word: a] # B[word: ] + 3*B[word: b] # B[word: ]
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
2*B[word: ] + 2*B[word: a] + 3*B[word: b]
sage: F.counit(S)
2
```

dual_pbw_basis()

Return the dual PBW of self.

EXAMPLES:

```
sage: A = ShuffleAlgebra(QQ, 'ab')
sage: A.dual_pbw_basis()
The dual Poincare-Birkhoff-Witt basis of Shuffle Algebra on 2 generators ['a', 'b'] over Rat
```

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])
```

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:

- w1, w2 – Basis elements

EXAMPLES:

```
sage: A = ShuffleAlgebra(QQ, 'abc')
sage: W = A.basis().keys()
sage: A.product_on_basis(W("acb"), W("cba"))
B[word: acbacb] + B[word: acbcab] + 2*B[word: acbcba] + 2*B[word: accbab] + 4*B[word: accbba]

sage: (a,b,c) = A.algebra_generators()
sage: a * (1-b)^2 * c
2*B[word: abbc] - 2*B[word: abc] + 2*B[word: abcb] + B[word: ac] - 2*B[word: acb] + 2*B[word: acbb]
```

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'}
```


THE STEENROD ALGEBRA

AUTHORS:

- John H. Palmieri (2008-07-30): version 0.9: Initial implementation.
- John H. Palmieri (2010-06-30): version 1.0: Implemented sub-Hopf algebras and profile functions; direct multiplication of admissible sequences (rather than conversion to the Milnor basis); implemented the Steenrod algebra using `CombinatorialFreeModule`; improved the test suite.

This module defines the mod p Steenrod algebra \mathcal{A}_p , some of its properties, and ways to define elements of it.

From a topological point of view, \mathcal{A}_p is the algebra of stable cohomology operations on mod p cohomology; thus for any topological space X , its mod p cohomology algebra $H^*(X, \mathbf{F}_p)$ is a module over \mathcal{A}_p .

From an algebraic point of view, \mathcal{A}_p is an \mathbf{F}_p -algebra; when $p = 2$, it is generated by elements Sq^i for $i \geq 0$ (the *Steenrod squares*), and when p is odd, it is generated by elements \mathcal{P}^i for $i \geq 0$ (the *Steenrod reduced p th powers*) along with an element β (the *mod p Bockstein*). The Steenrod algebra is graded: Sq^i is in degree i for each i , β is in degree 1, and \mathcal{P}^i is in degree $2(p-1)i$.

The unit element is Sq^0 when $p = 2$ and \mathcal{P}^0 when p is odd. The generating elements also satisfy the *Adem relations*. At the prime 2, these have the form

$$\text{Sq}^a \text{Sq}^b = \sum_{c=0}^{\lfloor a/2 \rfloor} \binom{b-c-1}{a-2c} \text{Sq}^{a+b-c} \text{Sq}^c.$$

At odd primes, they are a bit more complicated; see Steenrod and Epstein [SE] 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 [Mil] 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 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 $Sq^{m_1} Sq^{m_2} \dots Sq^{m_t}$ 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 $w(m, k) = Sq^{2^m(2^{k+1}-1)}$. 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 $w(m, k) = Sq^{2^m(2^{k+1}-1)}$. 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 $Q_k^m = Sq^{2^k} Sq^{2^{k+1}} \dots Sq^{2^m}$. 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 $X_k^m = Sq^{2^m} Sq^{2^{m-1}} \dots Sq^{2^k}$. 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 $Sq^{t_1} \dots Sq^{t_m}$ where for each i , we have $t_i \leq 2t_{i+1}$ and $2^i | t_{m-i}$.
- *P_t^s bases*. Let $p = 2$. For integers $s \geq 0$ and $t > 0$, the element P_t^s is the Milnor basis element $\mathcal{P}(0, \dots, 0, p^s, 0, \dots)$, with the nonzero entry in position t . To obtain a P_t^s -basis, for each set $\{P_{t_1}^{s_1}, \dots, P_{t_k}^{s_k}\}$ of (distinct) P_t^s ’s, one chooses an ordering and forms the monomials

$$(P_{t_1}^{s_1})^{i_1} \dots (P_{t_k}^{s_k})^{i_k}$$

for all exponents i_j with $0 < i_j < p$. When $p = 2$, the set of all such monomials then forms a basis, and when p is odd, if one multiplies each such monomial on the left by products of the form $Q_{e_1} Q_{e_2} \dots$ with $0 \leq e_1 < e_2 < \dots$, one obtains a basis.

Thus one gets a basis by choosing an ordering on each set of P_t^s ’s. There are infinitely many orderings possible, and we have implemented four of them:

- ‘rlex’: right lexicographic ordering
- ‘llex’: left lexicographic ordering
- ‘deg’: ordered by degree, which is the same as left lexicographic ordering on the pair $(s + t, t)$
- ‘revz’: left lexicographic ordering on the pair $(s + t, s)$, which is the reverse of the ordering used (on elements in the same degrees as the P_t^s ’s) in Wood’s Z basis: ‘revz’ stands for ‘reversed Z’. This is the default: ‘pst’ is the same as ‘pst_revz’.
- *Commutator bases*. Let $c_{i,1} = \mathcal{P}(p^i)$, let $c_{i,2} = [c_{i+1,1}, c_{i,1}]$, and inductively define $c_{i,k} = [c_{i+k-1,1}, c_{i,k-1}]$. Thus $c_{i,k}$ is a k -fold iterated commutator of the elements $\mathcal{P}(p^i), \dots, \mathcal{P}(p^{i+k-1})$. Note that $\dim c_{i,k} = \dim P_k^i$.
Commutator bases are obtained in much the same way as P_t^s -bases: for each set $\{c_{s_1, t_1}, \dots, c_{s_k, t_k}\}$ of (distinct) $c_{s,t}$ ’s, one chooses an ordering and forms the resulting monomials

$$c_{s_1, t_1}^{i_1} \dots c_{s_k, t_k}^{i_k}$$

for all exponents i_j with $0 < i_j < p$. When p is odd, one also needs to left-multiply by products of the Q_i ’s. As for P_t^s -bases, every ordering on each set of iterated commutators determines a basis, and the same four orderings have been defined for these bases as for the P_t^s bases: ‘rlex’, ‘llex’, ‘deg’, ‘revz’.

Sub-Hopf algebras of the Steenrod algebra

The sub-Hopf algebras of the Steenrod algebra have been classified. Milnor proved that at the prime 2, the dual of the Steenrod algebra A_* is isomorphic to a polynomial algebra

$$A_* \cong \mathbf{F}_2[\xi_1, \xi_2, \xi_3, \dots].$$

The Milnor basis is dual to the monomial basis. Furthermore, any sub-Hopf algebra corresponds to a quotient of this of the form

$$A_*/(\xi_1^{2^{e_1}}, \xi_2^{2^{e_2}}, \xi_3^{2^{e_3}}, \dots).$$

The list of exponents (e_1, e_2, \dots) may be considered a function e from the positive integers to the extended 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, 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 $Q_0^{\epsilon_0} Q_1^{\epsilon_1} \dots \mathcal{P}(r_1, r_2, \dots)$ 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 $Q_0^{\epsilon_0} Q_1^{\epsilon_1} \dots \mathcal{P}(r_1, r_2, \dots)$ 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()
{(5,): 1, (2, 1): 1}
sage: c.monomials()
[Sq(2,1), Sq(5)]
sage: c.support()
[(2, 1), (5,)]
sage: Adem = SteenrodAlgebra(basis='adem')
sage: (Adem.Sq(10) + Adem.Sq(9) * Adem.Sq(1)).monomials()
[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.

REFERENCES:

- [AM] J. F. Adams, and H. R. Margolis, "Sub-Hopf-algebras of the Steenrod algebra," Proc. Cambridge Philos. Soc. 76 (1974), 45-52.
- [Mil] J. W. Milnor, "The Steenrod algebra and its dual," Ann. of Math. (2) 67 (1958), 150-171.
- [Mon] K. G. Monks, "Change of basis, monomial relations, and P_t^s bases for the Steenrod algebra," J. Pure Appl. Algebra 125 (1998), no. 1-3, 235-260.
- [SE] N. E. Steenrod and D. B. A. Epstein, Cohomology operations, Ann. of Math. Stud. 50 (Princeton University Press, 1962).
- [Vo] V. Voevodsky, Reduced power operations in motivic cohomology, Publ. Math. Inst. Hautes Études Sci. No. 98 (2003), 1-57.

`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 $\text{Sq}(a, b, c, \dots)$.

INPUT:

- a, b, c, \dots - non-negative integers

OUTPUT: element of the Steenrod algebra

This returns the Milnor basis element $\text{Sq}(a, b, c, \dots)$.

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, ...]`, 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, ...]
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, ...]
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

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 [VO], 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
```

