1.1 Constructors for polynomial rings

This module provides the function `PolynomialRing()`, which constructs rings of univariate and multivariate polynomials, and implements caching to prevent the same ring being created in memory multiple times (which is wasteful and breaks the general assumption in Sage that parents are unique).

There is also a function `BooleanPolynomialRing_constructor()`, used for constructing Boolean polynomial rings, which are not technically polynomial rings but rather quotients of them (see module `sage.rings.polynomial.pbori` for more details).

```
construct a boolean polynomial ring with the following parameters:
INPUT:
  • n – number of variables (an integer > 1)
  • names – names of ring variables, may be a string or list/tuple of strings
  • order – term order (default: lex)
EXAMPLES:
```

```
sage: R.<x, y, z> = BooleanPolynomialRing() # indirect doctest
sage: R
Boolean PolynomialRing in x, y, z
sage: p = x^y + x^z + y^z
sage: x*p
x*y*z + x*y + x*z
sage: R.term_order()
Lexicographic term order
sage: R = BooleanPolynomialRing(5,'x',order='deglex(3),deglex(2)')
sage: R
Boolean PolynomialRing in x0, x1, x2, x3, x4
sage: R.term_order()
Block term order with blocks:
(Degree lexicographic term order of length 3,
  Degree lexicographic term order of length 2)
continues on next page
```
sage: R = BooleanPolynomialRing(3, 'x', order='degneglex')
sage: R.term_order()
Degree negative lexicographic term order
sage: BooleanPolynomialRing(names=('x', 'y'))
Boolean PolynomialRing in x, y
sage: BooleanPolynomialRing(names='x,y')
Boolean PolynomialRing in x, y

sage.rings.polynomial.polynomial_ring_constructor.PolynomialRing(base_ring, *args, **kwds)

Return the globally unique univariate or multivariate polynomial ring with given properties and variable name or names.

There are many ways to specify the variables for the polynomial ring:

1. PolynomialRing(base_ring, name, ...)
2. PolynomialRing(base_ring, names, ...)
3. PolynomialRing(base_ring, n, names, ...)
4. PolynomialRing(base_ring, n, ..., var_array=var_array, ...)

The ... at the end of these commands stands for additional keywords, like sparse or order:

**INPUT:**

- base_ring – a ring
- n – an integer
- name – a string
- names – a list or tuple of names (strings), or a comma separated string
- var_array – a list or tuple of names, or a comma separated string
- sparse – bool: whether or not elements are sparse. The default is a dense representation (sparse=False) for univariate rings and a sparse representation (sparse=True) for multivariate rings.
- order – string or TermOrder object, e.g.,
  - 'degrevlex' (default) – degree reverse lexicographic
  - 'lex' – lexicographic
  - 'deglex' – degree lexicographic
  - TermOrder('deglex',3) + TermOrder('deglex',3) – block ordering
- implementation – string or None; selects an implementation in cases where Sage includes multiple choices (currently \( \mathbb{Z}[x] \) can be implemented with 'NTL' or 'FLINT'; default is 'FLINT'). For many base rings, the "singular" implementation is available. One can always specify implementation="generic" for a generic Sage implementation which does not use any specialized library.

**Note:** If the given implementation does not exist for rings with the given number of generators and the given sparsity, then an error results.

**OUTPUT:**
PolynomialRing(base_ring, name, sparse=False) returns a univariate polynomial ring; also, PolynomialRing(base_ring, names, sparse=False) yields a univariate polynomial ring, if names is a list or tuple providing exactly one name. All other input formats return a multivariate polynomial ring.

UNIQUENESS and IMMUTABILITY: In Sage there is exactly one single-variate polynomial ring over each base ring in each choice of variable, sparseness, and implementation. There is also exactly one multivariate polynomial ring over each base ring for each choice of names of variables and term order. The names of the generators can only be temporarily changed after the ring has been created. Do this using the localvars context:

EXAMPLES:

1. PolynomialRing(base_ring, name, ...)

```python
sage: PolynomialRing(QQ, 'w')
Univariate Polynomial Ring in w over Rational Field
sage: PolynomialRing(QQ, name='w')
Univariate Polynomial Ring in w over Rational Field
```

Use the diamond brackets notation to make the variable ready for use after you define the ring:

```python
sage: R.<w> = PolynomialRing(QQ)
sage: (1 + w)^3
w^3 + 3*w^2 + 3*w + 1
```

You must specify a name:

```python
sage: PolynomialRing(QQ)
Traceback (most recent call last):
  ... TypeError: you must specify the names of the variables
```

```python
sage: R.<abc> = PolynomialRing(QQ, sparse=True); R
Sparse Univariate Polynomial Ring in abc over Rational Field
sage: R.<w> = PolynomialRing(PolynomialRing(GF(7), 'k')); R
Univariate Polynomial Ring in w over Univariate Polynomial Ring in k over Finite Field of size 7
```

The square bracket notation:

```python
sage: R.<y> = QQ['y']; R
Univariate Polynomial Ring in y over Rational Field
sage: y^2 + y
y^2 + y
```

In fact, since the diamond brackets on the left determine the variable name, you can omit the variable from the square brackets:

```python
sage: R.<zz> = QQ[]; R
Univariate Polynomial Ring in zz over Rational Field
sage: (zz + 1)^2
zz^2 + 2*zz + 1
```

This is exactly the same ring as what PolynomialRing returns:
However, rings with different variables are different:

```
sage: QQ['x'] == QQ['y']
False
```

Sage has two implementations of univariate polynomials over the integers, one based on NTL and one based on FLINT. The default is FLINT. Note that FLINT uses a “more dense” representation for its polynomials than NTL, so in particular, creating a polynomial like $2^{1000000} \times x^{1000000}$ in FLINT may be unwise.

```
sage: ZxNTL = PolynomialRing(ZZ, 'x', implementation='NTL'); ZxNTL
Univariate Polynomial Ring in x over Integer Ring (using NTL)
sage: ZxFLINT = PolynomialRing(ZZ, 'x', implementation='FLINT'); ZxFLINT
Univariate Polynomial Ring in x over Integer Ring
sage: ZxFLINT is ZZ['x']
True
sage: ZxFLINT is PolynomialRing(ZZ, 'x')
True
sage: xNTL = ZxNTL.gen()
sage: xFLINT = ZxFLINT.gen()
sage: xNTL.parent()
Univariate Polynomial Ring in x over Integer Ring (using NTL)
sage: xFLINT.parent()
Univariate Polynomial Ring in x over Integer Ring
```

There is a coercion from the non-default to the default implementation, so the values can be mixed in a single expression:

```
sage: (xNTL + xFLINT^2)
x^2 + x
```

The result of such an expression will use the default, i.e., the FLINT implementation:

```
sage: (xNTL + xFLINT^2).parent()
Univariate Polynomial Ring in x over Integer Ring
```

The generic implementation uses neither NTL nor FLINT:

```
sage: Zx = PolynomialRing(ZZ, 'x', implementation='generic'); Zx
Univariate Polynomial Ring in x over Integer Ring
sage: Zx.element_class
<... 'sage.rings.polynomial.polynomial_element.Polynomial_generic_dense'>
```

2. PolynomialRing(base_ring, names, ...)

```
sage: R = PolynomialRing(QQ, ['a,b,c']); R
Multivariate Polynomial Ring in a, b, c over Rational Field
sage: S = PolynomialRing(QQ, ['a','b','c']); S
Multivariate Polynomial Ring in a, b, c over Rational Field
```

(continues on next page)
sage: T = PolynomialRing(QQ, ('a','b','c')); T
Multivariate Polynomial Ring in a, b, c over Rational Field

All three rings are identical:

sage: R is S
True
sage: S is T
True

There is a unique polynomial ring with each term order:

sage: R = PolynomialRing(QQ, ('x,y,z'), order='degrevlex'); R
Multivariate Polynomial Ring in x, y, z over Rational Field
sage: S = PolynomialRing(QQ, ('x,y,z'), order='invlex'); S
Multivariate Polynomial Ring in x, y, z over Rational Field
sage: S is PolynomialRing(QQ, ('x,y,z'), order='invlex')
True
sage: R == S
False

Note that a univariate polynomial ring is returned, if the list of names is of length one. If it is of length zero, a
multivariate polynomial ring with no variables is returned.

sage: PolynomialRing(QQ, ['x'])
Univariate Polynomial Ring in x over Rational Field
sage: PolynomialRing(QQ, [])
Multivariate Polynomial Ring in no variables over Rational Field

The Singular implementation always returns a multivariate ring, even for 1 variable:

sage: PolynomialRing(QQ, ('x',), implementation="singular")
Univariate Polynomial Ring in x over Rational Field
sage: PolynomialRing(QQ, [])
Multivariate Polynomial Ring in no variables over Rational Field

3. PolynomialRing(base_ring, n, names, ...) (where the arguments n and names may be reversed)

If you specify a single name as a string and a number of variables, then variables labeled with numbers are created.

sage: PolynomialRing(QQ, 'x', 10)
Multivariate Polynomial Ring in x0, x1, x2, x3, x4, x5, x6, x7, x8, x9 over Rational Field
sage: PolynomialRing(QQ, 2, 'alpha0')
Multivariate Polynomial Ring in alpha00, alpha01 over Rational Field
sage: PolynomialRing(GF(7), 'y', 5)
Multivariate Polynomial Ring in y0, y1, y2, y3, y4 over Finite Field of size 7
sage: PolynomialRing(QQ, 'y', 3, sparse=True)
Multivariate Polynomial Ring in y0, y1, y2 over Rational Field

1.1. Constructors for polynomial rings
Note that a multivariate polynomial ring is returned when an explicit number is given.

```
sage: PolynomialRing(QQ,"x",1)
Multivariate Polynomial Ring in x over Rational Field
sage: PolynomialRing(QQ,"x",0)
Multivariate Polynomial Ring in no variables over Rational Field
```

It is easy in Python to create fairly arbitrary variable names. For example, here is a ring with generators labeled by the primes less than 100:

```
sage: R = PolynomialRing(ZZ, ['x%s' % p for p in primes(100)]); R
Multivariate Polynomial Ring in x2, x3, x5, x7, x11, x13, x17, x19, x23, x29, x31, ...
    x37, x41, x43, x47, x53, x59, x61, x67, x71, x73, x79, x83, x89, x97 over Integer Ring

sage: R.inject_variables()
Defining x2, x3, x5, x7, x11, x13, x17, x19, x23, x29, x31, ...
    x59, x61, x67, x71, x73, x79, x83, x89, x97

sage: (x2 + x41 + x71)^2
x2^2 + 2*x2*x41 + x41^2 + 2*x2*x71 + 2*x41*x71 + x71^2
```

4. PolynomialRing(base_ring, n, ..., var_array=var_array, ...)

This creates an array of variables where each variables begins with an entry in var_array and is indexed from 0 to n - 1.

```
sage: PolynomialRing(ZZ, 3, var_array=['x', 'y'])
Multivariate Polynomial Ring in x0, y0, x1, y1, x2, y2 over Integer Ring

sage: PolynomialRing(ZZ, 3, var_array='a,b')
Multivariate Polynomial Ring in a0, b0, a1, b1, a2, b2 over Integer Ring
```

It is possible to create higher-dimensional arrays:

```
sage: PolynomialRing(ZZ, 2, 3, var_array=('p', 'q'))
Multivariate Polynomial Ring in p00, q00, p01, q01, p02, q02, p10, q10, p11, q11, ...
    p12, q12 over Integer Ring

sage: PolynomialRing(ZZ, 2, 3, 4, var_array='m')
Multivariate Polynomial Ring in m000, m001, m002, m003, m010, m011, m012, m013, ...
    m020, m021, m022, m023, m100, m101, m102, m103, m110, m111, m112, m113, m120, ...
    m121, m122, m123 over Integer Ring
```

The array is always at least 2-dimensional. So, if var_array is a single string and only a single number n is given, this creates an \( n \times n \) array of variables:

```
sage: PolynomialRing(ZZ, 2, var_array='m')
Multivariate Polynomial Ring in m00, m01, m10, m11 over Integer Ring
```

Square brackets notation

You can alternatively create a polynomial ring over a ring \( R \) with square brackets:

```
sage: RR["x"]
Univariate Polynomial Ring in x over Real Field with 53 bits of precision
sage: RR["x,y"]
```

(continues on next page)
Multivariate Polynomial Ring in $x$, $y$ over Real Field with 53 bits of precision

```
sage: P.<x,y> = RR[]; P
Multivariate Polynomial Ring in $x$, $y$ over Real Field with 53 bits of precision
```

This notation does not allow to set any of the optional arguments.

### Changing variable names

Consider

```
sage: R.<x,y> = PolynomialRing(QQ,2); R
Multivariate Polynomial Ring in $x$, $y$ over Rational Field
sage: f = x^2 - 2*y^2
```

You can't just globally change the names of those variables. This is because objects all over Sage could have pointers to that polynomial ring.

```
sage: R._assign_names(['z','w'])
Traceback (most recent call last):
  ...
ValueError: variable names cannot be changed after object creation.
```

However, you can very easily change the names within a `with` block:

```
sage: with localvars(R, ['z','w']):
    print(f)
z^2 - 2*w^2
```

After the `with` block the names revert to what they were before:

```
sage: print(f)
x^2 - 2*y^2
```

### Choose an appropriate category for a polynomial ring

It is assumed that the corresponding base ring is nonzero.

**INPUT:**

- `base_ring_category` -- The category of ring over which the polynomial ring shall be defined
- `n_variables` -- number of variables

**EXAMPLES:**

```
sage: from sage.rings.polynomial.polynomial_ring_constructor import polynomial_default_category
sage: polynomial_default_category(Rings(),1) is Algebras(Rings()).Infinite()
True
sage: polynomial_default_category(Rings().Commutative(),1) is Algebras(Rings().Commutative()).Commutative().Infinite()
True
sage: polynomial_default_category(Fields(),1) is EuclideanDomains() &␣
    Algebras(Fields()).Infinite()
```

(continues on next page)
True
\[
\text{sage: polynomial_default_category(Fields(), 2) is UniqueFactorizationDomains() } & \\
\quad \text{CommutativeAlgebras(Fields()).Infinite()}
\]
True
\[
\text{sage: QQ['t'].category() is EuclideanDomains() } & \text{CommutativeAlgebras(QQ.category()).Infinite()}
\]
True
\[
\text{sage: QQ['s','t'].category() is UniqueFactorizationDomains() } & \\
\quad \text{CommutativeAlgebras(QQ.category()).Infinite()}
\]
True
\[
\text{sage: QQ['s']['t'].category() is UniqueFactorizationDomains() } & \\
\quad \text{CommutativeAlgebras(QQ['s'].category()).Infinite()}
\]
True

sage.rings.polynomial.polynomial_ring_constructor.unpickle_PolynomialRing(base_ring, arg1=None, arg2=None, sparse=False)

Custom unpickling function for polynomial rings.

This has the same positional arguments as the old PolynomialRing constructor before trac ticket #23338.
2.1 Univariate Polynomials and Polynomial Rings

Sage’s architecture for polynomials ‘under the hood’ is complex, interfacing to a variety of C/C++ libraries for polynomials over specific rings. In practice, the user rarely has to worry about which backend is being used.

The hierarchy of class inheritance is somewhat confusing, since most of the polynomial element classes are implemented as Cython extension types rather than pure Python classes and thus can only inherit from a single base class, whereas others have multiple bases.