TESTS:

Testing unique parents:

```
sage: S0 = SteenrodAlgebra(2)
sage: S1 = SteenrodAlgebra(2)
sage: S0 is S1
True
sage: S2 = SteenrodAlgebra(2, basis='adem')
sage: S0 is S2
False
sage: S0 == S2
False
sage: A1 = SteenrodAlgebra(profile=[2,1])
sage: B1 = SteenrodAlgebra(profile=[2,1,0,0])
sage: A1 is B1
True
```

```
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()
{(5,): 1, (2, 1): 1}
sage: c.monomials()
[Sq(2,1), Sq(5)]
sage: 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 (*basis*)

Representation of element with respect to basis.

INPUT:

- *basis* - string, basis in which to work.

OUTPUT: Representation of self in given basis

Warning: Deprecated (December 2010). Use `change_basis()` instead.

EXAMPLES:

```
sage: c = Sq(2) * Sq(1)
sage: c.basis('milnor')
doctest:...: DeprecationWarning: The .basis() method is deprecated. Use .change_basis()
See http://trac.sagemath.org/10052 for details.
Sq(0,1) + Sq(3)
```

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

```

TESTS:

```

sage: all([x.degree() == 10 for x in SteenrodAlgebra(basis='woody').basis(10)])
True
sage: all([x.degree() == 11 for x in SteenrodAlgebra(basis='woodz').basis(11)])
True
sage: all([x.degree() == x.milnor().degree() for x in SteenrodAlgebra(basis='wall').basis(12)])
True
sage: a = SteenrodAlgebra(basis='pst').basis(10)[0]
sage: a.degree() == a.change_basis('arnonc').degree()
True
sage: b = SteenrodAlgebra(basis='comm').basis(12)[1]
sage: b.degree() == b.change_basis('adem').change_basis('arnona').degree()
True

```



```

sage: all([x.degree() == 9 for x in SteenrodAlgebra(basis='comm').basis(9)])
True
sage: all([x.degree() == 8 for x in SteenrodAlgebra(basis='adem').basis(8)])
True
sage: all([x.degree() == 7 for x in SteenrodAlgebra(basis='milnor').basis(7)])
True
sage: all([x.degree() == 24 for x in SteenrodAlgebra(p=3).basis(24)])
True
sage: all([x.degree() == 40 for x in SteenrodAlgebra(p=5, basis='serre-cartan').basis(40)])
True

```

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 $Q_0^{e_0} Q_1^{e_1} \dots P(r_1, r_2, \dots)$ 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 [Kra] for the proofs of these assertions.

REFERENCES:

- [Kra] D. Kraines, “On excess in the Milnor basis,” Bull. London Math. Soc. 3 (1971), 363-365.

EXAMPLES:

```

sage: a = Sq(1, 2, 3)
sage: a.excess()
6
sage: (Sq(0, 0, 1) + Sq(4, 1) + Sq(7)).excess()
1
sage: [m.excess() for m in (Sq(0, 0, 1) + Sq(4, 1) + Sq(7)).monomials()]
[1, 5, 7]
sage: [m for m in (Sq(0, 0, 1) + Sq(4, 1) + Sq(7)).monomials()]
[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]
```

TESTS:

These all test changing bases and printing in various bases:

```
sage: A = SteenrodAlgebra(basis='milnor')
sage: [A.Sq(n) for n in range(9) if not A.Sq(n).is_decomposable()]
[1, Sq(1), Sq(2), Sq(4), Sq(8)]
sage: A = SteenrodAlgebra(basis='wall_long')
sage: [A.Sq(n) for n in range(9) if not A.Sq(n).is_decomposable()]
[1, Sq^1, Sq^2, Sq^4, Sq^8]
sage: A = SteenrodAlgebra(basis='arnona_long')
sage: [A.Sq(n) for n in range(9) if not A.Sq(n).is_decomposable()]
[1, Sq^1, Sq^2, Sq^4, Sq^8]
sage: A = SteenrodAlgebra(basis='woodz')
sage: [A.Sq(n) for n in range(20) if not A.Sq(n).is_decomposable()] # long time
[1, Sq^1, Sq^2, Sq^4, Sq^8, Sq^16]
sage: A = SteenrodAlgebra(basis='comm_long')
sage: [A.Sq(n) for n in range(25) if not A.Sq(n).is_decomposable()] # long time
[1, s_1, s_2, s_4, s_8, s_16]
```

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 [May], 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.)

REFERENCES:

- [May]: J. P. May, "The cohomology of restricted Lie algebras and of Hopf algebras; application to the Steenrod algebra." Thesis, Princeton Univ., 1964.

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 [Wall], 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 $j \leq k \leq i + j - 1$.