2.1.1 Univariate Polynomial Rings

Sage implements sparse and dense polynomials over commutative and non-commutative rings. In the non-commutative case, the polynomial variable commutes with the elements of the base ring.

AUTHOR:

- William Stein
- Kiran Kedlaya (2006-02-13): added macaulay2 option
- Martin Albrecht (2006-08-25): removed it again as it isn’t needed anymore
- Simon King (2011-05): Dense and sparse polynomial rings must not be equal.

EXAMPLES:

```
sage: z = QQ['z'].0
sage: (z^3 + z - 1)^3
z^9 + 3*z^7 - 3*z^6 + 3*z^5 - 6*z^4 + 4*z^3 - 3*z^2 + 3*z - 1
```

Saving and loading of polynomial rings works:

```
sage: loads(dumps(QQ['x'])) == QQ['x']
True
sage: k = PolynomialRing(QQ['x','y']); loads(dumps(k)) == k
True
sage: k = PolynomialRing(ZZ,'y'); loads(dumps(k)) == k
True
sage: k = PolynomialRing(ZZ,'y', sparse=True); loads(dumps(k))
Sparse Univariate Polynomial Ring in y over Integer Ring
```
Rings with different variable names are not equal; in fact, by trac ticket #9944, polynomial rings are equal if and only if they are identical (which should be the case for all parent structures in Sage):

```
sage: QQ['y'] != QQ['x']
True
sage: QQ['y'] != QQ['z']
True
```

We create a polynomial ring over a quaternion algebra:

```
sage: A.<i,j,k> = QuaternionAlgebra(QQ, -1,-1)
sage: R.<w> = PolynomialRing(A,sparse=True)
sage: f = w^3 + (i+j)*w + 1
sage: f
w^3 + (i + j)*w + 1
sage: f^2
w^6 + (2*i + 2*j)*w^4 + 2*w^3 - 2*w^2 + (2*i + 2*j)*w + 1
sage: f = w + i ; g = w + j
sage: f * g
w^2 + (i + j)*w + k
sage: g * f
w^2 + (i + j)*w - k
```

trac ticket #9944 introduced some changes related with coercion. Previously, a dense and a sparse polynomial ring with the same variable name over the same base ring evaluated equal, but of course they were not identical. Coercion maps are cached - but if a coercion to a dense ring is requested and a coercion to a sparse ring is returned instead (since the cache keys are equal!), all hell breaks loose.

Therefore, the coercion between rings of sparse and dense polynomials works as follows:

```
sage: R.<x> = PolynomialRing(QQ, sparse=True)
sage: S.<x> = QQ[]
sage: S == R
False
sage: S.has_coerce_map_from(R)
True
sage: R.has_coerce_map_from(S)
False
sage: (R.0+S.0).parent()
Univariate Polynomial Ring in x over Rational Field
sage: (S.0+R.0).parent()
Univariate Polynomial Ring in x over Rational Field
```

It may be that one has rings of dense or sparse polynomials over different base rings. In that situation, coercion works by means of the `pushout()` formalism:

```
sage: R.<x> = PolynomialRing(GF(5), sparse=True)
sage: S.<x> = PolynomialRing(ZZ)
sage: R.has_coerce_map_from(S)
False
sage: S.has_coerce_map_from(R)
False
sage: (R.0+S.0).parent()
Univariate Polynomial Ring in x over Rational Field
sage: (S.0+R.0).parent()
Univariate Polynomial Ring in x over Rational Field
```
Univariate Polynomial Ring in x over Finite Field of size 5
```python
sage: (S.0 + R.0).parent().is_sparse()
False
```

Similarly, there is a coercion from the (non-default) NTL implementation for univariate polynomials over the integers to the default FLINT implementation, but not vice versa:

```python
sage: R.<x> = PolynomialRing(ZZ, implementation = 'NTL')
sage: S.<x> = PolynomialRing(ZZ, implementation = 'FLINT')
sage: (S.0+R.0).parent() is S
True
sage: (R.0+S.0).parent() is S
True
```

```python
class sage.rings.polynomial.polynomial_ring.PolynomialRing_cdvf(base_ring, name=None, sparse=False, element_class=None, category=None):
    Bases: sage.rings.polynomial.polynomial_ring.PolynomialRing_cdvr, sage.rings.polynomial.polynomial_ring.PolynomialRing_field

A class for polynomial ring over complete discrete valuation fields

class sage.rings.polynomial.polynomial_ring.PolynomialRing_cdvr(base_ring, name=None, sparse=False, element_class=None, category=None):
    Bases: sage.rings.polynomial.polynomial_ring.PolynomialRing_integral_domain

A class for polynomial ring over complete discrete valuation rings

class sage.rings.polynomial.polynomial_ring.PolynomialRing_commutative(base_ring, name=None, sparse=False, element_class=None, category=None):
    Bases: sage.rings.polynomial.polynomial_ring.PolynomialRing_general, sage.rings.ring.CommutativeAlgebra

Univariate polynomial ring over a commutative ring.

```python
quotient_by_principal_ideal(f, names=None, **kwds)
```

Return the quotient of this polynomial ring by the principal ideal (generated by) f.

**INPUT:**
- f - either a polynomial in self, or a principal ideal of self.
- further named arguments that are passed to the quotient constructor.

**EXAMPLES:**

```python
sage: R.<x> = QQ[]
sage: I = (x^2-1)*R
sage: R.quotient_by_principal_ideal(I)
Univariate Quotient Polynomial Ring in xbar over Rational Field with modulus x^2 - 1
```

The same example, using the polynomial instead of the ideal, and customizing the variable name:
```python
sage: R.<x> = QQ[]
sage: R.quotient_by_principal_ideal(x^2-1, names=('foo',))
Univariate Quotient Polynomial Ring in foo over Rational Field with modulus x^2 → 1
```

**weyl_algebra()**

Return the Weyl algebra generated from self.

**EXAMPLES:**

```python
sage: R = QQ['x']
sage: W = R.weyl_algebra(); W
Differential Weyl algebra of polynomials in x over Rational Field
sage: W.polynomial_ring() == R
True
```

**class** `sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_finite_field`

Univariate polynomial ring over a finite field.

**EXAMPLES:**

```python
sage: R = PolynomialRing(GF(27, 'a'), 'x')
sage: type(R)
<class 'sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_finite_field_with_category'>
```

**irreducible_element**(n, algorithm=None)

Construct a monic irreducible polynomial of degree n.

**INPUT:**

- n – integer: degree of the polynomial to construct
- algorithm – string: algorithm to use, or None
  - 'random': try random polynomials until an irreducible one is found.
  - 'first_lexicographic': try polynomials in lexicographic order until an irreducible one is found.

**OUTPUT:**

A monic irreducible polynomial of degree n in self.

**EXAMPLES:**

```python
sage: GF(5^3, 'a')['x'].irreducible_element(2)
x^2 + 3*a^2 + a + 2
sage: GF(19)['x'].irreducible_element(21, algorithm='first_lexicographic')
x^21 + x + 5
sage: GF(5**2, 'a')['x'].irreducible_element(17, algorithm='first_lexicographic')
x^17 + a^8*x + 4*a + 3
```
AUTHORS:
  • Peter Bruin (June 2013)
  • Jean-Pierre Flori (May 2014)

class sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_mod_n(base_ring, name=None, element_class=None, implementation=None, category=None):

Bases: sage.rings.polynomial.polynomial_ring.PolynomialRing_commutative

modulus()

EXAMPLES:

```
sage: R.<x> = Zmod(15)[]
sage: R.modulus()
15
```

residue_field(ideal, names=None)

Return the residue finite field at the given ideal.

EXAMPLES:

```
sage: R.<t> = GF(2)[]
sage: k.<a> = R.residue_field(t^3 + t + 1); k
Residue field in a of Principal ideal (t^3 + t + 1) of Univariate Polynomial Ring in t over Finite Field of size 2 (using GF2X)
sage: k.list()
[0, a, a^2, a + 1, a^2 + a, a^2 + a + 1, a^2 + 1, 1]
sage: R.residue_field(t)
Residue field of Principal ideal (t) of Univariate Polynomial Ring in t over Finite Field of size 2 (using GF2X)
sage: P = R.irreducible_element(8) * R
sage: P
Principal ideal (t^8 + t^4 + t^3 + t^2 + 1) of Univariate Polynomial Ring in t over Finite Field of size 2 (using GF2X)
sage: k.<a> = R.residue_field(P); k
Residue field in a of Principal ideal (t^8 + t^4 + t^3 + t^2 + 1) of Univariate Polynomial Ring in t over Finite Field of size 2 (using GF2X)
sage: k.cardinality()
256
```

Non-maximal ideals are not accepted:

```
sage: R.residue_field(t^2 + 1)
Traceback (most recent call last):
  ... ArithmeticError: ideal is not maximal
sage: R.residue_field(0)
Traceback (most recent call last):
  ... ArithmeticError: ideal is not maximal
sage: R.residue_field(1)
Traceback (most recent call last):
  ... ArithmeticError: ideal is not maximal
```
class sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_mod_p(base_ring, name='x',
implementation=None, category=None)

Bases: sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_finite_field, sage.
rings.polynomial.polynomial_ring.PolynomialRing_dense_mod_n, sage.rings.polynomial.
polynomial_singular_interface.PolynomialRing_singular_repr

irreducible_element(n, algorithm=None)
Construct a monic irreducible polynomial of degree n.

INPUT:

• n – integer: the degree of the polynomial to construct
• algorithm – string: algorithm to use, or None. Currently available options are:
  – 'adleman-lenstra': a variant of the Adleman–Lenstra algorithm as implemented in PARI.
  – 'conway': look up the Conway polynomial of degree n over the field of p elements in the database;
    raise a RuntimeError if it is not found.
  – 'ffprimroot': use the ffprimroot() function from PARI.
  – 'first_lexicographic': return the lexicographically smallest irreducible polynomial of de-
    gree n.
  – 'minimal_weight': return an irreducible polynomial of degree n with minimal number of non-
    zero coefficients. Only implemented for p = 2.
  – 'primitive': return a polynomial f such that a root of f generates the multiplicative group of
    the finite field extension defined by f. This uses the Conway polynomial if possible, otherwise it
    uses ffprimroot.
  – 'random': try random polynomials until an irreducible one is found.

If algorithm is None, use x − 1 in degree 1. In degree > 1, the Conway polynomial is used if it is
found in the database. Otherwise, the algorithm minimal_weight is used if p = 2, and the algorithm
adleman-lenstra if p > 2.

OUTPUT:
A monic irreducible polynomial of degree n in self.

EXAMPLES:

```
sage: GF(5)勋[x'].irreducible_element(2)
x^2 + 4*x + 2
sage: GF(5)勋[x'].irreducible_element(2, algorithm="adleman-lenstra")
x^2 + x + 1
sage: GF(5)勋[x'].irreducible_element(2, algorithm="primitive")
x^2 + 4*x + 2
sage: GF(5)勋[x'].irreducible_element(32, algorithm="first_lexicographic")
x^32 + 2
sage: GF(5)勋[x'].irreducible_element(32, algorithm="conway")
Traceback (most recent call last):
  ...
RuntimeError: requested Conway polynomial not in database.
sage: GF(5)勋[x'].irreducible_element(32, algorithm="primitive")
x^32 + ...
```

In characteristic 2:
In degree 1:

\[
sage: \text{GF}(97)[\text{'}x\text{'].irreducible_element(1)}
\]
\[
x + 96
\]
\[
sage: \text{GF}(97)[\text{'}x\text{'].irreducible_element(1, algorithm="conway")}
\]
\[
x + 92
\]
\[
sage: \text{GF}(97)[\text{'}x\text{'].irreducible_element(1, algorithm="adleman-lenstra")}
\]
\[
x
\]

AUTHORS:

- Peter Bruin (June 2013)
- Jeroen Demeyer (September 2014): add “ffprimroot” algorithm, see trac ticket #8373.

class sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_padic_field_capped_relative(base_ring, name=None, element_class=None, category=None)

Bases: sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_padic_field_generic

class sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_padic_field_generic(base_ring, name=None, element_class=None, category=None)

Bases: sage.rings.polynomial.polynomial_ring.PolynomialRing_cdvf

A class for dense polynomial ring over padic fields

class sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_padic_ring_capped_absolute(base_ring, name=None, element_class=None, category=None)

Bases: sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_padic_ring_generic

2.1. Univariate Polynomials and Polynomial Rings
class sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_padic_ring_capped_relative

Bases: sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_padic_ring_generic

class sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_padic_ring_fixed_mod

Bases: sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_padic_ring_generic

class sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_padic_ring_generic

Bases: sage.rings.polynomial.polynomial_ring.PolynomialRing_cdvr

A class for dense polynomial ring over padic rings

class sage.rings.polynomial.polynomial_ring.PolynomialRing_field

Bases: sage.rings.polynomial.polynomial_ring.PolynomialRing_integral_domain, sage.rings.ring.PrincipalIdealDomain

divided_difference(points, full_table=False)

Return the Newton divided-difference coefficients of the Lagrange interpolation polynomial through points.

INPUT:

- points – a list of pairs \((x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\) of elements of the base ring of self, where \(x_i - x_j\) is invertible for \(i \neq j\). This method converts the \(x_i\) and \(y_i\) into the base ring of self.

- full_table – boolean (default: False): If True, return the full divided-difference table. If False, only return entries along the main diagonal; these are the Newton divided-difference coefficients \(F_{i,i}\).

OUTPUT:

The Newton divided-difference coefficients of the \(n\)-th Lagrange interpolation polynomial \(P_n(x)\) that passes through the points in points (see lagrange_polynomial()). These are the coefficients \(F_{0,0}, F_{1,1}, \ldots\) in the base ring of self such that

\[
P_n(x) = \sum_{i=0}^{n} F_{i,i} \prod_{j=0}^{i-1} (x - x_j)
\]

EXAMPLES:
Only return the divided-difference coefficients $F_{i,j}$. This example is taken from Example 1, page 121 of [BF2005]:

```sage
sage: points = [(1.0, 0.7651977), (1.3, 0.6200860), (1.6, 0.4554022), (1.9, 0.2818186), (2.2, 0.1103623)]
sage: R = PolynomialRing(RR, "x")
sage: R.divided_difference(points)
[0.765197700000000,
-0.483705666666666,
-0.108733888888889,
0.0658783950617283,
0.00182510288066044]
```

Now return the full divided-difference table:

```sage
sage: points = [(1.0, 0.7651977), (1.3, 0.6200860), (1.6, 0.4554022), (1.9, 0.2818186), (2.2, 0.1103623)]
sage: R = PolynomialRing(RR, "x")
sage: R.divided_difference(points, full_table=True)
[[0.765197700000000],
[0.620086000000000, -0.483705666666666],
[0.455402200000000, -0.548946000000000, -0.108733888888889],
[0.281818600000000, -0.578612000000000,
-0.049443333333339,
0.0658783950617283],
[0.110362300000000, -0.571520999999999,
0.011818333333349,
0.0680685185185209,
0.00182510288066044]]
```

The following example is taken from Example 4.12, page 225 of [MF1999]:

```sage
sage: points = [(1, -3), (2, 0), (3, 15), (4, 48), (5, 105), (6, 192)]
sage: R = PolynomialRing(QQ, "x")
sage: R.divided_difference(points)
[-3, 3, 6, 1, 0, 0, 0]
sage: R.divided_difference(points, full_table=True)
[[-3],
[3, 6, 1, 0, 0],
[15, 15, 6],
[48, 33, 9, 1],
[105, 57, 12, 1, 0],
[192, 87, 15, 1, 0, 0]]
```

```
fraction_field()
Returns the fraction field of self.