REFERENCES:

- [Wall]: C. T. C. Wall, "Generators and relations for the Steenrod algebra," Ann. of Math. (2) **72** (1960), 429-444.

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)
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
```

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)
```

TESTS:

```
sage: Milnor = SteenrodAlgebra()
sage: all([x.antipode().antipode() == x for x in Milnor.basis(11)]) # long time
True
sage: A5 = SteenrodAlgebra(p=5, basis='adem')
sage: all([x.antipode().antipode() == x for x in A5.basis(25)])
True
sage: H = SteenrodAlgebra(profile=[2,2,1])
sage: H.Sq(1,2).antipode() in H
True
```

`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 doesn't 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)
sage: for (idx,a) in zip((1,..,9),A7.basis()):
...     print 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),

```

`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 - 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
```

TESTS:

```
sage: all([A.coproduct_on_basis((n,1), algorithm='milnor') == A.coproduct_on_basis((n,1), al
True
sage: A7 = SteenrodAlgebra(p=7, basis='adem')
sage: all([A7.coproduct_on_basis((0,n,1), algorithm='milnor') == A7.coproduct_on_basis((0,n,
True
```

`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 $\text{Sq}(2^i)$ 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
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
```

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
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')
```

TESTS:

The following sort of thing is also tested by the function `steenrod_basis_error_check`:

```
sage: H = SteenrodAlgebra(p=5, profile=[[2,1], [2,2,2]])
sage: G = SteenrodAlgebra(p=5, profile=[[2,1], [2,2,2]], basis='pst')
sage: max([H[n].dimension() - G[n].dimension() for n in range(100)])
0
```

`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=[[1], [2, 2]]).dimension()
4
sage: SteenrodAlgebra(p=5, profile=[[1], [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:

• $t1, t2$ – 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)
```

TESTS:

```
sage: all([Adem(Milnor.Sq(n) ** 3)._repr_() == (Adem.Sq(n) ** 3)._repr_() for n in range(10)])
True
sage: Wall = SteenrodAlgebra(basis='wall')
sage: Wall(Adem.Sq(4,4) * Milnor.Sq(4)) == Adem(Wall.Sq(4,4) * Milnor.Sq(4))
True

sage: A3 = SteenrodAlgebra(p=3, basis='adem')
sage: M3 = SteenrodAlgebra(p=3, basis='milnor')
sage: all([A3(M3.P(n) * M3.Q(0) * M3.P(n))._repr_() == (A3.P(n) * A3.Q(0) * A3.P(n))._repr_() for n in range(10)])
True

sage: EA = SteenrodAlgebra(generic=True)
sage: EA.product_on_basis(((1, 3), (2, 1)), ((2, ), (0, 0, 1)))
Q_1 Q_2 Q_3 P(2,1,1)

sage: EA2 = SteenrodAlgebra(basis='serre-cartan', generic=True)
sage: EA2.product_on_basis((1, 2, 0, 1, 0), (1, 2, 0, 1, 0))
beta P^4 P^2 beta + beta P^5 beta P^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))).top_class()
Q_1 Q_2 Q_3 Q_4 P(8,8,2)
```

TESTS:

```
sage: SteenrodAlgebra(2,profile=(3,2,1),basis='pst').top_class()
P^0_1 P^0_2 P^1_1 P^0_3 P^1_2 P^2_1
sage: SteenrodAlgebra(5,profile=((0,),(2,1,2,2))).top_class()
Q_0 Q_2 Q_3
sage: SteenrodAlgebra(5).top_class()
Traceback (most recent call last):
...
ValueError: the algebra is not finite dimensional
```

Currently, we create the top class in the Milnor basis version and transform this result back into the requested basis. This approach is easy to implement but far from optimal for the 'pst' basis. Occasionally, it also gives an awkward leading coefficient:

```
sage: SteenrodAlgebra(3,profile=((2,1),(1,2,2)),basis='pst').top_class()
2 Q_1 Q_2 (P^0_1)^2 (P^0_2)^2 (P^1_1)^2
```

TESTS:

```
sage: A=SteenrodAlgebra(2,profile=(3,2,1),basis='pst')
sage: A.top_class().parent() is A
True
```

```
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.

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 [M] and Wood [W] for more information about them. For commutator bases, see the preprint by Palmieri and Zhang [PZ].

- ‘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`.

REFERENCES:

- [M] K. G. Monks, “Change of basis, monomial relations, and P_t^s bases for the Steenrod algebra,” J. Pure Appl. Algebra 125 (1998), no. 1-3, 235-260.
- [PZ] J. H. Palmieri and J. J. Zhang, “Commutators in the Steenrod algebra,” preprint (2008)
- [W] R. M. W. Wood, “Problems in the Steenrod algebra,” Bull. London Math. Soc. 30 (1998), no. 5, 449-517.

`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, 1))),))

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, basis, 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 1 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 1 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 1 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 1 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 1 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 1 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 1 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 1 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 1 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 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,
                                                                    basis,
                                                                    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 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, 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]]

```

'1' may have repeated elements. If 'no_repeats' is False, this has no effect. If 'no_repeats' is True, and if the repeated elements appear consecutively in '1', then each element may be used only as many times as it appears in '1':

```
sage: 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])
[[6, 4], [6, 2, 2], [4, 4, 2], [4, 2, 2, 2], [2, 2, 2, 2, 2]]
```

```
sage.algebras.steenrod.steenrod_algebra_bases.serre_cartan_basis(n, p=2,
                                                                    bound=1,
                                                                    **kws)
```

Serre-Cartan basis in dimension n .

INPUT:

- n - 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 $Sq^{m_1}Sq^{m_2}...Sq^{m_t}$ with $m_i \geq 2m_{i+1}$ for each i . At odd primes, it consists of monomials $\beta^{e_0}P^{s_1}\beta^{e_1}P^{s_2}...P^{s_k}\beta^{e_k}$ with each e_i either 0 or 1, $s_i \geq ps_{i+1} + e_i$ for all i , and $s_k \geq 1$.

EXAMPLES:

```
sage: from sage.algebras.steenrod.steenrod_algebra_bases import serre_cartan_basis
sage: serre_cartan_basis(7)
((7,), (6, 1), (4, 2, 1), (5, 2))
sage: serre_cartan_basis(13,3)
((1, 3, 0), (0, 3, 1))
sage: serre_cartan_basis(50,5)
((1, 5, 0, 1, 1), (1, 6, 1))
```

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,
                                                                    **kws)
```

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,
                                                                           **kws)
```

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 ξ_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]
```


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, latex=False,
                                                                    p=2)
```

String representation of element of Arnon's A basis.

This is used by the `_repr_` and `_latex_` methods.

INPUT:

- `mono` - tuple of pairs of 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, latex=False,
                                                                        generic=False)
```

Alternate string representation of element of a commutator basis.