EXAMPLES:

sage: R.<t> = GF(5)[]
sage: R.fraction_field()
Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 5
```
lagrange_polynomial(points, algorithm='divided_difference', previous_row=None)

Return the Lagrange interpolation polynomial through the given points.

**INPUT:**

- **points** – a list of pairs \((x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\) of elements of the base ring of **self**, where \(x_i - x_j\) is invertible for \(i \neq j\). This method converts the \(x_i\) and \(y_i\) into the base ring of **self**.
- **algorithm** – (default: 'divided_difference'): one of the following:
  - 'divided_difference': use the method of divided differences.
  - algorithm='neville': adapt Neville's method as described on page 144 of [BF2005] to recursively generate the Lagrange interpolation polynomial. Neville's method generates a table of approximating polynomials, where the last row of that table contains the \(n\)-th Lagrange interpolation polynomial. The adaptation implemented by this method is to only generate the last row of this table, instead of the full table itself. Generating the full table can be memory inefficient.
- **previous_row** – (default: None): This option is only relevant if used with algorithm='neville'. If provided, this should be the last row of the table resulting from a previous use of Neville's method. If such a row is passed, then points should consist of both previous and new interpolating points. Neville's method will then use that last row and the interpolating points to generate a new row containing an interpolation polynomial for the new points.

**OUTPUT:**

The Lagrange interpolation polynomial through the points \((x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\). This is the unique polynomial \(P_n\) of degree at most \(n\) in **self** satisfying \(P_n(x_i) = y_i\) for \(0 \leq i \leq n\).

**EXAMPLES:**

By default, we use the method of divided differences:

```
sage: R = PolynomialRing(QQ, 'x')
sage: f = R.lagrange_polynomial([(0,1),(2,2),(3,-2),(-4,9)]); f
-23/84*x^3 - 11/84*x^2 + 13/7*x + 1
sage: f(0)
1
sage: f(2)
2
sage: f(3)
-2
sage: f(-4)
9
sage: R = PolynomialRing(GF(2**3,'a'), 'x')
sage: a = R.base_ring().gen()
sage: f = R.lagrange_polynomial([(a^2+a,a),(a,1),(a^2,a^2+a+1)]); f
a^2*x^2 + a^2*x + a^2
sage: f(a^2+a)
a
sage: f(a)
1
sage: f(a^2)
a^2 + a + 1
```

Now use a memory efficient version of Neville's method:

```
sage: R = PolynomialRing(QQ, 'x')
sage: R.lagrange_polynomial([(0,1),(2,2),(3,-2),(-4,9)], algorithm="neville")
```
[9, 
-11/7*x + 19/7, 
-17/42*x^2 - 83/42*x + 53/7, 
-23/84*x^3 - 11/84*x^2 + 13/7*x + 1]

sage: R = PolynomialRing(GF(2**3, 'a'), 'x')
sage: a = R.base_ring().gen()
sage: R.lagrange_polynomial([(a^2+a,a),(a,1),(a^2,a^2+a+1)], algorithm="neville")
[a^2 + a + 1, x + a + 1, a^2*x^2 + a^2*x + a^2]

Repeated use of Neville’s method to get better Lagrange interpolation polynomials:

sage: R = PolynomialRing(QQ, 'x')
sage: p = R.lagrange_polynomial([(0,1),(2,2)], algorithm="neville")
sage: R.lagrange_polynomial([(0,1),(2,2),(3,-2),(-4,9)], algorithm="neville", previous_row=p)[-1]
-23/84*x^3 - 11/84*x^2 + 13/7*x + 1

sage: R = PolynomialRing(GF(2**3, 'a'), 'x')
sage: a = R.base_ring().gen()
sage: p = R.lagrange_polynomial([(a^2+a,a),(a,1)], algorithm="neville")
sage: R.lagrange_polynomial([(a^2+a,a),(a,1),(a^2,a^2+a+1)], algorithm="neville", previous_row=p)[-1]
a^2*x^2 + a^2*x + a^2

```python
class sage.rings.polynomial.polynomial_ring.PolynomialRing_general(base_ring, name=None, sparse=False, element_class=None, category=None):
    Bases: sage.rings.ring.Algebra

    Univariate polynomial ring over a ring.

    base_extend(R)
    Return the base extension of this polynomial ring to R.

    EXAMPLES:

    sage: R.<x> = RR []; R
    Univariate Polynomial Ring in x over Real Field with 53 bits of precision
    sage: R.base_extend(CC)
    Univariate Polynomial Ring in x over Complex Field with 53 bits of precision
    sage: R.base_extend(QQ)
    Traceback (most recent call last):
    ...  
    TypeError: no such base extension
    sage: R.change_ring(QQ)
    Univariate Polynomial Ring in x over Rational Field

    change_ring(R)
    Return the polynomial ring in the same variable as self over R.

    EXAMPLES:

    sage: R.<ZZZ> = RealIntervalField() []; R
    Univariate Polynomial Ring in ZZZ over Real Interval Field with 53 bits of
    --precision
```

2.1. Univariate Polynomials and Polynomial Rings 19
sage: R.change_ring(GF(19^2, 'b'))
Univariate Polynomial Ring in ZZZ over Finite Field in b of size 19^2

**change_var**(var)

Return the polynomial ring in variable var over the same base ring.

**EXAMPLES:**

```python
sage: R.<x> = ZZ[]; R
Univariate Polynomial Ring in x over Integer Ring
sage: R.change_var('y')
Univariate Polynomial Ring in y over Integer Ring
```

**characteristic**()

Return the characteristic of this polynomial ring, which is the same as that of its base ring.

**EXAMPLES:**

```python
sage: R.<ZZZ> = RealIntervalField(); R
Univariate Polynomial Ring in ZZZ over Real Interval Field with 53 bits of precision
sage: R.characteristic()
0
sage: S = R.change_ring(GF(19^2, 'b')); S
Univariate Polynomial Ring in ZZZ over Finite Field in b of size 19^2
sage: S.characteristic()
19
```

**completion**(p, prec=20, extras=None)

Return the completion of self with respect to the irreducible polynomial p. Currently only implemented for p=self.gen(), i.e. you can only complete R[x] with respect to x, the result being a ring of power series in x. The prec variable controls the precision used in the power series ring.

**EXAMPLES:**

```python
sage: P.<x> = PolynomialRing(QQ)
sage: P
Univariate Polynomial Ring in x over Rational Field
sage: PP = P.completion(x)
sage: PP
Power Series Ring in x over Rational Field
sage: f = 1 - x
sage: PP(f)
1 - x
sage: 1/f
-1/(x - 1)
```

**construction()**

**cyclotomic_polynomial**(n)

Return the nth cyclotomic polynomial as a polynomial in this polynomial ring. For details of the implementation, see the documentation for `sage.rings.polynomial.cyclotomic.cyclotomic_coeffs()`.
EXAMPLES:

```
sage: R = ZZ['x']
sage: R.cyclotomic_polynomial(8)
x^4 + 1
sage: R.cyclotomic_polynomial(12)
x^4 - x^2 + 1
sage: S = PolynomialRing(FiniteField(7), 'x')
sage: S.cyclotomic_polynomial(12)
x^4 + 6*x^2 + 1
sage: S.cyclotomic_polynomial(1)
x + 6
```

```
extend_variables(added_names, order='degrevlex')
Returns a multivariate polynomial ring with the same base ring but with added_names as additional variables.
```

```
sage: R.<x> = ZZ[]; R
Univariate Polynomial Ring in x over Integer Ring
sage: R.extend_variables(('y', 'z'))
Multivariate Polynomial Ring in x, y, z over Integer Ring
```

```
flattening_morphism()
Return the flattening morphism of this polynomial ring
```

```
sage: QQ['a','b']['x'].flattening_morphism()
Flattening morphism:
  From: Univariate Polynomial Ring in x over Multivariate Polynomial Ring in a,b over Rational Field
  To:  Multivariate Polynomial Ring in a, b, x over Rational Field
```

```
gen(n=0)
Return the indeterminate generator of this polynomial ring.
```

```
sage: R.<abc> = Integers(8)[]; R
Univariate Polynomial Ring in abc over Ring of integers modulo 8
sage: t = R.gen(); t
abc
sage: t.is_gen()
True
```

An identical generator is always returned.

```
sage: t is R.gen()
True
```
gens_dict()
Return a dictionary whose entries are {name:variable,...}, where name stands for the variable names of this object (as strings) and variable stands for the corresponding generators (as elements of this object).

EXAMPLES:

```
sage: R.<y,x,a42> = RR[]
sage: R.gens_dict()
{''a42'': a42, 'x': x, 'y': y}
```

is_exact()
EXAMPLES:

```
sage: class Foo:
....:     def __init__(self, x):
....:         self._x = x
....:     @cached_method
....:     def f(self):
....:         return self._x^2
sage: a = Foo(2)
sage: print(a.f.cache)
None
sage: a.f()
4
sage: a.f.cache
4
```

is_field(proof=True)
Return False, since polynomial rings are never fields.

EXAMPLES:

```
sage: R.<z> = Integers(2)[]; R
Univariate Polynomial Ring in z over Ring of integers modulo 2 (using GF2X)
sage: R.is_field()
False
```

is_integral_domain(proof=True)
EXAMPLES:

```
sage: ZZ['x'].is_integral_domain()
True
sage: Integers(8)['x'].is_integral_domain()
False
```

is_noetherian()

is_sparse()
Return true if elements of this polynomial ring have a sparse representation.

EXAMPLES:

```
sage: R.<z> = Integers(8)[]; R
Univariate Polynomial Ring in z over Ring of integers modulo 8
sage: R.is_sparse()
False
```

(continues on next page)

(continued from previous page)

```python
sage: R.<W> = PolynomialRing(QQ, sparse=True); R
Sparse Univariate Polynomial Ring in W over Rational Field
sage: R.is_sparse()
True
```

**is_unique_factorization_domain**(proof=True)

EXAMPLES:

```python
sage: ZZ['x'].is_unique_factorization_domain()
True
sage: Integers(8)['x'].is_unique_factorization_domain()
False
```

**karatsuba_threshold**()

Return the Karatsuba threshold used for this ring by the method _mul_karatsuba to fall back to the schoolbook algorithm.

EXAMPLES:

```python
sage: K = QQ['x']
sage: K.karatsuba_threshold()
8
sage: K = QQ['x']['y']
sage: K.karatsuba_threshold()
0
```

**krull_dimension**()

Return the Krull dimension of this polynomial ring, which is one more than the Krull dimension of the base ring.

EXAMPLES:

```python
sage: R.<x> = QQ[]
sage: R.krull_dimension()
1
sage: R.<z> = GF(9,'a')[]; R
Univariate Polynomial Ring in z over Finite Field in a of size 3^2
sage: R.krull_dimension()
1
sage: S.<t> = R[]
sage: S.krull_dimension()
2
sage: for n in range(10):
    ....: S = PolynomialRing(S,'w')
    sage: S.krull_dimension()
    12
```

**monics**(of_degree=None, max_degree=None)

Return an iterator over the monic polynomials of specified degree.

INPUT: Pass exactly one of:

- max_degree - an int; the iterator will generate all monic polynomials which have degree less than or equal to max_degree
- of_degree - an int; the iterator will generate all monic polynomials which have degree of_degree

2.1. Univariate Polynomials and Polynomial Rings 23
OUTPUT: an iterator

EXAMPLES:

```python
sage: P = PolynomialRing(GF(4, 'a'), 'y')
sage: for p in P.monomics(of_degree = 2): print(p)
y^2
y^2 + a
y^2 + a + 1
y^2 + 1
y^2 + a*y
y^2 + a*y + a
y^2 + a*y + a + 1
y^2 + a*y + 1
y^2 + (a + 1)*y
y^2 + (a + 1)*y + a
y^2 + (a + 1)*y + a + 1
y^2 + (a + 1)*y + 1
y^2 + y
y^2 + y + a
y^2 + y + a + 1
y^2 + y + 1
sage: for p in P.monomics(max_degree = 1): print(p)
1
y
y + a
y + a + 1
y + 1
sage: for p in P.monomics(max_degree = 1, of_degree = 3): print(p)
Traceback (most recent call last):
...
ValueError: you should pass exactly one of of_degree and max_degree
```

AUTHORS:
- Joel B. Mohler

`monomial(exponent)`

Return the monomial with the exponent.

**INPUT:**
- `exponent` – nonnegative integer

**EXAMPLES:**

```python
sage: R.<x> = PolynomialRing(ZZ)
sage: R.monomial(5)
x^5
sage: e=(10,)
sage: R.monomial(*e)
x^10
sage: m = R.monomial(100)
sage: R.monomial(m.degree()) == m
True
```
ngens()

Return the number of generators of this polynomial ring, which is 1 since it is a univariate polynomial ring.

EXAMPLES:

```python
sage: R.<z> = Integers(8)[]; R
Univariate Polynomial Ring in z over Ring of integers modulo 8
sage: R.ngens()
1
```

parameter()

Return the generator of this polynomial ring.

This is the same as self.gen().

polynomials(of_degree=None, max_degree=None)

Return an iterator over the polynomials of specified degree.

INPUT: Pass exactly one of:

- max_degree - an int; the iterator will generate all polynomials which have degree less than or equal to max_degree
- of_degree - an int; the iterator will generate all polynomials which have degree of_degree

OUTPUT: an iterator

EXAMPLES:

```python
sage: P = PolynomialRing(GF(3),'y')
sage: for p in P.polynomials( of_degree = 2 ): print(p)
y^2
y^2 + 1
y^2 + 2
y^2 + y
y^2 + y + 1
y^2 + y + 2
y^2 + 2*y
y^2 + 2*y + 1
y^2 + 2*y + 2
2*y^2
2*y^2 + 1
2*y^2 + 2
2*y^2 + y
2*y^2 + y + 1
2*y^2 + y + 2
2*y^2 + 2*y
2*y^2 + 2*y + 1
2*y^2 + 2*y + 2
sage: for p in P.polynomials( max_degree = 1 ): print(p)
0
1
2
y
y + 1
y + 2
2*y
2*y + 1
```

(continues on next page)
AUTHORS:

• Joel B. Mohler

**random_element**(degree=(_1, 2), *args, **kwds)

Return a random polynomial of given degree or with given degree bounds.

**INPUT:**

• degree - optional integer for fixing the degree or or a tuple of minimum and maximum degrees. By default set to (-1,2).

• *args, **kwds - Passed on to the random_element method for the base ring

**EXAMPLES:**

```python
sage: R.<x> = ZZ[]
sage: R.random_element(10, 5,10)
5*x^10 + 5*x^9 + 9*x^8 + 8*x^7 + 6*x^6 + 8*x^5 + 8*x^4 + 9*x^3 + 8*x^2 + 8*x + 8
sage: R.random_element(6)
x^6 - 2*x^5 - 2*x^3 + 2*x^2 - 3*x
sage: R.random_element(6)
-x^6 + x^5 + x^2 - x
sage: R.random_element(6)
-5*x^6 + x^5 + 14*x^4 - x^3 + x^2 - x + 4
```

If a tuple of two integers is given for the degree argument, a degree is first uniformly chosen, then a polynomial of that degree is given:

```python
sage: R.random_element(degree=(0,8))
4*x^4 + 2*x^3 - x + 4
sage: R.random_element(degree=(0,8))
x + 1
```

Note that the zero polynomial has degree -1, so if you want to consider it set the minimum degree to -1:

```python
sage: any(R.random_element(degree=(-1,2),x=-1,y=1) == R.zero() for _ in range(100))
True
```

**set_karatsuba_threshold**(Karatsuba_threshold)

Changes the default threshold for this ring in the method _mul_karatsuba to fall back to the schoolbook algorithm.

**Warning:** This method may have a negative performance impact in polynomial arithmetic. So use it at your own risk.
```python
sage: K = QQ['x']
sage: K.karatsuba_threshold()
8
sage: K.set_karatsuba_threshold(0)
sage: K.karatsuba_threshold()
0
```

**some_elements()**

Return a list of polynomials.

This is typically used for running generic tests.

**EXAMPLES:**

```python
sage: R.<x> = QQ[]
sage: R.some_elements()
[x, 0, 1, 1/2, x^2 + 2*x + 1, x^3, x^2 - 1, x^2 + 1, 2*x^2 + 2]
```

**variable_names_recursive(depth=+ Infinity)**

Return the list of variable names of this ring and its base rings, as if it were a single multi-variate polynomial.

**INPUT:**

- depth – an integer or Infinity.

**OUTPUT:**

A tuple of strings.

**EXAMPLES:**

```python
sage: R = QQ['x']['y']['z']
sage: R.variable_names_recursive()
('x', 'y', 'z')
sage: R.variable_names_recursive(2)
('y', 'z')
```

**class** `sage.rings.polynomial.polynomial_ring.PolynomialRing_integral_domain` *(base_ring, name='x', sparse=False, implementation=None, element_class=None, category=None)*

**Bases:** `sage.rings.polynomial.polynomial_ring.PolynomialRing_commutative`, `sage.rings.polynomial.polynomial_singular_interface.PolynomialRing_singular_repr`, `sage.rings.ring.IntegralDomain`

**well_polynomials**(d, q, sign=1, lead=1)

Return all integer polynomials whose complex roots all have a specified absolute value.

Such polynomials $f$ satisfy a functional equation

$$T^d f(q/T) = sq^{d/2} f(T)$$

where $d$ is the degree of $f$, $s$ is a sign and $q^{1/2}$ is the absolute value of the roots of $f$.

**INPUT:**
• \(d\) – integer, the degree of the polynomials
• \(q\) – integer, the square of the complex absolute value of the roots
• \(\text{sign}\) – integer (default 1), the sign \(s\) of the functional equation
• \(\text{lead}\) – integer, list of integers or list of pairs of integers (default 1), constraints on the leading few coefficients of the generated polynomials. If pairs \((a, b)\) of integers are given, they are treated as a constraint of the form \(a \equiv b \pmod{m}\); the moduli must be in decreasing order by divisibility, and the modulus of the leading coefficient must be 0.

See also:

More documentation and additional options are available using the iterator \sage{rings.polynomial.weil.weil_polynomials.WeilPolynomials} directly. In addition, polynomials have a method \sage{is_weil_polynomial} to test whether or not the given polynomial is a Weil polynomial.

EXAMPLES:

```python
sage: R.<T> = ZZ[]
sage: L = R.weil_polynomials(4, 2)
sage: len(L)
35
sage: L[9]
T^4 + T^3 + 2*T^2 + 2*T + 4
sage: all(p.is_weil_polynomial() for p in L)
True
```

Setting multiple leading coefficients:

```python
sage: R.<T> = QQ[]
sage: l = R.weil_polynomials(4, 2, lead=((1,0),(2,4),(1,2)))
sage: l
[T^4 + 2*T^3 + 5*T^2 + 4*T + 2, T^4 + 2*T^3 + 3*T^2 + 4*T + 2, T^4 - 2*T^3 + 5*T^2 - 4*T + 2, T^4 - 2*T^3 + 3*T^2 - 4*T + 2]
```

We do not require Weil polynomials to be monic. This example generates Weil polynomials associated to K3 surfaces over \(GF(2)\) of Picard number at least 12:

```python
sage: R.<T> = QQ[]
sage: l = R.weil_polynomials(10, 1, lead=2)
sage: len(l)
4865
sage: l[len(l)//2]
2*T^10 + T^8 + T^4 + T^2 + 2
```

\sage{rings.polynomial.polynomial_ring.is_PolynomialRing}(x)

Return True if \(x\) is a univariate polynomial ring (and not a sparse multivariate polynomial ring in one variable).

EXAMPLES:

```python
sage: from sage.rings.polynomial.polynomial_ring import is_PolynomialRing
sage: from sage.rings.polynomial.multi_polynomial_ring import is_MPolynomialRing
sage: is_PolynomialRing(2)
False
sage: is_PolynomialRing(ZZ[['x,y,z']])
False
```

This polynomial ring is not univariate.

```python
sage: is_PolynomialRing(ZZ['x,y,z'])
False
```

(continues on next page)
Univariate means not only in one variable, but is a specific data type. There is a multivariate (sparse) polynomial ring data type, which supports a single variable as a special case.

```
sage: R.<w> = PolynomialRing(ZZ, implementation="singular"); R
Multivariate Polynomial Ring in w over Integer Ring
sage: is_PolynomialRing(R)
False
sage: type(R)
<type 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomialRing_libsingular'>
```

```
sage: z = polygen(QQ, 'z')
sage: z^3 + z +1
z^3 + z + 1
sage: parent(z)
Univariate Polynomial Ring in z over Rational Field
```

Note: If you give a list or comma separated string to polygen, you’ll get a tuple of indeterminates, exactly as if you called polysgens.

```
sage: x,y,z = polysgens(QQ,['x','y','z'])
sage: (x+y+z)^2
x^2 + 2*x*y + y^2 + 2*x*z + 2*y*z + z^2
sage: parent(x)
Multivariate Polynomial Ring in x, y, z over Rational Field
sage: t = polysgens(QQ,['x','yz','abc'])
sage: t
(x, yz, abc)
```

The number of generators can be passed as a third argument:
2.1.2 Ring homomorphisms from a polynomial ring to another ring

This module currently implements the canonical ring homomorphism from $A[x]$ to $B[x]$ induced by a ring homomorphism from $A$ to $B$.

Todo: Implement homomorphisms from $A[x]$ to an arbitrary ring $R$, given by a ring homomorphism from $A$ to $R$ and the image of $x$ in $R$.

AUTHORS:

• Peter Bruin (March 2014): initial version

class

sage.rings.polynomial.polynomial_ring_homomorphism.PolynomialRingHomomorphism_from_base

Bases: sage.rings.morphism.RingHomomorphism_from_base

The canonical ring homomorphism from $R[x]$ to $S[x]$ induced by a ring homomorphism from $R$ to $S$.

EXAMPLES:

```python
sage: QQ['x'].coerce_map_from(ZZ['x'])
Ring morphism:
    From: Univariate Polynomial Ring in x over Integer Ring
    To:   Univariate Polynomial Ring in x over Rational Field
    Defn: Induced from base ring by
           Natural morphism:
           From: Integer Ring
           To:   Rational Field
```

is_injective()

Return whether this morphism is injective.

EXAMPLES:

```python
sage: R.<x> = ZZ[]
sage: S.<x> = QQ[]
sage: R.hom(S).is_injective()
True
```

is_surjective()

Return whether this morphism is surjective.

EXAMPLES:

```python
sage: R.<x> = ZZ[]
sage: S.<x> = Zmod(2)[]
sage: R.hom(S).is_surjective()
True
```
2.1.3 Univariate Polynomial Base Class

AUTHORS:

• William Stein: first version.
• Martin Albrecht: Added singular coercion.
• Robert Bradshaw: Move Polynomial_generic_dense to Cython.
• Miguel Marco: Implemented resultant in the case where PARI fails.
• Simon King: Use a faster way of conversion from the base ring.
• Julian Rueth (2012-05-25,2014-05-09): Fixed is_squarefree() for imperfect fields, fixed division without remainder over QQbar; added _cache_key for polynomials with unhashable coefficients
• Kiran Kedlaya (2016-03): Added root counting.
• Edgar Costa (2017-07): Added rational reconstruction.
• Kiran Kedlaya (2017-09): Added reciprocal transform, trace polynomial.
• David Zureick-Brown (2017-09): Added is_weil_polynomial.
• Sebastian Oehms (2018-10): made roots() and factor() work over more cases of proper integral domains (see trac ticket #26421)
• Kiran Kedlaya (2016-03): Added root counting.
• Edgar Costa (2017-07): Added rational reconstruction.
• Kiran Kedlaya (2017-09): Added reciprocal transform, trace polynomial.
• David Zureick-Brown (2017-09): Added is_weil_polynomial.
• Sebastian Oehms (2018-10): made roots() and factor() work over more cases of proper integral domains (see trac ticket #26421)

class sage.rings.polynomial.polynomial_element.ConstantPolynomialSection
Bases: sage.categories.map.Map

This class is used for conversion from a polynomial ring to its base ring.

Since trac ticket #9944, it calls the constant_coefficient method, which can be optimized for a particular polynomial type.

EXAMPLES:

```python
sage: P0.<y_1> = GF(3)[]
sage: P1.<y_2,y_1,y_0> = GF(3)[]
sage: P0(-y_1) # indirect doctest
2*y_1
sage: phi = GF(3).convert_map_from(P0); phi
Generic map:
  From: Univariate Polynomial Ring in y_1 over Finite Field of size 3
  To:   Finite Field of size 3
sage: type(phi)
<type 'sage.rings.polynomial.polynomial_element.ConstantPolynomialSection'>
sage: phi(P0.one())
1
sage: phi(y_1)
Traceback (most recent call last):
  ... TypeError: not a constant polynomial
```

class sage.rings.polynomial.polynomial_element.Polynomial
Bases: sage.structure.element.CommutativeAlgebraElement

A polynomial.
EXAMPLES:

```
sage: R.<y> = QQ['y']
sage: S.<x> = R['x']
sage: S
Univariate Polynomial Ring in x over Univariate Polynomial Ring in y
    over Rational Field
sage: f = x*y; f
    y*x
sage: type(f)
<type 'sage.rings.polynomial.polynomial_element.Polynomial_generic_dense'>
sage: p = (y+1)^10; p
    1024
```

_add_ (right)

Add two polynomials.

```sage
sage: R = ZZ['x']
sage: p = R([1,2,3,4])
sage: q = R([4,-3,2,-1])
sage: p + q
    3*x^3 + 5*x^2 - x + 5
```

_sub_ (other)

Default implementation of subtraction using addition and negation.

```sage
sage: R.<x> = ZZ[]
sage: f = (x^3 + x + 5)
sage: f._lmul_(7)
    7*x^3 + 7*x + 35
sage: 7*f
    7*x^3 + 7*x + 35
```

_mul_ (right)

Multiply self on the right by a scalar.

```sage
sage: R.<x> = ZZ[]
sage: f = (x^3 + x + 5)
sage: f._rmul_(7)
    7*x^3 + 7*x + 35
sage: f*7
    7*x^3 + 7*x + 35
```

_mul_ (right)

```sage
sage: R.<x> = ZZ[]
sage: f = (x^3 + x + 5)
sage: f._rmul_(7)
    7*x^3 + 7*x + 35
sage: f*7
    7*x^3 + 7*x + 35
```
```python
sage: R.<x> = ZZ
sage: (x - 4)*(x^2 - 8*x + 16)
x^3 - 12*x^2 + 48*x - 64
sage: C.<t> = PowerSeriesRing(ZZ)
sage: z = (1 + O(t)) + t*s^2
sage: z*z
t^2*s^4 + (2*t + O(t^2))*s^2 + 1 + O(t)
```

## More examples from trac 2943, added by Kiran S. Kedlaya 2 Dec 09
```python
sage: C.<t> = PowerSeriesRing(Integers())
sage: D.<s> = PolynomialRing(C)
sage: z = 1 + (t + O(t^2))*s + (t^2 + O(t^3))*s^2
sage: z*z
(t^4 + O(t^5))*s^4 + (2*t^3 + O(t^4))*s^3 + (3*t^2 + O(t^3))*s^2 + (2*t + O(t^2))*s + 1
```

### _mul_trunc_(right, n)

Return the truncated multiplication of two polynomials up to n.

This is the default implementation that does the multiplication and then truncate! There are custom implementations in several subclasses:

- on dense polynomial over integers (via FLINT)
- on dense polynomial over Z/nZ (via FLINT)
- on dense rational polynomial (via FLINT)
- on dense polynomial on Z/nZ (via NTL)

**EXAMPLES:**
```python
sage: R = QQ['x']['y']
sage: y = R.gen()
sage: x = R.base_ring().gen()
sage: p1 = 1 - x*y + 2*y**3
sage: p2 = -1/3 + y**5
sage: p1._mul_trunc_(p2, 5)
-2/3*y^3 + 1/3*x*y - 1/3
```

**Todo:** implement a generic truncated Karatsuba and use it here.

### adams_operator(n, monic=False)

Return the polynomial whose roots are the n-th power of the roots of this.

**INPUT:**

- n – an integer
- monic – boolean (default False) if set to True, force the output to be monic

**EXAMPLES:**
```python
sage: f = cyclotomic_polynomial(30)
sage: f.adams_operator(7)==f
```
sage: f.adams_operator(6) == cyclotomic_polynomial(5)**2
True
sage: f.adams_operator(10) == cyclotomic_polynomial(3)**4
True
sage: f.adams_operator(15) == cyclotomic_polynomial(2)**8
True
sage: f.adams_operator(30) == cyclotomic_polynomial(1)**8
True

sage: x = polygen(QQ)
sage: f = x^2-2*x+2
sage: f.adams_operator(10)
x^2 + 1024

When $f$ is monic the output will have leading coefficient $\pm 1$ depending on the degree, but we can force it to be monic:

sage: R.<a,b,c> = ZZ[]
sage: x = polygen(R)
sage: f = (x-a)*(x-b)*(x-c)
sage: f.adams_operator(3).factor()
(-1) * (x - c^3) * (x - b^3) * (x - a^3)
sage: f.adams_operator(3,monic=True).factor()
(x - c^3) * (x - b^3) * (x - a^3)

add_bigoh(prec)
Return the power series of precision at most prec got by adding $O(q^{\text{prec}})$ to self, where $q$ is its variable.

EXAMPLES:

sage: R.<x> = ZZ[]
sage: f = 1 + 4*x + x^3
sage: f.add_bigoh(7)
1 + 4*x + x^3 + O(x^7)
sage: f.add_bigoh(2)
1 + 4*x + O(x^2)
sage: f.add_bigoh(2).parent()
Power Series Ring in x over Integer Ring

all_roots_in_interval(a=None, b=None)
Return True if the roots of this polynomial are all real and contained in the given interval.

EXAMPLES:

sage: R.<x> = PolynomialRing(ZZ)
sage: pol = (x-1)^2 * (x-2)^2 * (x-3)
sage: pol.all_roots_in_interval(1, 3)
True
sage: pol.all_roots_in_interval(1.01, 3)
False
sage: pol = chebyshev_T(5,x)
sage: pol.all_roots_in_interval(-1,1)
True

(continues on next page)
sage: pol = chebyshev_T(5, x/2)
sage: pol.all_roots_in_interval(-1,1)
False
sage: pol.all_roots_in_interval()
True

any_root(ring=None, degree=None, assume_squarefree=False)
Return a root of this polynomial in the given ring.

INPUT:

• ring – The ring in which a root is sought. By default this is the coefficient ring.
• degree (None or nonzero integer) – Used for polynomials over finite fields. Return a root of degree abs(degree) over the ground field. If negative, also assumes that all factors of this polynomial are of degree abs(degree). If None, returns a root of minimal degree contained within the given ring.
• assume_squarefree (bool) – Used for polynomials over finite fields. If True, this polynomial is assumed to be squarefree.

EXAMPLES:

sage: R.<x> = GF(11)[]
sage: f = 7*x^7 + 8*x^6 + 4*x^5 + x^4 + 6*x^3 + 10*x^2 + 8*x + 5
sage: f.any_root()
2
sage: f.factor()
(7) * (x + 9) * (x^6 + 10*x^4 + 6*x^3 + 5*x^2 + 2*x + 2)
sage: f = x^6 + 10*x^4 + 6*x^3 + 5*x^2 + 2*x + 2
sage: f.any_root(GF(11^6, 'a'))
a^5 + a^4 + 7*a^3 + 2*a^2 + 10*a
sage: sorted(f.roots(GF(11^6, 'a')))
[(10*a^5 + 2*a^4 + 8*a^3 + 9*a^2 + 2*a + 1, 1),
 (a^5 + a^4 + 7*a^3 + 2*a^2 + 10*a, 1),
 ...
 (9*a^5 + 5*a^4 + 10*a^3 + 3*a^2 + 1, 1),
 (2*a^5 + 8*a^4 + 3*a^3 + 6*a^2 + 2, 1),
 (a^5 + 3*a^4 + 8*a^3 + 2*a^2 + 3*a + 4, 1),
 (10*a^5 + 3*a^4 + 8*a^3 + 2*a^2 + 3*a + 4, 1),
 (2*a^5 + 8*a^4 + 3*a^3 + 6*a^2 + 2, 1),
 (a^5 + 3*a^4 + 8*a^3 + 2*a^2 + 3*a + 4, 1)]

sage: f.any_root(GF(11^6, 'a'))
a^5 + a^4 + 7*a^3 + 2*a^2 + 10*a

sage: g = (x-1)*(x^2 + 3*x + 9) * (x^5 + 5*x^4 + 8*x^3 + 5*x^2 + 3*x + 5)
sage: g.any_root(ring=GF(11^10, 'b'), degree=1)
1
sage: g.any_root(ring=GF(11^10, 'b'), degree=2)
5*b^9 + 4*b^7 + 4*b^6 + 8*b^5 + 10*b^2 + 10*b + 5
sage: g.any_root(ring=GF(11^10, 'b'), degree=5)
5*b^9 + b^8 + 3*b^7 + 2*b^6 + b^5 + 4*b^4 + 3*b^3 + 7*b^2 + 10*b

args()  
Return the generator of this polynomial ring, which is the (only) argument used when calling self.

EXAMPLES:

sage: R.<x> = QQ[]
sage: x.args()
(x,)
A constant polynomial has no variables, but still takes a single argument.

```sage
R(2).args()
(x,)
```

**base_extend**(*R*)

Return a copy of this polynomial but with coefficients in R, if there is a natural map from coefficient ring of self to R.

**EXAMPLES:**

```sage
R.<x> = QQ[]
f = x^3 - 17*x + 3
f.base_extend(GF(7))
Traceback (most recent call last):
  ...
TypeError: no such base extension
f.change_ring(GF(7))
x^3 + 4*x + 3
```

**base_ring()**

Return the base ring of the parent of self.

**EXAMPLES:**

```sage
R.<x> = ZZ[]
x.base_ring()
Integer Ring
(2*x+3).base_ring()
Integer Ring
```

**change_ring**(*R*)

Return a copy of this polynomial but with coefficients in R, if at all possible.

**INPUT:**

- R - a ring or morphism.

**EXAMPLES:**

```sage
K.<z> = CyclotomicField(3)
f = K.defining_polynomial()
f.change_ring(GF(7))
x^2 + x + 1
```

```sage
K.<z> = CyclotomicField(3)
R.<x> = K[]
f = x^2 + z
f.change_ring(K.embeddings(CC)[1])
x^2 - 0.500000000000000 - 0.866025403784439*I
```

```sage
R.<x> = QQ[]
f = x^2 + 1
f.change_ring(QQ.embeddings(CC)[0])
x^2 + 1.00000000000000
```
**change_variable_name**(*var*)

Return a new polynomial over the same base ring but in a different variable.

**EXAMPLES:**

```python
sage: x = polygen(QQ,'x')
sage: f = -2/7*x^3 + (2/3)*x - 19/993; f
-2/7*x^3 + 2/3*x - 19/993
sage: f.change_variable_name('theta')
-2/7*theta^3 + 2/3*theta - 19/993
```

**coefficients**(*sparse=True*)

Return the coefficients of the monomials appearing in self. If sparse=True (the default), it returns only the non-zero coefficients. Otherwise, it returns the same value as self.list(). (In this case, it may be slightly faster to invoke self.list() directly.)

**EXAMPLES:**

```python
sage: _.<x> = PolynomialRing(ZZ)
sage: f = x^4+2*x^2+1
sage: f.coefficients()
[1, 2, 1]
sage: f.coefficients(sparse=False)
[1, 0, 2, 0, 1]
```

**complex_roots**()

Return the complex roots of this polynomial, without multiplicities.

Calls self.roots(ring=CC), unless this is a polynomial with floating-point coefficients, in which case it is uses the appropriate precision from the input coefficients.

**EXAMPLES:**

```python
sage: x = polygen(ZZ)
sage: (x^3 - 1).complex_roots()  # note: low order bits slightly different on __ppc_
[1.00000000000000, -0.500000000000000 - 0.86602540378443...*I, -0.500000000000000 + 0.86602540378443...*I]
```

**compose_power**(*k, algorithm=None, monic=False*)

Return the *k*-th iterate of the composed product of this polynomial with itself.

**INPUT:**

- *k* – a non-negative integer
- algorithm – None (default),"resultant" or "BFSS". See composed_op()
- monic - False (default) or True. See composed_op()

**OUTPUT:**

The polynomial of degree *d*^k where *d* is the degree, whose roots are all *k*-fold products of roots of this polynomial. That is, *f* * f * ··· * f where this is *f* and *f* * f = f.composed_op(f.operator.mul).

**EXAMPLES:**

```python
sage: R.<a,b,c> = ZZ[]
sage: x = polygen(R)
```

(continues on next page)
sage: f = (x-a)*(x-b)*(x-c)
sage: f.compose_power(2).factor()
(x - c^2) * (x - b^2) * (x - a^2) * (x - b*c)^2 * (x - a*c)^2 * (x - a*b)^2

sage: x = polygen(QQ)
sage: f2 = f.compose_power(2); f2
x^4 - 4*x^3 + 8*x^2 - 16*x + 16
sage: f2 == f.composed_op(f,operator.mul)
True
sage: f3 = f.compose_power(3); f3
x^8 - 8*x^7 + 32*x^6 - 64*x^5 + 128*x^4 - 512*x^3 + 2048*x^2 - 4096*x + 4096
sage: f3 == f2.composed_op(f,operator.mul)
True
sage: f4 = f.compose_power(4)
sage: f4 == f3.composed_op(f,operator.mul)
True

**compose_trunc**(other, n)

Return the composition of self and other, truncated to $O(x^n)$.

This method currently works for some specific coefficient rings only.

**EXAMPLES:**

```python
sage: Pol.<x> = CBF[]
sage: (1 + x + x^2/2 + x^3/6 + x^4/24 + x^5/120).compose_trunc(1 + x, 2)
([2.708333333333333 +/- ...e-16])*x + [2.71666666666667 +/- ...e-15]
sage: Pol.<x> = QQ['y'][]
sage: (1 + x + x^2/2 + x^3/6 + x^4/24 + x^5/120).compose_trunc(1 + x, 2)
Traceback (most recent call last):
... NotImplementedError: truncated composition is not implemented for this subclass, ...
```

**composed_op**(p1, p2, algorithm=None, monic=False)

Return the composed sum, difference, product or quotient of this polynomial with another one.

In the case of two monic polynomials $p_1$ and $p_2$ over an integral domain, the composed sum, difference, etc. are given by

$$\prod_{p_1(a)=p_2(b)=0} (x - (a * b)), \quad * \in \{+,-,\times, /\}$$

where the roots $a$ and $b$ are to be considered in the algebraic closure of the fraction field of the coefficients and counted with multiplicities. If the polynomials are not monic this quantity is multiplied by $\alpha_1^{deg(p_2)} \alpha_2^{deg(p_1)}$ where $\alpha_1$ and $\alpha_2$ are the leading coefficients of $p_1$ and $p_2$ respectively.

**INPUT:**

- p2 -- univariate polynomial belonging to the same polynomial ring as this polynomial
• `op` – `operator.OP` where `OP=add` or `sub` or `mul` or `truediv`.

• `algorithm` – can be “resultant” or “BFSS”; by default the former is used when the polynomials have few nonzero coefficients and small degrees or if the base ring is not `Z` or `Q`. Otherwise the latter is used.

• `monic` – whether to return a monic polynomial. If `True` the coefficients of the result belong to the fraction field of the coefficients.

ALGORITHM:
The computation is straightforward using resultants. Indeed for the composed sum it would be \( Res_y(p_1(x-y),p_2(y)) \). However, the method from [BFSS2006] using series expansions is asymptotically much faster. Note that the algorithm BFSS with polynomials with coefficients in `Z` needs to perform operations over `Q`.

Todo:
• The [BFSS2006] algorithm has been implemented here only in the case of polynomials over rationals. For other rings of zero characteristic (or if the characteristic is larger than the product of the degrees), one needs to implement a generic method `_exp_series`. In the general case of non-zero characteristic there is an alternative algorithm in the same paper.

• The Newton series computation can be done much more efficiently! See [BFSS2006].

EXAMPLES:

```
sage: x = polygen(ZZ)
sage: p1 = x^2 - 1
sage: p2 = x^4 - 1
sage: p1.composed_op(p2, operator.add)
x^8 - 4*x^6 + 4*x^4 - 16*x^2
sage: p1.composed_op(p2, operator.mul)
x^8 - 2*x^4 + 1
sage: p1.composed_op(p2, operator.truediv)
x^8 - 2*x^4 + 1
```

This function works over any field. However for base rings other than `Z` and `Q` only the resultant algorithm is available:

```
sage: x = polygen(QQbar)
sage: p1 = x^2 - AA(2).sqrt()
```

(continues on next page)
constant_coefficient()
Return the constant coefficient of this polynomial.

OUTPUT: element of base ring

EXAMPLES:

sage: R.<x> = QQ[]
 sage: f = -2*x^3 + 2*x - 1/3
 sage: f.constant_coefficient()
 -1/3

content_ideal()
Return the content ideal of this polynomial, defined as the ideal generated by its coefficients.

EXAMPLES:

sage: R.<x> = IntegerModRing(4)[]
 sage: f = x^4 + 3*x^2 + 2
 sage: f.content_ideal()
 Ideal (2, 3, 1) of Ring of integers modulo 4

When the base ring is a gcd ring, the content as a ring element is the generator of the content ideal:

sage: R.<x> = ZZ[]
 sage: f = 2*x^3 - 4*x^2 + 6*x - 10
 sage: f.content_ideal().gen()
 2

cyclotomic_part()
Return the product of the irreducible factors of this polynomial which are cyclotomic polynomials.

The algorithm assumes that the polynomial has rational coefficients.

See also:

is_cyclotomic() is_cyclotomic_product() has_cyclotomic_factor()
EXAMPLES:

```python
sage: P.<x> = PolynomialRing(Integers())
sage: pol = 2*(x^4 + 1)
sage: pol.cyclotomic_part()
\(x^4 + 1\)
sage: pol = x^4 + 2
sage: pol.cyclotomic_part()
1
sage: pol = (x^4 + 1)^2 * (x^4 + 2)
\(x^8 + 2*x^4 + 1\)
```

```python
sage: P.<x> = PolynomialRing(QQ)
sage: pol = (x^4 + 1)^2 * (x^4 + 2)
\(x^8 + 2*x^4 + 1\)
sage: pol = (x - 1) * x * (x + 2)
\(x - 1\)
```

`degree(gen=None)`

Return the degree of this polynomial. The zero polynomial has degree -1.

EXAMPLES:

```python
sage: x = ZZ['x'].0
sage: f = x^93 + 2*x + 1
sage: f.degree()
93
sage: x = PolynomialRing(QQ, 'x', sparse=True).0
sage: f = x^100000
sage: f.degree()
100000
```

```python
sage: x = QQ['x'].0
sage: f = 2006*x^2006 - x^2 + 3
sage: f.degree()
2006
sage: f = 0*x
sage: f.degree()
-1
sage: f = x + 33
sage: f.degree()
1
```

AUTHORS:

- Naqi Jaffery (2006-01-24): examples

`denominator()`

Return a denominator of self.

First, the lcm of the denominators of the entries of self is computed and returned. If this computation fails, the unit of the parent of self is returned.

2.1. Univariate Polynomials and Polynomial Rings
Note that some subclasses may implement their own denominator function. For example, see `sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint`

**Warning:** This is not the denominator of the rational function defined by self, which would always be 1 since self is a polynomial.

**EXAMPLES:**

First we compute the denominator of a polynomial with integer coefficients, which is of course 1.

```python
sage: R.<x> = ZZ[

sage: f = x^3 + 17*x + 1

sage: f.denominator()
1
```

Next we compute the denominator of a polynomial with rational coefficients.

```python
sage: R.<x> = PolynomialRing(QQ)

sage: f = (1/17)*x^19 - (2/3)*x + 1/3; f
1/17*x^19 - 2/3*x + 1/3

sage: f.denominator()
51
```

Finally, we try to compute the denominator of a polynomial with coefficients in the real numbers, which is a ring whose elements do not have a denominator method.

```python
sage: R.<x> = RR[

sage: f = x + RR('0.3'); f
x + 0.300000000000000

sage: f.denominator()
1.00000000000000
```

Check that the denominator is an element over the base whenever the base has no denominator function. This closes trac ticket #9063.

```python
sage: R.<a> = GF(5)[

sage: x = R(0)

sage: x.denominator()
1

sage: type(x.denominator())
<type 'sage.rings.finite_rings.integer_mod.IntegerMod_int'>

sage: isinstance(x.numerator() / x.denominator(), Polynomial)
True

sage: isinstance(x.numerator() / R(1), Polynomial)
False
```

**derivative(**`*args`**)**

The formal derivative of this polynomial, with respect to variables supplied in args.

Multiple variables and iteration counts may be supplied; see documentation for the global derivative() function for more details.

**See also:**

`_derivative()`
EXAMPLES:

```sage
sage: R.<x> = PolynomialRing(QQ)
sage: g = -x^4 + x^2/2 - x
sage: g.derivative()
-4*x^3 + x - 1
sage: g.derivative(x)
-4*x^3 + x - 1
sage: g.derivative(x, x)
-12*x^2 + 1
sage: g.derivative(x, 2)
-12*x^2 + 1
```

```sage
sage: R.<t> = PolynomialRing(ZZ)
sage: S.<x> = PolynomialRing(R)
sage: f = t^3*x^2 + t^4*x^3
sage: f.derivative()
3*t^4*x^2 + 2*t^3*x
sage: f.derivative(x)
3*t^4*x^2 + 2*t^3*x
sage: f.derivative(t)
4*t^3*x^3 + 3*t^2*x^2
```

```dict()
Return a sparse dictionary representation of this univariate polynomial.
```

```sage
sage: R.<x> = QQ[]
sage: f = x^3 + -1/7*x + 13
sage: f.dict()
{0: 13, 1: -1/7, 3: 1}
```

```diff(*args)
The formal derivative of this polynomial, with respect to variables supplied in args.