Okay in low dimensions, but gets unwieldy as the dimension increases.

INPUT:

- `mono` - tuple of pairs of integers (s,t) with $s \geq 0, t > 0$
- `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 $s \geq 0, t > 0$
- `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 $1,2,\dots,\text{len}(m)$

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

$$e(r) \geq \min(e(r-i) - i, e(i)) \text{ for all } 0 < i < r.$$

At odd primes, a pair of profile functions e and k are valid if they satisfy

$$e(r) \geq \min(e(r-i) - i, e(i)) \text{ for all } 0 < i < r.$$

$$\text{• if } k(i+j) = 1, \text{ then either } e(i) \leq j \text{ or } k(j) = 1 \text{ for all } i \geq 1, j \geq 0.$$

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, preci-
                                                                    sion=None,
                                                                    trunca-
                                                                    tion_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, latex=False, generic=False)
```

String representation of element of the Serre-Cartan basis.

This is used by the `_repr_` and `_latex_` methods.

INPUT:

- `mono` - tuple of positive integers $(a,b,c,...)$ when *generic* = *False*, or tuple $(e_0, n_1, e_1, n_2, ...)$ when *generic* = *True*, where each e_i is 0 or 1, and each n_i 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,
                                                                    la-
                                                                    tex=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'
```


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 [Mil] for proofs, etc.

To multiply Milnor basis elements $Sq(r_1, r_2, \dots)$ and $Sq(s_1, s_2, \dots)$ 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 $Sq(2)$ and $Sq(1, 1)$, 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

$$Sq(r_1, r_2, \dots) Sq(s_1, s_2, \dots) = \sum Sq(t_1, t_2, \dots)$$

The function `milnor_multiplication()` takes as input two tuples of non-negative integers, r and s , which represent $Sq(r) = Sq(r_1, r_2, \dots)$ and $Sq(s) = Sq(s_1, s_2, \dots)$; 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 $Sq(t)$ 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 [Mil] for proofs.

Fix an odd prime p . There are three steps to multiply Milnor basis elements $Q_{f_1}Q_{f_2}\dots\mathcal{P}(q_1, q_2, \dots)$ and $Q_{g_1}Q_{g_2}\dots\mathcal{P}(s_1, s_2, \dots)$: first, use the formula

$$\mathcal{P}(q_1, q_2, \dots)Q_k = Q_k\mathcal{P}(q_1, q_2, \dots) + Q_{k+1}\mathcal{P}(q_1 - p^k, q_2, \dots) + Q_{k+2}\mathcal{P}(q_1, q_2 - p^k, \dots) + \dots$$

Second, use the fact that the Q_k 's form an exterior algebra: $Q_k^2 = 0$ for all k , and if $i \neq j$, then Q_i and Q_j anticommute: $Q_iQ_j = -Q_jQ_i$. After these two steps, the product is a linear combination of terms of the form

$$Q_{e_1}Q_{e_2}\dots\mathcal{P}(r_1, r_2, \dots)\mathcal{P}(s_1, s_2, \dots).$$

Finally, use Milnor matrices to multiply the pairs of $\mathcal{P}(\dots)$ terms, as at the prime 2: form all possible matrices M with rows and columns indexed starting at 0, with position (0,0) deleted (or ignored), with s_i equal to the sum of column i for each i , and with r_j equal to the weighted sum of row j : elements from column i are multiplied by p^i . For example when $p = 5$, to multiply $\mathcal{P}(5)$ and $\mathcal{P}(1, 1)$, form the matrices

$$\begin{vmatrix} * & 1 & 1 \\ 5 & 0 & 0 \end{vmatrix} \quad \text{and} \quad \begin{vmatrix} * & 0 & 1 \\ 0 & 1 & 0 \end{vmatrix}$$

For each such matrix M , compute a multinomial coefficient, mod p : for each diagonal $\{m_{ij} : i + j = n\}$, compute $(\sum m_{i,j}!)/(m_{0,n}!m_{1,n-1}!\dots m_{n,0}!)$. Multiply these together for all n .

Now, for each matrix with nonzero multinomial coefficient b_M , let t_n be the sum of the n -th diagonal in the matrix; then

$$\mathcal{P}(r_1, r_2, \dots)\mathcal{P}(s_1, s_2, \dots) = \sum b_M \mathcal{P}(t_1, t_2, \dots)$$

For example when $p = 5$, we have

$$\mathcal{P}(5)\mathcal{P}(1, 1) = \mathcal{P}(6, 1) + 2\mathcal{P}(0, 2).$$

The function `milnor_multiplication()` takes as input two pairs of tuples of 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 p th power operations and Bocksteins into a dictionary representing a sum of admissible monomials.

REFERENCES:

- [Mil] J. W. Milnor, “The Steenrod algebra and its dual”, Ann. of Math. (2) 67 (1958), 150–171.
- [SE] N. E. Steenrod, “Cohomology operations (Lectures by N. E. Steenrod written and revised by D. B. A. Epstein)”. Annals of Mathematics Studies, No. 50, 1962, Princeton University Press.

`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)
{(1, 2, 0): 1, (0, 2, 1): 1}
sage: adem(1,1,2, p=5)
{(1, 3, 0): 2, (0, 3, 1): 1}
```

`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))
{(6, 1): 1, (0, 3): 1, (2, 0, 1): 1}
sage: milnor_multiplication((2,4), (0,1))
{(2, 5): 1, (2, 0, 0, 1): 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)
{((, (4, 1)): 1, ((, (8,)): 1, ((, (0, 2)): 1}

```

These examples correspond to the following product computations:

$$\begin{aligned}
 p = 5: \quad Q_0 Q_2 P(5) Q_1 P(1) &= 4 Q_0 Q_1 Q_2 P(0, 1) + 4 Q_0 Q_1 Q_2 P(6) \\
 p = 7: \quad (Q_0 Q_2 Q_4)(Q_1 Q_3) &= 6 Q_0 Q_1 Q_2 Q_3 Q_4 \\
 p = 7: \quad (Q_0 Q_2 Q_4)(Q_1 Q_5) &= Q_0 Q_1 Q_2 Q_3 Q_5 \\
 p = 3: \quad P(6) P(2) &= P(0, 2) + P(4, 1) + P(8)
 \end{aligned}$$

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(1,2); 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,

```

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])
463
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
```

INDICES AND TABLES

- [Index](#)
- [Module Index](#)
- [Search Page](#)

BIBLIOGRAPHY

- [I64] N. Iwahori, On the structure of a Hecke ring of a Chevalley group over a finite field, J. Fac. Sci. Univ. Tokyo Sect. I, 10 (1964), 215–236 (1964). [MathSciNet MR0165016](#)
- [HKP] T. J. Haines, R. E. Kottwitz, A. Prasad, Iwahori-Hecke Algebras, J. Ramanujan Math. Soc., 25 (2010), 113–145. [Arxiv 0309168v3](#) [MathSciNet MR2642451](#)
- [J87] V. Jones, Hecke algebra representations of braid groups and link polynomials. Ann. of Math. (2) 126 (1987), no. 2, 335–388. [doi:10.2307/1971403](#) [MathSciNet MR0908150](#)
- [Schiffmann] Oliver Schiffmann. *Lectures on Hall algebras*. [Arxiv 0611617v2](#).
- [Reuten1993] C. Reutenauer. *Free Lie Algebras*. Number 7 in London Math. Soc. Monogr. (N.S.). Oxford University Press. (1993).