Multiple variables and iteration counts may be supplied; see documentation for the global derivative() function for more details.
```

```sage
sage: R.<x> = PolynomialRing(QQ)
sage: g = -x^4 + x^2/2 - x
sage: g.derivative()
-4*x^3 + x - 1
sage: g.derivative(x)
-4*x^3 + x - 1
sage: g.derivative(x, x)
-12*x^2 + 1
sage: g.derivative(x, 2)
-12*x^2 + 1
```
sage: R.<t> = PolynomialRing(ZZ)
sage: S.<x> = PolynomialRing(R)
sage: f = t^3*x^2 + t^4*x^3
sage: f.derivative()
3*t^4*x^2 + 2*t^3*x
sage: f.derivative(x)
3*t^4*x^2 + 2*t^3*x
sage: f.derivative(t)
4*t^3*x^3 + 3*t^2*x^2

**differentiate(*args)**

The formal derivative of this polynomial, with respect to variables supplied in args.

Multiple variables and iteration counts may be supplied; see documentation for the global derivative() function for more details.

See also:

_._derivative()

EXAMPLES:

sage: R.<x> = PolynomialRing(QQ)
sage: g = -x^4 + x^2/2 - x
sage: g.derivative()
-4*x^3 + x - 1
sage: g.derivative(x)
-4*x^3 + x - 1
sage: g.derivative(x, x)
-12*x^2 + 1
sage: g.derivative(x, 2)
-12*x^2 + 1

sage: R.<t> = PolynomialRing(ZZ)
sage: S.<x> = PolynomialRing(R)
sage: f = t^3*x^2 + t^4*x^3
sage: f.derivative()
3*t^4*x^2 + 2*t^3*x
sage: f.derivative(x)
3*t^4*x^2 + 2*t^3*x
sage: f.derivative(t)
4*t^3*x^3 + 3*t^2*x^2

**discriminant()**

Return the discriminant of self.

The discriminant is

\[ R_n := a_n^{2n-2} \prod_{1<i<j<n} \left( r_i - r_j \right)^2, \]

where \( n \) is the degree of self, \( a_n \) is the leading coefficient of self and the roots of self are \( r_1, \ldots, r_n \).

OUTPUT: An element of the base ring of the polynomial ring.

ALGORITHM:

Uses the identity \( R_n(f) := (-1)^{n(n-1)/2} R(f, f') a_n^{n-k-2} \), where \( n \) is the degree of self, \( a_n \) is the leading coefficient of self, \( f' \) is the derivative of \( f \), and \( k \) is the degree of \( f' \). Calls resultant().
EXAMPLES:

In the case of elliptic curves in special form, the discriminant is easy to calculate:

```python
sage: R.<x> = QQ[]
sage: f = x^3 + x + 1
sage: d = f.discriminant(); d
-31
sage: d.parent() is QQ
True
sage: EllipticCurve([1, 1]).discriminant()/16
-31
```

```python
sage: R.<x> = QQ[]
sage: f = 2*x^3 + x + 1
sage: d = f.discriminant(); d
-116
```

We can compute discriminants over univariate and multivariate polynomial rings:

```python
sage: R.<a> = QQ[]
sage: S.<x> = R[]
sage: f = a*x + x + a + 1
sage: d = f.discriminant(); d
1
sage: d.parent() is R
True
```

```python
sage: R.<a, b> = QQ[]
sage: S.<x> = R[]
sage: f = x^2 + a + b
sage: d = f.discriminant(); d
-4*a - 4*b
sage: d.parent() is R
True
```

`dispersion(other=None)`

Compute the dispersion of a pair of polynomials.

The dispersion of $f$ and $g$ is the largest nonnegative integer $n$ such that $f(x + n)$ and $g(x)$ have a nonconstant common factor.

When `other` is `None`, compute the auto-dispersion of `self`, i.e., its dispersion with itself.

See also:

`dispersion_set()`

EXAMPLES:

```python
sage: Pol.<x> = QQ[]
sage: x.dispersion(x + 1)
1
sage: (x + 1).dispersion(x)
-Infinity
```
```python
sage: Pol.<x> = QQbar[]
sage: pol = Pol([sqrt(5), 1, 3/2])
sage: pol.dispersion()
\theta
sage: (pol*pol(x+3)).dispersion()
3
```

dispersion_set(other=None)

Compute the dispersion set of two polynomials.

The dispersion set of \( f \) and \( g \) is the set of nonnegative integers \( n \) such that \( f(x + n) \) and \( g(x) \) have a nonconstant common factor.

When \( other \) is \( None \), compute the auto-dispersion set of \( self \), i.e., its dispersion set with itself.

ALGORITHM:
See Section 4 of Man & Wright [MW1994].

See also:
dispersion()

EXAMPLES:

```python
sage: Pol.<x> = QQ[]
sage: x.dispersion_set(x + 1)
[1]
sage: (x + 1).dispersion_set(x)
[]

sage: pol = x^3 + x - 7
sage: (pol*pol(x+3)^2).dispersion_set()
[0, 3]
```
divides(p)

Return \( True \) if this polynomial divides \( p \).

This method is only implemented for polynomials over an integral domain.

EXAMPLES:

```python
sage: R.<x> = ZZ[]
sage: (2*x + 1).divides(4*x^2 - 1)
True
sage: (2*x + 1).divides(4*x^2 + 1)
False
sage: (2*x + 1).divides(R(0))
True
sage: R(0).divides(2*x + 1)
False
sage: R(0).divides(R(0))
True
sage: S.<y> = R[]
sage: p = x*y^2 + (2*x + 1)*y + x + 1
sage: q = (x + 1)*y + (3*x + 2)
sage: q.divides(p)
```

(continues on next page)
euclidean_degree()

Return the degree of this element as an element of an Euclidean domain.

If this polynomial is defined over a field, this is simply its degree().

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: x.euclidean_degree()
1
sage: R.<x> = ZZ[]
sage: x.euclidean_degree()
Traceback (most recent call last):
...  
NotImplementedError
```

exponents()

Return the exponents of the monomials appearing in self.

EXAMPLES:

```
sage: _.<x> = PolynomialRing(ZZ)
sage: f = x^4+2*x^2+1
sage: f.exponents()
[0, 2, 4]
```

factor(**kwargs)

Return the factorization of self over its base ring.

INPUT:

- **kwargs – any keyword arguments are passed to the method _factor_univariate_polynomial() of the base ring if it defines such a method.

OUTPUT:

- A factorization of self over its parent into a unit and irreducible factors. If the parent is a polynomial ring over a field, these factors are monic.

EXAMPLES:

Factorization is implemented over various rings. Over Q:

```
sage: x = QQ['x'].0
sage: f = (x^3 - 1)^2
```
\texttt{sage}: \texttt{f.factor()}
\begin{verbatim}
(x - 1)^2 \times (x^2 + x + 1)^2
\end{verbatim}

Since $\mathbb{Q}$ is a field, the irreducible factors are monic:

\texttt{sage}: \texttt{f = 10*x^5 - 1}
\texttt{sage}: \texttt{f.factor()}
\begin{verbatim}
(10) \times (x^5 - 1/10)
\end{verbatim}
\texttt{sage}: \texttt{f = 10*x^5 - 10}
\texttt{sage}: \texttt{f.factor()}
\begin{verbatim}
(10) \times (x - 1) \times (x^4 + x^3 + x^2 + x + 1)
\end{verbatim}

Over $\mathbb{Z}$ the irreducible factors need not be monic:

\texttt{sage}: \texttt{x = ZZ['x'].0}
\texttt{sage}: \texttt{f = 10*x^5 - 1}
\texttt{sage}: \texttt{f.factor()}
\begin{verbatim}
10*x^5 - 1
\end{verbatim}

We factor a non-monic polynomial over a finite field of 25 elements:

\texttt{sage}: \texttt{k.<a> = GF(25)}
\texttt{sage}: \texttt{R.<x> = k[]}
\texttt{sage}: \texttt{f = 2*x^10 + 2*x + 2*a}
\texttt{sage}: \texttt{F = f.factor(); F}
\begin{verbatim}
(2) \times (x + a + 2) \times (x^2 + 3*x + 4*a + 4) \times (x^2 + (a + 1)*x + a + 2) \times (x^5 + \rightarrow (3*a + 4)*x^4 + (3*a + 3)*x^3 + 2*a^2*x^2 + (3*a + 1)^2*x + 3*a + 1)
\end{verbatim}

Notice that the unit factor is included when we multiply $F$ back out:

\texttt{sage}: \texttt{expand(F)}
\begin{verbatim}
2*x^10 + 2*x + 2*a
\end{verbatim}

A new ring. In the example below, we set the special method \_factor_univariate_polynomial() in the base ring which is called to factor univariate polynomials. This facility can be used to easily extend polynomial factorization to work over new rings you introduce:

\texttt{sage}: \texttt{R.<x> = PolynomialRing(IntegerModRing(4),implementation="NTL")}
\texttt{sage}: \texttt{f = x^2}
\texttt{sage}: \texttt{f.factor()}
\begin{verbatim}
Traceback (most recent call last):
... Not ImplementedError: factorization of polynomials over rings with composite␣ \texttt{-characteristic is not implemented}
\end{verbatim}
\texttt{sage}: \texttt{R.base_ring()._factor_univariate_polynomial = lambda f: f.change_ring(ZZ).factor()}
\texttt{sage}: \texttt{f = x^2}
\texttt{sage}: \texttt{f.factor()}
\begin{verbatim}
x^2
\end{verbatim}
\texttt{sage}: \texttt{del R.base_ring()._factor_univariate_polynomial # clean up}

Arbitrary precision real and complex factorization:

\texttt{sage}: \texttt{R.<x> = RealField(100)[]}
\texttt{sage}: \texttt{F = factor(x^2-3); F}
(continued from previous page)

\[(x - 1.7320508075688772935274463415) \times (x + 1.7320508075688772935274463415)\]

```
sage: expand(F)
x^2 - 3.0000000000000000000000000000
```

```
sage: factor(x^2 + 1)
x^2 + 1.0000000000000000000000000000
```

```
sage: R.<x> = ComplexField(100)[]
sage: F = factor(x^2+3); F
(x - 1.7320508075688772935274463415*I) \times (x + 1.7320508075688772935274463415*I)
sage: expand(F)
x^2 + 3.0000000000000000000000000000
```

```
sage: factor(x^2+1)
(x - I) \times (x + I)
sage: f = R(I) * (x^2 + 1); f
I*x^2 + I
sage: F = factor(f); F
(1.0000000000000000000000000000*I) \times (x - I) \times (x + I)
sage: expand(F)
I*x^2 + I
```

Over a number field:

```
sage: K.<z> = CyclotomicField(15)
sage: x = polygen(K)
sage: ((x^3 + z*x + 1)^3*(x - z)).factor()
(x - z) \times (x^3 + z*x + 1)^3
```

```
sage: cyclotomic_polynomial(12).change_ring(K).factor()
(x^2 - z^5 - 1) \times (x^2 + z^5)
```

```
sage: ((x^3 + z*x + 1)^3*(x/(z+2) - 1/3)).factor()
(-1/331*z^7 + 3/331*z^6 - 6/331*z^5 + 11/331*z^4 - 21/331*z^3 + 41/331*z^2 - 82/331*z + 165/331) \times (x - 1/3*z - 2/3) \times (x^3 + z*x + 1)^3
```

Over a relative number field:

```
sage: x = polygen(QQ)
sage: K.<z> = CyclotomicField(3)
sage: L.<a> = K.extension(x^3 - 2)
sage: t = polygen(L, 't')
sage: f = (t^3 + t + a)*(t^5 + t + z); f
t^8 + t^6 + a*t^5 + t^4 + z*t^3 + t^2 + (a + z)*t + z*a
sage: f.factor()
(t^3 + t + a) \times (t^5 + t + z)
```

Over the real double field:

```
sage: R.<x> = RDF[]
sage: (-2*x^2 - 1).factor()
(-2.0) \times (x^2 + 0.5000000000000001)
sage: (-2*x^2 - 1).factor().expand()
-2.0*x^2 - 1.0000000000000002
sage: f = (x - 1)^3
sage: f.factor()  # abs tol 2e-5
(x - 1.00000065719436413) \times (x^2 - 1.999934280563585*x + 0.999934280995487)
```

2.1. Univariate Polynomials and Polynomial Rings 49
The above output is incorrect because it relies on the `roots()` method, which does not detect that all the roots are real:

```
sage: f.roots()  # abs tol 2e-5
[(1.0000006571943613, 1)]
```

Over the complex double field the factors are approximate and therefore occur with multiplicity 1:

```
sage: R.<x> = CDF[]
sage: f = (x^2 + 2*R(I))^3
sage: F = f.factor()
sage: F  # abs tol 3e-5
(x - 1.0000138879287663 + 1.0000013435286879*I) *
(x - 0.9999918923847313 + 1.0000113554909125*I) *
(x + 0.99990875955227 - 1.000069659624138*I) *
(x + 0.999985293216753 - 0.9999886153831807*I) *
(x + 1.00000105947233 - 1.0000044186544053*I)
sage: [f(t[0][0]).abs() for t in F]  # abs tol 1e-13
[1.979365054e-14, 1.97936298566e-14, 1.97936990747e-14, 3.6812407475e-14, 3.65220890052e-14]
```

Factoring polynomials over \( \mathbb{Z}/n\mathbb{Z} \) for composite \( n \) is not implemented:

```
sage: R.<x> = PolynomialRing(Integers(35))
sage: f = (x^2+2*x+2)*(x^2+3*x+9)
sage: f.factor()
Traceback (most recent call last):
  ...NotImplementedError: factorization of polynomials over rings with composite characteristic is not implemented
```

Factoring polynomials over the algebraic numbers (see trac ticket #8544):

```
sage: R.<x> = QQbar[]
sage: (x^8-1).factor()
(x - 1) * (x - 0.7071067811865475? - 0.7071067811865475?*I) *
(x - 0.7071067811865475? + 0.7071067811865475?*I) *
(x - 0.7071067811865475? - 0.7071067811865475?*I) *
(x + 0.7071067811865475? + 0.7071067811865475?*I) *
(x + 0.7071067811865475? - 0.7071067811865475?*I) *
(x + 1)
```

Factoring polynomials over the algebraic reals (see trac ticket #8544):

```
sage: R.<x> = AA[]
sage: (x^8+1).factor()
(x^2 - 1.84775906522574?*x + 1.0000000000000000)*
(x^2 + 0.7653668647301795? + 1.0000000000000000)*
(x^2 - 0.7653668647301795?*x + 1.0000000000000000)
```

\[ \gcd(\text{other}) \]

Return a greatest common divisor of this polynomial and other.

**INPUT:**

- **other** – a polynomial in the same ring as this polynomial

**OUTPUT:**
A greatest common divisor as a polynomial in the same ring as this polynomial. If the base ring is a field, the return value is a monic polynomial.

**Note:** The actual algorithm for computing greatest common divisors depends on the base ring underlying the polynomial ring. If the base ring defines a method `_gcd_univariate_polynomial`, then this method will be called (see examples below).

**EXAMPLES:**

```python
sage: R.<x> = QQ[]
sage: (2*x^2).gcd(2*x)
x
sage: R.zero().gcd(0)
0
sage: (2*x).gcd(0)
x
```

One can easily add gcd functionality to new rings by providing a method `_gcd_univariate_polynomial`:

```python
sage: O = ZZ[-sqrt(5)]
sage: R.<x> = O[]
sage: a = O.1
sage: p = x + a
sage: q = x^2 - 5
sage: p.gcd(q)
Traceback (most recent call last):
  ...  
NotImplementedError: Order in Number Field in a with defining polynomial x^2 -> 5 with a = -2.236067977499790? does not provide a gcd implementation for univariate polynomials
sage: S.<x> = O.number_field()[]
sage: O._gcd_univariate_polynomial = lambda f,g : R(S(f).gcd(S(g)))
sage: p.gcd(q)
x + a
sage: del O._gcd_univariate_polynomial
```

Use multivariate implementation for polynomials over polynomials rings:

```python
sage: R.<x> = ZZ[]
sage: S.<y> = R[]
sage: T.<z> = S[]
sage: r = 2*x*y + z
sage: p = r * (3*x*y*z - 1)
sage: q = r * (x + y + z - 2)
sage: p.gcd(q)
z + 2*x*y
sage: R.<x> = QQ[]
sage: S.<y> = R[]
sage: r = Zx^2 + 1
sage: p = r * (x - 1/2 * y)
sage: q = r * (x*y^2 - x + 1/3)
sage: p.gcd(q)
```

(continues on next page)
\[ 2 \times x \times y + 1 \]

**gradient()**

Return a list of the partial derivative of `self` with respect to the variable of this univariate polynomial.

There is only one partial derivative.

**EXAMPLES:**

```python
sage: P.<x> = QQ[]
sage: f = x^2 + (2/3)*x + 1
sage: f.gradient()
[2\times x + 2/3]
sage: f = P(1)
sage: f.gradient()
[0]
```

**hamming_weight()**

Return the number of non-zero coefficients of `self`.

Also called weight, Hamming weight or sparsity.

**EXAMPLES:**

```python
sage: R.<x> = ZZ[]
sage: f = x^3 - x
sage: f.number_of_terms()
2
sage: R(0).number_of_terms()
0
sage: f = (x+1)^100
sage: f.number_of_terms()
101
sage: S = GF(5)['y']
sage: S(f).number_of_terms()
5
sage: cyclotomic_polynomial(105).number_of_terms()
33
```

The method `hamming_weight()` is an alias:

```python
sage: f.hamming_weight()
101
```

**has_cyclotomic_factor()**

Return True if the given polynomial has a nontrivial cyclotomic factor.

The algorithm assumes that the polynomial has rational coefficients.

If the polynomial is known to be irreducible, it may be slightly more efficient to call `is_cyclotomic` instead.

**See also:**

`is_cyclotomic()` `is_cyclotomic_product()` `cyclotomic_part()`

**EXAMPLES:**
homogenize(var='h')

Return the homogenization of this polynomial.

The polynomial itself is returned if it is homogeneous already. Otherwise, its monomials are multiplied
with the smallest powers of var such that they all have the same total degree.

INPUT:

• var – a variable in the polynomial ring (as a string, an element of the ring, or 0) or a name for a new
variable (default: ‘h’)

OUTPUT:

If var specifies the variable in the polynomial ring, then a homogeneous element in that ring is returned. Otherwise, a homogeneous element is returned in a polynomial ring with an extra last variable var.

EXAMPLES:

```python
sage: R.<x> = QQ[]
sage: f = x^2 + 1
sage: f.homogenize()
x^2 + h^2
```

The parameter var can be used to specify the name of the variable:

```python
sage: g = f.homogenize('z'); g
x^2 + z^2
sage: g.parent()
Multivariate Polynomial Ring in x, z over Rational Field
```

However, if the polynomial is homogeneous already, then that parameter is ignored and no extra variable
is added to the polynomial ring:

```python
sage: f = x^2
sage: g = f.homogenize('z'); g
x^2
sage: g.parent()
Univariate Polynomial Ring in x over Rational Field
```

For compatibility with the multivariate case, if var specifies the variable of the polynomial ring, then the
monomials are multiplied with the smallest powers of var such that the result is homogeneous; in other
words, we end up with a monomial whose leading coefficient is the sum of the coefficients of the polynomial:

```python
sage: f = x^2 + x + 1
sage: f.homogenize('x')
3*x^2
```

In positive characteristic, the degree can drop in this case:
For compatibility with the multivariate case, the parameter `var` can also be 0 to specify the variable in the polynomial ring:

```sage
sage: R.<x> = QQ[]
sage: f = x^2 + x + 1
sage: f.homogenize(0)
3*x^2
```

`integral(var=None)`

Return the integral of this polynomial.

By default, the integration variable is the variable of the polynomial. Otherwise, the integration variable is the optional parameter `var`.

**Note:** The integral is always chosen so that the constant term is 0.

**EXAMPLES:**

```sage
sage: R.<x> = ZZ[]
sage: R(0).integral()
0
sage: f = R(2).integral(); f
2*x
```

Note that the integral lives over the fraction field of the scalar coefficients:

```sage
sage: f.parent()
Univariate Polynomial Ring in x over Rational Field
sage: R(0).integral().parent()
Univariate Polynomial Ring in x over Rational Field
sage: f = x^3 + x - 2
sage: g = f.integral(); g
1/4*x^4 + 1/2*x^2 - 2*x
sage: g.parent()
Univariate Polynomial Ring in x over Rational Field
```

This shows that the issue at trac ticket #7711 is resolved:

```sage
sage: P.<x,z> = PolynomialRing(GF(2147483647))
sage: Q.<y> = PolynomialRing(P)
sage: p=x+y+z
sage: p.integral()
-1073741823*y^2 + (x + z)*y
sage: sage: P.<x,z> = PolynomialRing(GF(next_prime(2147483647)))
```

(continues on next page)
... (continued from previous page)...

```bash
sage: p=x+y+z
sage: p.integral()
1073741830*y^2 + (x + z)*y
```

A truly convoluted example:

```bash
sage: A.<a1, a2> = PolynomialRing(ZZ)
sage: B.<b> = PolynomialRing(A)
sage: C.<c> = PowerSeriesRing(B)
sage: R.<x> = PolynomialRing(C)
sage: f = a2*x^2 + c*x - a1*b
sage: f.parent()
Univariate Polynomial Ring in x over Power Series Ring in c
over Univariate Polynomial Ring in b over Multivariate Polynomial
Ring in a1, a2 over Integer Ring
sage: f.integral()
1/3*a2*x^3 + 1/2*c*x^2 - a1*b*x
sage: f.integral().parent()
Univariate Polynomial Ring in x over Power Series Ring in c
over Univariate Polynomial Ring in b over Multivariate Polynomial
Ring in a1, a2 over Rational Field
sage: g = 3*a2*x^2 + 2*c*x - a1*b
sage: g.integral()
a2*x^3 + c*x^2 - a1*b*x
sage: g.integral().parent()
Univariate Polynomial Ring in x over Power Series Ring in c
over Univariate Polynomial Ring in b over Multivariate Polynomial
Ring in a1, a2 over Rational Field
```

Integration with respect to a variable in the base ring:

```bash
sage: R.<x> = QQ[]
sage: t = PolynomialRing(R, 't').gen()
sage: f = x*t + 5*t^2
sage: f.integral(x)
5*x*t^2 + 1/2*x^2*t
```

### inverse_mod(a, m)

Inverts the polynomial a with respect to m, or raises a ValueError if no such inverse exists. The parameter m may be either a single polynomial or an ideal (for consistency with inverse_mod in other rings).

**See also:**

If you are only interested in the inverse modulo a monomial \(x^k\) then you might use the specialized method `inverse_series_trunc()` which is much faster.

**EXAMPLES:**

```bash
sage: S.<t> = QQ[]
sage: f = inverse_mod(t^2 + 1, t^3 + 1); f
```

... (continues on next page)
\[-t^6 - 7*t^5 - 21*t^4 - 35*t^3 - 35*t^2 - 21*t - 7\]
\[
sage: (f * t) + (t+1)^7
1
\]
\[
sage: t.inverse_mod(S.ideal((t + 1)^7)) == f
True
\]

This also works over inexact rings, but note that due to rounding error the product may not always exactly equal the constant polynomial 1 and have extra terms with coefficients close to zero.

\[
sage: R.<x> = RDF[]
sage: epsilon = RDF(1).ulp()*50  # Allow an error of up to 50 ulp
sage: f = inverse_mod(x^2 + 1, x^5 + x + 1); f  # abs tol 1e-14
0.4*x^4 - 0.2*x^3 - 0.4*x^2 + 0.2*x + 0.8
sage: poly = f * (x^2 + 1) % (x^5 + x + 1)
sage: # Remove noisy zero terms:
sage: parent(poly)([ 0.0 if abs(c)<=epsilon else c for c in poly.˓→coefficients(sparse=False) ])
1.0
\]

ALGORITHM: Solve the system as + mt = 1, returning s as the inverse of a mod m.

Uses the Euclidean algorithm for exact rings, and solves a linear system for the coefficients of s and t for inexact rings (as the Euclidean algorithm may not converge in that case).

AUTHORS:


inverse_of_unit()

EXAMPLES:

\[
sage: R.<x> = QQ[]
sage: f = x - 90283
sage: f.inverse_of_unit()
Traceback (most recent call last):
...
ArithmeticError: x - 90283 is not a unit in Univariate Polynomial Ring in x over Rational Field
\]
\[
sage: f = R(-90283); g = f.inverse_of_unit(); g
-1/90283
sage: parent(g)
Univariate Polynomial Ring in x over Rational Field
\]
inverse_series_trunc(prec)

Return a polynomial approximation of precision prec of the inverse series of this polynomial.

See also:
The method inverse_mod() allows more generally to invert this polynomial with respect to any ideal.

EXAMPLES:

```
sage: x = polygen(ZZ)
sage: s = (1+x).inverse_series_trunc(5)
sage: s
x^4 - x^3 + x^2 - x + 1
sage: s * (1+x)
x^5 + 1
```

Note that the constant coefficient needs to be a unit:

```
sage: ZZx.<x> = ZZ[]
sage: ZZxy.<y> = ZZx[]
sage: (1+x + y**2).inverse_series_trunc(4)
Traceback (most recent call last):
... ValueError: constant term x + 1 is not a unit
sage: (1+x + y**2).change_ring(ZZx.fraction_field()).inverse_series_trunc(4)
(-1/(x^2 + 2*x + 1))*y^2 + 1/(x + 1)
```

The method works over any polynomial ring:

```
sage: R = Zmod(4)
sage: Rx.<x> = R[]
sage: Rxy.<y> = Rx[]
```

```
sage: p = 1 + (1+2*x)*y + x**2*y**4
sage: q = p.inverse_series_trunc(10)
sage: (p*q).truncate(11)
(2*x^4 + 3*x^2 + 3)*y^10 + 1
```

Even noncommutative ones:

```
sage: M = MatrixSpace(ZZ,2)
sage: x = polygen(M)
sage: p = M([1,2,3,4])*x^3 + M([-1,0,1,0])*x^2 + M([1,1,-1,0])*x + M.one()
sage: q = p.inverse_series_trunc(5)
sage: (p*q).truncate(5) == M.one()
True
sage: q = p.inverse_series_trunc(13)
sage: (p*q).truncate(13) == M.one()
True
```

AUTHORS:
- David Harvey (2006-09-09): Newton’s method implementation for power series
- Vincent Delecroix (2014-2015): move the implementation directly in polynomial

is_constant()

Return True if this is a constant polynomial.
OUTPUT:

• `bool` - True if and only if this polynomial is constant

EXAMPLES:

```python
sage: R.<x> = ZZ[]
sage: x.is_constant()
False
sage: R(2).is_constant()
True
sage: R(0).is_constant()
True
```

```

is_cyclotomic(certificate=False, algorithm='pari')
```

Test if this polynomial is a cyclotomic polynomial.

A cyclotomic polynomial is a monic, irreducible polynomial such that all roots are roots of unity.

By default the answer is a boolean. But if `certificate` is `True`, the result is a non-negative integer: it is

0 if `self` is not cyclotomic, and a positive integer \( n \) if `self` is the \( n \)-th cyclotomic polynomial.

See also:

- `is_cyclotomic_product()`
- `cyclotomic_part()`
- `has_cyclotomic_factor()`

INPUT:

• `certificate` – boolean, default to `False`. Only works with `algorithm` set to “pari”.

• `algorithm` – either “pari” or “sage” (default is “pari”)

ALGORITHM:

The native algorithm implemented in Sage uses the first algorithm of [BD1989]. The algorithm in pari (using `pari:poliscyclo`) is more subtle since it does compute the inverse of the Euler \( \phi \) function to determine the \( n \) such that the polynomial is the \( n \)-th cyclotomic polynomial.

EXAMPLES:

Quick tests:

```python
sage: P.<x> = ZZ['x']
sage: (x - 1).is_cyclotomic()
True
sage: (x + 1).is_cyclotomic()
True
sage: (x^2 - 1).is_cyclotomic()
False
sage: (x^2 + x + 1).is_cyclotomic(certificate=True)
3
sage: (x^2 + 2^x + 1).is_cyclotomic(certificate=True)
0
```

Test first 100 cyclotomic polynomials:

```python
sage: all(cyclotomic_polynomial(i).is_cyclotomic() for i in range(1,101))
True
```

Some more tests:
```python
sage: (x^16 + x^14 - x^10 + x^8 - x^6 + x^2 + 1).is_cyclotomic(algorithm="pari")
False
sage: (x^16 + x^14 - x^10 + x^8 - x^6 + x^2 + 1).is_cyclotomic(algorithm="sage")
False
sage: (x^16 + x^14 - x^10 - x^8 - x^6 + x^2 + 1).is_cyclotomic(algorithm="pari")
True
sage: (x^16 + x^14 - x^10 - x^8 - x^6 + x^2 + 1).is_cyclotomic(algorithm="sage")
True
sage: y = polygen(QQ)
sage: (y/2 - 1/2).is_cyclotomic()
False
sage: (2*(y/2 - 1/2)).is_cyclotomic()
True
```

Invalid arguments:
```
sage: (x - 3).is_cyclotomic(algorithm="sage", certificate=True)
Traceback (most recent call last):
  ...
ValueError: no implementation of the certificate within Sage
```

Test using other rings:
```
sage: z = polygen(GF(5))
sage: (z - 1).is_cyclotomic()
Traceback (most recent call last):
  ...
NotImplementedError: not implemented in non-zero characteristic
```

**is_cyclotomic_product()**

Test whether this polynomial is a product of cyclotomic polynomials.

This method simply calls the function `pari:poliscycloprod` from the Pari library.