PYTHON MODULE INDEX

a

`sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra`, 1
`sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element`, 9
`sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_ideal`, 13
`sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism`, 15
`sage.algebras.free_algebra`, 17
`sage.algebras.free_algebra_element`, 29
`sage.algebras.free_algebra_quotient`, 63
`sage.algebras.free_algebra_quotient_element`, 67
`sage.algebras.group_algebra_new`, 69
`sage.algebras.hall_algebra`, 93
`sage.algebras.iwahori_hecke_algebra`, 75
`sage.algebras.letterplace.free_algebra_element_letterplace`, 43
`sage.algebras.letterplace.free_algebra_letterplace`, 31
`sage.algebras.letterplace.letterplace_ideal`, 53
`sage.algebras.shuffle_algebra`, 101
`sage.algebras.steenrod.steenrod_algebra`, 109
`sage.algebras.steenrod.steenrod_algebra_bases`, 145
`sage.algebras.steenrod.steenrod_algebra_misc`, 157
`sage.algebras.steenrod.steenrod_algebra_mult`, 169

INDEX

A

`a_realization()` (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra method), 88
`AA()` (in module sage.algebras.steenrod.steenrod_algebra), 115
`additive_order()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), 122
`adem()` (in module sage.algebras.steenrod.steenrod_algebra_mult), 171
`algebra_generators()` (sage.algebras.free_algebra.FreeAlgebra_generic method), 21
`algebra_generators()` (sage.algebras.free_algebra.PBWBasisOffFreeAlgebra method), 26
`algebra_generators()` (sage.algebras.group_algebra_new.GroupAlgebra method), 72
`algebra_generators()` (sage.algebras.shuffle_algebra.DualPBWBasis method), 102
`algebra_generators()` (sage.algebras.shuffle_algebra.ShuffleAlgebra method), 105
`algebra_generators()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 130
`an_element()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 131
`antipode_on_basis()` (sage.algebras.hall_algebra.HallAlgebra method), 95
`antipode_on_basis()` (sage.algebras.hall_algebra.HallAlgebraMonomials method), 99
`antipode_on_basis()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 131
`arnonA_long_mono_to_string()` (in module sage.algebras.steenrod.steenrod_algebra_misc), 157
`arnonA_mono_to_string()` (in module sage.algebras.steenrod.steenrod_algebra_misc), 158
`arnonC_basis()` (in module sage.algebras.steenrod.steenrod_algebra_bases), 146
`atomic_basis()` (in module sage.algebras.steenrod.steenrod_algebra_bases), 147
`atomic_basis_odd()` (in module sage.algebras.steenrod.steenrod_algebra_bases), 148

B

`bar_on_basis()` (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T method), 85
`base_extend()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 1
`basis()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 2
`basis()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 132
`basis()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), 122
`basis_matrix()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_ideal.FiniteDimensionalAlgebraIdeal method), 13
`basis_name()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 133
`basis_name()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), 122
`binomial_mod2()` (in module sage.algebras.steenrod.steenrod_algebra_mult), 172
`binomial_modp()` (in module sage.algebras.steenrod.steenrod_algebra_mult), 172

C

`cardinality()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 2

`cartan_type()` (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra method), 88

`change_basis()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), 123

`characteristic_polynomial()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebra method), 9

`comm_long_mono_to_string()` (in module sage.algebras.steenrod.steenrod_algebra_misc), 158

`comm_mono_to_string()` (in module sage.algebras.steenrod.steenrod_algebra_misc), 159

`commutative_ring()` (sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace method), 33

`construction()` (sage.algebras.group_algebra_new.GroupAlgebra method), 72

`convert_from_milnor_matrix()` (in module sage.algebras.steenrod.steenrod_algebra_bases), 148

`convert_perm()` (in module sage.algebras.steenrod.steenrod_algebra_misc), 159

`convert_to_milnor_matrix()` (in module sage.algebras.steenrod.steenrod_algebra_bases), 149

`coproduct()` (sage.algebras.shuffle_algebra.ShuffleAlgebra method), 105

`coproduct()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 133

`coproduct()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), 123

`coproduct_on_basis()` (sage.algebras.hall_algebra.HallAlgebra method), 96

`coproduct_on_basis()` (sage.algebras.hall_algebra.HallAlgebraMonomials method), 99

`coproduct_on_basis()` (sage.algebras.shuffle_algebra.ShuffleAlgebra method), 105

`coproduct_on_basis()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 133

`counit()` (sage.algebras.hall_algebra.HallAlgebra method), 96

`counit()` (sage.algebras.hall_algebra.HallAlgebraMonomials method), 99

`counit()` (sage.algebras.shuffle_algebra.ShuffleAlgebra method), 105

`counit_on_basis()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 134

`coxeter_group()` (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra method), 88

`create_key()` (sage.algebras.free_algebra.FreeAlgebraFactory method), 20

`create_object()` (sage.algebras.free_algebra.FreeAlgebraFactory method), 20

`current_ring()` (sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace method), 33

D

`degbound()` (sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace method), 33

`degree()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 2

`degree()` (sage.algebras.letterplace.free_algebra_element_letterplace.FreeAlgebraElement_letterplace method), 43

`degree()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), 124

`degree_on_basis()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 134

`dimension()` (sage.algebras.free_algebra_quotient.FreeAlgebraQuotient method), 64

`dimension()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 135

`dual_pbw_basis()` (sage.algebras.shuffle_algebra.ShuffleAlgebra method), 106

`DualPBWBasis` (class in sage.algebras.shuffle_algebra), 101

`DualPBWBasis.Element` (class in sage.algebras.shuffle_algebra), 102

E

`Element` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra attribute), 1

`Element` (sage.algebras.free_algebra.FreeAlgebra_generic attribute), 21

`Element` (sage.algebras.free_algebra_quotient.FreeAlgebraQuotient attribute), 64

`excess()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), 125

`expand()` (sage.algebras.free_algebra.PBWBasisOffFreeAlgebra.Element method), 26

[expand\(\)](#) (sage.algebras.shuffle_algebra.DualPBWBasis.Element method), [102](#)
[expansion\(\)](#) (sage.algebras.free_algebra.PBWBasisOfFreeAlgebra method), [26](#)
[expansion\(\)](#) (sage.algebras.shuffle_algebra.DualPBWBasis method), [102](#)
[expansion_on_basis\(\)](#) (sage.algebras.shuffle_algebra.DualPBWBasis method), [102](#)

F

[FiniteDimensionalAlgebra](#) (class in sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra), [1](#)
[FiniteDimensionalAlgebraElement](#) (class in sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element), [9](#)
[FiniteDimensionalAlgebraHomset](#) (class in sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism), [15](#)
[FiniteDimensionalAlgebraIdeal](#) (class in sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_ideal), [13](#)
[FiniteDimensionalAlgebraMorphism](#) (class in sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism), [15](#)
[free_algebra\(\)](#) (sage.algebras.free_algebra.PBWBasisOfFreeAlgebra method), [27](#)
[free_algebra\(\)](#) (sage.algebras.free_algebra_quotient.FreeAlgebraQuotient method), [64](#)
[FreeAlgebra_generic](#) (class in sage.algebras.free_algebra), [20](#)
[FreeAlgebra_letterplace](#) (class in sage.algebras.letterplace.free_algebra_letterplace), [32](#)
[FreeAlgebraElement](#) (class in sage.algebras.free_algebra_element), [29](#)
[FreeAlgebraElement_letterplace](#) (class in sage.algebras.letterplace.free_algebra_element_letterplace), [43](#)
[FreeAlgebraFactory](#) (class in sage.algebras.free_algebra), [18](#)
[FreeAlgebraQuotient](#) (class in sage.algebras.free_algebra_quotient), [63](#)
[FreeAlgebraQuotientElement](#) (class in sage.algebras.free_algebra_quotient_element), [67](#)
[from_base_ring\(\)](#) (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), [2](#)

G

[g_algebra\(\)](#) (sage.algebras.free_algebra.FreeAlgebra_generic method), [21](#)
[gen\(\)](#) (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), [2](#)
[gen\(\)](#) (sage.algebras.free_algebra.FreeAlgebra_generic method), [22](#)
[gen\(\)](#) (sage.algebras.free_algebra.PBWBasisOfFreeAlgebra method), [27](#)
[gen\(\)](#) (sage.algebras.free_algebra_quotient.FreeAlgebraQuotient method), [64](#)
[gen\(\)](#) (sage.algebras.group_algebra_new.GroupAlgebra method), [72](#)
[gen\(\)](#) (sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace method), [34](#)
[gen\(\)](#) (sage.algebras.shuffle_algebra.DualPBWBasis method), [102](#)
[gen\(\)](#) (sage.algebras.shuffle_algebra.ShuffleAlgebra method), [106](#)
[gen\(\)](#) (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), [135](#)
[generator_degrees\(\)](#) (sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace method), [34](#)
[gens\(\)](#) (sage.algebras.free_algebra.FreeAlgebra_generic method), [22](#)
[gens\(\)](#) (sage.algebras.free_algebra.PBWBasisOfFreeAlgebra method), [27](#)
[gens\(\)](#) (sage.algebras.group_algebra_new.GroupAlgebra method), [72](#)
[gens\(\)](#) (sage.algebras.shuffle_algebra.DualPBWBasis method), [103](#)
[gens\(\)](#) (sage.algebras.shuffle_algebra.ShuffleAlgebra method), [106](#)
[gens\(\)](#) (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), [136](#)
[get_basis_name\(\)](#) (in module sage.algebras.steenrod.steenrod_algebra_misc), [159](#)
[groebner_basis\(\)](#) (sage.algebras.letterplace.letterplace_ideal.LetterplaceIdeal method), [55](#)
[group\(\)](#) (sage.algebras.group_algebra_new.GroupAlgebra method), [72](#)
[group\(\)](#) (sage.algebras.group_algebra_new.GroupAlgebraFunctor method), [74](#)
[GroupAlgebra](#) (class in sage.algebras.group_algebra_new), [70](#)

GroupAlgebraFunctor (class in sage.algebras.group_algebra_new), 74

H

HallAlgebra (class in sage.algebras.hall_algebra), 93

HallAlgebra.Element (class in sage.algebras.hall_algebra), 95

HallAlgebraMonomials (class in sage.algebras.hall_algebra), 97

HallAlgebraMonomials.Element (class in sage.algebras.hall_algebra), 98

hamilton_quatalg() (in module sage.algebras.free_algebra_quotient), 66

hash_involution_on_basis() (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.C method), 81

hash_involution_on_basis() (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.Cp method), 82

hash_involution_on_basis() (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T method), 85

homogeneous_component() (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 137

I

ideal() (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 3

ideal_monoid() (sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace method), 34

index_cmp() (in module sage.algebras.iwahori_hecke_algebra), 91

inverse() (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement method), 9

inverse() (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T.Element method), 84

inverse_generator() (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T method), 86

inverse_generators() (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T method), 86

inverse_image() (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism.FiniteDimensionalAlgebraMorphism method), 16

is_associative() (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 3

is_commutative() (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 3

is_commutative() (sage.algebras.free_algebra.FreeAlgebra_generic method), 22

is_commutative() (sage.algebras.group_algebra_new.GroupAlgebra method), 72

is_commutative() (sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace method), 34

is_commutative() (sage.algebras.shuffle_algebra.ShuffleAlgebra method), 106

is_commutative() (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 138

is_decomposable() (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), 125

is_division_algebra() (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 139

is_exact() (sage.algebras.group_algebra_new.GroupAlgebra method), 72

is_field() (sage.algebras.free_algebra.FreeAlgebra_generic method), 22

is_field() (sage.algebras.group_algebra_new.GroupAlgebra method), 73

is_field() (sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace method), 34

is_field() (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 139

is_finite() (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 3

is_finite() (sage.algebras.group_algebra_new.GroupAlgebra method), 73

is_finite() (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 139

is_FreeAlgebra() (in module sage.algebras.free_algebra), 28

is_FreeAlgebraQuotientElement() (in module sage.algebras.free_algebra_quotient_element), 67

is_generic() (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 139

is_homogeneous() (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), 126

is_integral_domain() (sage.algebras.group_algebra_new.GroupAlgebra method), 73

is_integral_domain() (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 140

[is_invertible\(\)](#) (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement method), [10](#)
[is_nilpotent\(\)](#) (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement method), [10](#)
[is_nilpotent\(\)](#) (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), [126](#)
[is_noetherian\(\)](#) (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), [140](#)
[is_unit\(\)](#) (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), [126](#)
[is_unitary\(\)](#) (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), [4](#)
[is_valid_profile\(\)](#) (in module sage.algebras.steenrod.steenrod_algebra_misc), [161](#)
[is_zero\(\)](#) (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), [4](#)
[is_zerodivisor\(\)](#) (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement method), [10](#)
[IwahoriHeckeAlgebra](#) (class in sage.algebras.iwahori_hecke_algebra), [75](#)
[IwahoriHeckeAlgebra.C](#) (class in sage.algebras.iwahori_hecke_algebra), [80](#)
[IwahoriHeckeAlgebra.Cp](#) (class in sage.algebras.iwahori_hecke_algebra), [81](#)
[IwahoriHeckeAlgebra.T](#) (class in sage.algebras.iwahori_hecke_algebra), [83](#)
[IwahoriHeckeAlgebra.T.Element](#) (class in sage.algebras.iwahori_hecke_algebra), [84](#)
[IwahoriHeckeAlgebra_nonstandard](#) (class in sage.algebras.iwahori_hecke_algebra), [89](#)
[IwahoriHeckeAlgebra_nonstandard.C](#) (class in sage.algebras.iwahori_hecke_algebra), [89](#)
[IwahoriHeckeAlgebra_nonstandard.Cp](#) (class in sage.algebras.iwahori_hecke_algebra), [90](#)
[IwahoriHeckeAlgebra_nonstandard.T](#) (class in sage.algebras.iwahori_hecke_algebra), [90](#)
[IwahoriHeckeAlgebraT\(\)](#) (in module sage.algebras.iwahori_hecke_algebra), [89](#)

L

[lc\(\)](#) (sage.algebras.letterplace.free_algebra_element_letterplace.FreeAlgebraElement_letterplace method), [44](#)
[left_matrix\(\)](#) (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement method), [10](#)
[left_table\(\)](#) (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), [4](#)
[letterplace_polynomial\(\)](#) (sage.algebras.letterplace.free_algebra_element_letterplace.FreeAlgebraElement_letterplace method), [44](#)
[LetterplaceIdeal](#) (class in sage.algebras.letterplace.letterplace_ideal), [53](#)
[lie_polynomial\(\)](#) (sage.algebras.free_algebra.FreeAlgebra_generic method), [22](#)
[lm\(\)](#) (sage.algebras.letterplace.free_algebra_element_letterplace.FreeAlgebraElement_letterplace method), [44](#)
[lm_divides\(\)](#) (sage.algebras.letterplace.free_algebra_element_letterplace.FreeAlgebraElement_letterplace method), [44](#)
[lt\(\)](#) (sage.algebras.letterplace.free_algebra_element_letterplace.FreeAlgebraElement_letterplace method), [45](#)

M

[make_mono_admissible\(\)](#) (in module sage.algebras.steenrod.steenrod_algebra_mult), [173](#)
[matrix\(\)](#) (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement method), [10](#)
[matrix\(\)](#) (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism.FiniteDimensionalAlgebraMorphism method), [16](#)
[matrix_action\(\)](#) (sage.algebras.free_algebra_quotient.FreeAlgebraQuotient method), [65](#)
[maximal_ideal\(\)](#) (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), [4](#)
[maximal_ideals\(\)](#) (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), [5](#)

`may_weight()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), 127
`milnor()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 140
`milnor()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), 127
`milnor_basis()` (in module sage.algebras.steenrod.steenrod_algebra_bases), 150
`milnor_mono_to_string()` (in module sage.algebras.steenrod.steenrod_algebra_misc), 162
`milnor_multiplication()` (in module sage.algebras.steenrod.steenrod_algebra_mult), 174
`milnor_multiplication_odd()` (in module sage.algebras.steenrod.steenrod_algebra_mult), 174
`minimal_polynomial()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement method), 11
`module()` (sage.algebras.free_algebra_quotient.FreeAlgebraQuotient method), 65
`monoid()` (sage.algebras.free_algebra.FreeAlgebra_generic method), 23
`monoid()` (sage.algebras.free_algebra_quotient.FreeAlgebraQuotient method), 65
`monomial_basis()` (sage.algebras.free_algebra_quotient.FreeAlgebraQuotient method), 65
`monomial_basis()` (sage.algebras.hall_algebra.HallAlgebra method), 96
`multinomial()` (in module sage.algebras.steenrod.steenrod_algebra_mult), 175
`multinomial_odd()` (in module sage.algebras.steenrod.steenrod_algebra_mult), 176

N

`ngens()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 5
`ngens()` (sage.algebras.free_algebra.FreeAlgebra_generic method), 23
`ngens()` (sage.algebras.free_algebra_quotient.FreeAlgebraQuotient method), 65
`ngens()` (sage.algebras.group_algebra_new.GroupAlgebra method), 73
`ngens()` (sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace method), 35
`ngens()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 140
`normal_form()` (sage.algebras.letterplace.free_algebra_element_letterplace.FreeAlgebraElement_letterplace method), 45
`normalize_profile()` (in module sage.algebras.steenrod.steenrod_algebra_misc), 162
`normalized_laurent_polynomial()` (in module sage.algebras.iwahori_hecke_algebra), 91

O

`one()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 5
`one_basis()` (sage.algebras.free_algebra.FreeAlgebra_generic method), 23
`one_basis()` (sage.algebras.free_algebra.PBWBasisOffFreeAlgebra method), 27
`one_basis()` (sage.algebras.hall_algebra.HallAlgebra method), 97
`one_basis()` (sage.algebras.hall_algebra.HallAlgebraMonomials method), 100
`one_basis()` (sage.algebras.shuffle_algebra.DualPBWBasis method), 103
`one_basis()` (sage.algebras.shuffle_algebra.ShuffleAlgebra method), 106
`one_basis()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 141
`order()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 141

P

`P()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 128
`pbw_basis()` (sage.algebras.free_algebra.FreeAlgebra_generic method), 23
`pbw_element()` (sage.algebras.free_algebra.FreeAlgebra_generic method), 24
`PBWBasisOffFreeAlgebra` (class in sage.algebras.free_algebra), 25
`PBWBasisOffFreeAlgebra.Element` (class in sage.algebras.free_algebra), 26
`poincare_birkhoff_witt_basis()` (sage.algebras.free_algebra.FreeAlgebra_generic method), 24
`poly_reduce()` (in module sage.algebras.letterplace.free_algebra_element_letterplace), 46
`poly_reduce()` (in module sage.algebras.letterplace.free_algebra_letterplace), 35
`poly_reduce()` (in module sage.algebras.letterplace.letterplace_ideal), 56

`primary_decomposition()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 5
`prime()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 141
`prime()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element method), 128
`product()` (sage.algebras.free_algebra.PBWBasisOffFreeAlgebra method), 27
`product()` (sage.algebras.shuffle_algebra.DualPBWBasis method), 103
`product_by_generator()` (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T method), 86
`product_by_generator_on_basis()` (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T method), 87
`product_on_basis()` (sage.algebras.free_algebra.FreeAlgebra_generic method), 24
`product_on_basis()` (sage.algebras.hall_algebra.HallAlgebra method), 97
`product_on_basis()` (sage.algebras.hall_algebra.HallAlgebraMonomials method), 100
`product_on_basis()` (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T method), 87
`product_on_basis()` (sage.algebras.shuffle_algebra.ShuffleAlgebra method), 107
`product_on_basis()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 141
`profile()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 142
`pst()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 143
`pst_mono_to_string()` (in module sage.algebras.steenrod.steenrod_algebra_misc), 165

Q

`Q()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 129
`q1()` (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra method), 88
`q2()` (sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra method), 88
`Q_exp()` (sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic method), 129
`quo()` (sage.algebras.free_algebra.FreeAlgebra_generic method), 24
`quotient()` (sage.algebras.free_algebra.FreeAlgebra_generic method), 24
`quotient_map()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 6

R

`random_element()` (sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra method), 6
`random_element()` (sage.algebras.group_algebra_new.GroupAlgebra method), 74
`rank()` (sage.algebras.free_algebra_quotient.FreeAlgebraQuotient method), 65
`reduce()` (sage.algebras.letterplace.free_algebra_element_letterplace.FreeAlgebraElement_letterplace method), 45
`reduce()` (sage.algebras.letterplace.letterplace_ideal.LetterplaceIdeal method), 56
`restricted_partitions()` (in module sage.algebras.steenrod.steenrod_algebra_bases), 151

S

`sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra` (module), 1
`sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element` (module), 9
`sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_ideal` (module), 13
`sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism` (module), 15
`sage.algebras.free_algebra` (module), 17
`sage.algebras.free_algebra_element` (module), 29
`sage.algebras.free_algebra_quotient` (module), 63
`sage.algebras.free_algebra_quotient_element` (module), 67
`sage.algebras.group_algebra_new` (module), 69
`sage.algebras.hall_algebra` (module), 93
`sage.algebras.iwahori_hecke_algebra` (module), 75
`sage.algebras.letterplace.free_algebra_element_letterplace` (module), 43

`sage.algebras.letterplace.free_algebra_letterplace` (module), 31
`sage.algebras.letterplace.letterplace_ideal` (module), 53
`sage.algebras.shuffle_algebra` (module), 101
`sage.algebras.steenrod.steenrod_algebra` (module), 109
`sage.algebras.steenrod.steenrod_algebra_bases` (module), 145
`sage.algebras.steenrod.steenrod_algebra_misc` (module), 157
`sage.algebras.steenrod.steenrod_algebra_mult` (module), 169
`scalar()` (`sage.algebras.hall_algebra.HallAlgebra.Element` method), 95
`scalar()` (`sage.algebras.hall_algebra.HallAlgebraMonomials.Element` method), 98
`serre_cartan_basis()` (in module `sage.algebras.steenrod.steenrod_algebra_bases`), 152
`serre_cartan_mono_to_string()` (in module `sage.algebras.steenrod.steenrod_algebra_misc`), 165
`set_degbound()` (`sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace` method), 35
`shuffle_algebra()` (`sage.algebras.shuffle_algebra.DualPBWBasis` method), 103
`ShuffleAlgebra` (class in `sage.algebras.shuffle_algebra`), 104
`singular_system()` (in module `sage.algebras.letterplace.free_algebra_element_letterplace`), 49
`singular_system()` (in module `sage.algebras.letterplace.free_algebra_letterplace`), 38
`singular_system()` (in module `sage.algebras.letterplace.letterplace_ideal`), 58
`Sq()` (in module `sage.algebras.steenrod.steenrod_algebra`), 115
`Sq()` (`sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_mod_two` method), 144
`steenrod_algebra_basis()` (in module `sage.algebras.steenrod.steenrod_algebra_bases`), 152
`steenrod_basis_error_check()` (in module `sage.algebras.steenrod.steenrod_algebra_bases`), 154
`SteenrodAlgebra()` (in module `sage.algebras.steenrod.steenrod_algebra`), 116
`SteenrodAlgebra_generic` (class in `sage.algebras.steenrod.steenrod_algebra`), 121
`SteenrodAlgebra_generic.Element` (class in `sage.algebras.steenrod.steenrod_algebra`), 121
`SteenrodAlgebra_mod_two` (class in `sage.algebras.steenrod.steenrod_algebra`), 144

T

`table()` (`sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra.FiniteDimensionalAlgebra` method), 7
`term_order_of_block()` (`sage.algebras.letterplace.free_algebra_letterplace.FreeAlgebra_letterplace` method), 35
`to_C_basis()` (`sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T` method), 87
`to_C_basis()` (`sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_nonstandard.T` method), 90
`to_Cp_basis()` (`sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra.T` method), 88
`to_Cp_basis()` (`sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_nonstandard.T` method), 91
`to_dual_pbw_element()` (`sage.algebras.shuffle_algebra.ShuffleAlgebra` method), 107
`to_pbw_basis()` (`sage.algebras.free_algebra_element.FreeAlgebraElement` method), 29
`to_T_basis()` (`sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_nonstandard.C` method), 89
`to_T_basis()` (`sage.algebras.iwahori_hecke_algebra.IwahoriHeckeAlgebra_nonstandard.Cp` method), 90
`top_class()` (`sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic` method), 143
`transpose_cmp()` (in module `sage.algebras.hall_algebra`), 100

V

`variable_names()` (`sage.algebras.shuffle_algebra.ShuffleAlgebra` method), 107
`variables()` (`sage.algebras.free_algebra_element.FreeAlgebraElement` method), 29
`vector()` (`sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_element.FiniteDimensionalAlgebraElement` method), 11
`vector()` (`sage.algebras.free_algebra_quotient_element.FreeAlgebraQuotientElement` method), 67
`vector_space()` (`sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_ideal.FiniteDimensionalAlgebraIdeal` method), 13

W

`wall_height()` (`sage.algebras.steenrod.steenrod_algebra.SteenrodAlgebra_generic.Element` method), [128](#)

`wall_long_mono_to_string()` (in module `sage.algebras.steenrod.steenrod_algebra_misc`), [166](#)

`wall_mono_to_string()` (in module `sage.algebras.steenrod.steenrod_algebra_misc`), [166](#)

`wood_mono_to_string()` (in module `sage.algebras.steenrod.steenrod_algebra_misc`), [167](#)

X

`xi_degrees()` (in module `sage.algebras.steenrod.steenrod_algebra_bases`), [154](#)

Z

`zero()` (`sage.algebras.finite_dimensional_algebras.finite_dimensional_algebra_morphism.FiniteDimensionalAlgebraHomset` method), [15](#)