**See also:**

`is_cyclotomic()` `cyclotomic_part()` `has_cyclotomic_factor()`

**EXAMPLES:**
```
sage: x = polygen(ZZ)
sage: (x^5 - 1).is_cyclotomic_product()
True
sage: (x^5 + x^4 - x^2 + 1).is_cyclotomic_product()
False
sage: p = prod(cyclotomic_polynomial(i) for i in [2,5,7,12])
sage: p.is_cyclotomic_product()
True
sage: (x^5 - 1/3).is_cyclotomic_product()
False
```

(continues on next page)
```python
sage: x = polygen(Zmod(5))
sage: (x-1).is_cyclotomic_product()
Traceback (most recent call last):
...
NotImplementedError: not implemented in non-zero characteristic
```

**is_gen()**

Return True if this polynomial is the distinguished generator of the parent polynomial ring.

**EXAMPLES:**

```python
sage: R.<x> = QQ[]
sage: R(1).is_gen()
False
sage: R(x).is_gen()
True
```

Important - this function doesn’t return True if self equals the generator; it returns True if self *is* the generator.

```python
sage: f = R([0,1]); f
x
sage: f.is_gen()
False
sage: f is x
False
sage: f == x
True
```

**is_homogeneous()**

Return True if this polynomial is homogeneous.

**EXAMPLES:**

```python
sage: P.<x> = PolynomialRing(QQ)
sage: x.is_homogeneous()
True
sage: P(0).is_homogeneous()
True
sage: (x+1).is_homogeneous()
False
```

**is_irreducible()**

Return whether this polynomial is irreducible.

**EXAMPLES:**

```python
sage: R.<x> = ZZ[]
sage: (x^3 + 1).is_irreducible()
False
sage: (x^2 - 1).is_irreducible()
False
sage: (x^3 + 2).is_irreducible()
True
```
sage: R(0).is_irreducible()
False

The base ring does matter: for example, $2x$ is irreducible as a polynomial in $\mathbb{Q}[x]$, but not in $\mathbb{Z}[x]$:

sage: R.<x> = ZZ[]
sage: R(2*x).is_irreducible()
False
sage: R.<x> = QQ[]
sage: R(2*x).is_irreducible()
True

is_monic()
Returns True if this polynomial is monic. The zero polynomial is by definition not monic.

EXAMPLES:

sage: x = QQ['x'].0
sage: f = x + 33
sage: f.is_monic()
True
sage: f = 0*x
sage: f.is_monic()
False
sage: f = 3*x^3 + x^4 + x^2
sage: f.is_monic()
True
sage: f = 2*x^2 + x^3 + 56*x^5
sage: f.is_monic()
False

AUTHORS:

• Naqi Jaffery (2006-01-24): examples

is_monomial()
Return True if self is a monomial, i.e., a power of the generator.

EXAMPLES:

sage: R.<x> = QQ[]
sage: x.is_monomial()
True
sage: (x+1).is_monomial()
False
sage: (x^2).is_monomial()
True
sage: R(1).is_monomial()
True

The coefficient must be 1:

sage: (2*x^5).is_monomial()
False

To allow a non-1 leading coefficient, use is_term():
is_term()

Return True if this polynomial is a term.

EXAMPLES:

```sage
sage: (2*x^5).is_term()
True
```

Warning: The definition of is_monomial in Sage up to 4.7.1 was the same as is_term, i.e., it allowed a coefficient not equal to 1.

is_nilpotent()

Return True if this polynomial is nilpotent.

EXAMPLES:

```sage
sage: R = Integers(12)
sage: S.<x> = R[]
sage: f = 5 + 6*x
sage: f.is_nilpotent()
False
sage: f = 6 + 6*x^2
sage: f.is_nilpotent()
True
sage: f^2
0
```

EXERCISE (Atiyah-McDonald, Ch 1): Let \( A[x] \) be a polynomial ring in one variable. Then \( f = \sum a_i x^i \in A[x] \) is nilpotent if and only if every \( a_i \) is nilpotent.

is_one()

Test whether this polynomial is 1.

EXAMPLES:

```sage
sage: R.<x> = QQ[]
sage: (x-3).is_one()
False
sage: R(1).is_one()
True
sage: R2.<y> = R[]
sage: R2(x).is_one()
False
sage: R2(1).is_one()
True
sage: R2(-1).is_one()
False
```

is_primitive(n=None, n_prime_divs=None)

Return True if the polynomial is primitive. The semantics of “primitive” depend on the polynomial coefficients.

- (field theory) A polynomial of degree \( m \) over a finite field \( \mathbb{F}_q \) is primitive if it is irreducible and its root in \( \mathbb{F}_{q^m} \) generates the multiplicative group \( \mathbb{F}_{q^m}^* \).
- (ring theory) A polynomial over a ring is primitive if its coefficients generate the unit ideal.

Calling isPrimitive on a polynomial over an infinite field will raise an error.
The additional inputs to this function are to speed up computation for field semantics (see note).

**INPUT:**

- **n** (default: None) - if provided, should equal \( q - 1 \) where `self.parent()` is the field with \( q \) elements; otherwise it will be computed.
- **n_prime_divs** (default: None) - if provided, should be a list of the prime divisors of \( n \); otherwise it will be computed.

**Note:** Computation of the prime divisors of \( n \) can dominate the running time of this method, so performing this computation externally (e.g. `pdivs=n.prime_divisors()`) is a good idea for repeated calls to `is_primitive` for polynomials of the same degree.

Results may be incorrect if the wrong \( n \) and/or factorization are provided.

**EXAMPLES:**

Field semantics examples.

```python
::
sage: R.<x> = GF(2)['x']
sage: f = x^4+x^3+x^2+x+1
sage: f.is_irreducible(), f.is_primitive()
(True, False)
sage: f = x^3+x+1
sage: f.is_irreducible(), f.is_primitive()
(True, True)
sage: R.<x> = GF(3)[]
sage: f = x^3-x+1
sage: f.is_irreducible(), f.is_primitive()
(True, True)
sage: f = x^2+1
sage: f.is_irreducible(), f.is_primitive()
(True, False)
sage: R.<x> = GF(5)[]
sage: f = x^2+x+1
sage: f.isPrimitive()
False
sage: f = x^2-x+2
sage: f.isPrimitive()
True
sage: x=polygen(QQ); f=x^2+1
sage: f.isPrimitive()
Traceback (most recent call last):
...
NotImplementedError: isPrimitive() not defined for polynomials over infinite,
˓→fields.
```

Ring semantics examples.

```python
::
```

(continues on next page)
```python
sage: x=polygen(ZZ)
sage: f = 5*x^2+2
sage: f.is_primitive()
True
sage: f = 5*x^2+5
sage: f.is_primitive()
False

sage: K=NumberField(x^2+5,'a')
sage: R=K.ring_of_integers()
sage: a=R.gen(1)
sage: a^2
-5
sage: f=a*x+2
sage: f.is_primitive()
True
sage: f=(1+a)*x+2
sage: f.is_primitive()
False

sage: x = polygen(Integers(10))
sage: f = 5*x^2+2
sage: #f.is_primitive() #BUG:: elsewhere in Sage, should return True
sage: f=4*x^2+2
sage: #f.is_primitive() #BUG:: elsewhere in Sage, should return False
```

**is_real_rooted()**

Return True if the roots of this polynomial are all real.

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(ZZ)
sage: pol = chebyshev_T(5, x)
sage: pol.is_real_rooted()
True
sage: pol = x^2 + 1
sage: pol.is_real_rooted()
False
```

**is_square(root=False)**

Return whether or not polynomial is square.

If the optional argument root is set to True, then also returns the square root (or None, if the polynomial is not square).

INPUT:

- **root** - whether or not to also return a square root (default: False)

OUTPUT:

- **bool** - whether or not a square
- **root** - (optional) an actual square root if found, and None otherwise.

EXAMPLES:
```python
sage: R.<x> = PolynomialRing(QQ)
sage: (x^2 + x + 1).is_square()
True
sage: (x^4 + x^3 - x^2 - 2*x + 1).is_square(root=True)
(True, x^2 + x - 1)

sage: f = 12*(x+1)^2 * (x+3)^2
sage: f.is_square()
False
sage: f.is_square(root=True)
(False, None)

sage: h = f/3; h
4*x^4 + 32*x^3 + 88*x^2 + 96*x + 36
sage: h.is_square(root=True)
(True, 2*x^2 + 8*x + 6)

sage: S.<y> = PolynomialRing(RR)
sage: g = 12*(y+1)^2 * (y+3)^2
sage: g.is_square()
True

is_squarefree()  
Return False if this polynomial is not square-free, i.e., if there is a non-unit g in the polynomial ring such that g^2 divides self.

Warning: This method is not consistent with squarefree_decomposition() since the latter does not factor the content of a polynomial. See the examples below.

EXAMPLES:

sage: R.<x> = QQ[]
sage: f = (x-1)*(x-2)*(x^2-5)*(x^17-3); f
x^21 - 3*x^20 - 3*x^19 + 15*x^18 - 10*x^17 - 3*x^4 + 9*x^3 + 9*x^2 - 45*x + 30
sage: f.is_squarefree()
True
sage: (f*(x^2-5)).is_squarefree()
False

A generic implementation is available, which relies on gcd computations:

sage: R.<x> = ZZ[]
sage: (2*x).is_squarefree()
True
sage: (4*x).is_squarefree()
False
sage: (2*x^2).is_squarefree()
False
sage: R(0).is_squarefree()
False
sage: S.<y> = QQ[]
```

(continues on next page)
In positive characteristic, we compute the square-free decomposition or a full factorization, depending on which is available:

```python
sage: R.<x> = S[]
sage: (2*x*y).is_squarefree()
True
sage: (2*x*y^2).is_squarefree()
False
```

In the following example, \( t^2 \) is a unit in the base field:

```python
sage: K.<t> = FunctionField(GF(3))
sage: R.<x> = K[]
sage: (x^3-x).is_squarefree()
True
sage: (x^3-1).is_squarefree()
False
sage: (x^3+t).is_squarefree()
True
sage: (x^3+t^3).is_squarefree()
False
```

This method is not consistent with `squarefree_decomposition()`:

```python
sage: R.<x> = ZZ[]
sage: f = 4 * x
sage: f.is_squarefree()
False
sage: f.squarefree_decomposition()
(4) * x
```

If you want this method equally not to consider the content, you can remove it as in the following example:

```python
sage: c = f.content()
sage: (f/c).is_squarefree()
True
```

If the base ring is not an integral domain, the question is not mathematically well-defined:

```python
sage: R.<x> = IntegerModRing(9)[]
sage: pol = (x + 3)*(x + 6); pol
x^2
sage: pol.is_squarefree()
Traceback (most recent call last):
...  TypeError: is_squarefree() is not defined for polynomials over Ring of integers modulo 9
```

Return `True` if this polynomial is a nonzero element of the base ring times a power of the variable.
EXAMPLES:

```
sage: R.<x> = QQ[]
sage: x.is_term()
  True
sage: R(0).is_term()
  False
sage: R(1).is_term()
  True
sage: (3*x^5).is_term()
  True
sage: (1+3*x^5).is_term()
  False
```

To require that the coefficient is 1, use `is_monomial()` instead:

```
sage: (3*x^5).is_monomial()
  False
```

`is_unit()`

Return True if this polynomial is a unit.

EXAMPLES:

```
sage: a = Integers(90384098234^3)
sage: b = a(2*191*236607587)
sage: b.is_nilpotent()
  True
sage: R.<x> = a[]
sage: f = 3 + b*x + b^2*x^2
sage: f.is_unit()
  True
sage: f = 3 + b*x + b^2*x^2 + 17*x^3
sage: f.is_unit()
  False
```

EXERCISE (Atiyah-McDonald, Ch 1): Let $A[x]$ be a polynomial ring in one variable. Then $f = \sum a_ix^i \in A[x]$ is a unit if and only if $a_0$ is a unit and $a_1, \ldots, a_n$ are nilpotent.

`is_weil_polynomial(return_q=False)`

Return True if this is a Weil polynomial.

This polynomial must have rational or integer coefficients.

INPUT:

- `self` – polynomial with rational or integer coefficients
- `return_q` – (default False) if True, return a second value $q$ which is the prime power with respect to which this is $q$-Weil, or 0 if there is no such value.

EXAMPLES:

```
sage: polRing.<x> = PolynomialRing(Rationals())
sage: P0 = x^4 + 5*x^3 + 15*x^2 + 25*x + 25
sage: P1 = x^4 + 25*x^3 + 15*x^2 + 5*x + 25
sage: P2 = x^4 + 5*x^3 + 25*x^2 + 25*x + 25
```

(continues on next page)
See also:

Polynomial rings have a method \texttt{weil\_polynomials} to compute sets of Weil polynomials. This computation uses the iterator \texttt{sage.rings.polynomial.weil\_polynomials.WeilPolynomials}.

AUTHORS:

David Zureick-Brown (2017-10-01)

\begin{Verbatim}
\texttt{sage: R = GF(2)[x][y]}
\texttt{sage: R([0,1]).is\_zero()}
\texttt{False}
\texttt{sage: R([0]).is\_zero()}
\texttt{True}
\texttt{sage: R([-1]).is\_zero()}
\texttt{False}
\end{Verbatim}

\begin{Verbatim}
\texttt{sage: R.<x> = QQ[]}
\texttt{sage: f = (-2/5)*x^3 + 2*x - 1/3}
\texttt{sage: f.lc()}
\texttt{-2/5}
\end{Verbatim}

1cm\texttt{(other)}

Let \(f\) and \(g\) be two polynomials. Then this function returns the monic least common multiple of \(f\) and \(g\).

\begin{Verbatim}
\texttt{sage: R.<x> = QQ[]}
\texttt{sage: f = (-2/5)*x^3 + 2*x - 1/3}
\texttt{sage: f.lc()}
\texttt{-2/5}
\end{Verbatim}

\begin{Verbatim}
\texttt{sage: R.<x> = QQ[]}
\texttt{sage: f = (-2/5)*x^3 + 2*x - 1/3}
\end{Verbatim}

(continues on next page)
\begin{verbatim}
sage: f.leading_coefficient()
-2/5

\textbf{list\(\{\text{copy=True}\)}

Return a new copy of the list of the underlying elements of \textit{self}.

EXAMPLES:

\begin{verbatim}
sage: R.<x> = QQ[]
sage: f = (-2/5)*x^3 + 2*x - 1/3
sage: v = f.list(); v
[-1/3, 2, 0, -2/5]
\end{verbatim}

Note that \texttt{v} is a list, it is mutable, and each call to the list method returns a new list:

\begin{verbatim}
sage: type(v)
<... 'list'>
sage: v[0] = 5
sage: f.list()
[-1/3, 2, 0, -2/5]
\end{verbatim}

Here is an example with a generic polynomial ring:

\begin{verbatim}
sage: R.<x> = QQ[]
sage: S.<y> = R[

sage: f = y^3 + x*y - 3*x; f
y^3 + x*y - 3*x
sage: type(f)
<type 'sage.rings.polynomial.polynomial_element.Polynomial_generic_dense'>
sage: v = f.list(); v
[-3*x, x, 0, 1]
sage: v[0] = 10
sage: f.list()
[-3*x, x, 0, 1]
\end{verbatim}

\textbf{lm()}

Return the leading monomial of this polynomial.

EXAMPLES:

\begin{verbatim}
sage: R.<x> = QQ[]
sage: f = (-2/5)*x^3 + 2*x - 1/3
sage: f.lm()
x^3
sage: R(5).lm()
1
sage: R(0).lm()
0
sage: R(0).lm().parent() is R
True
\end{verbatim}

\textbf{lt()}

Return the leading term of this polynomial.

EXAMPLES:

\end{verbatim}

\end{verbatim}

2.1. Univariate Polynomials and Polynomial Rings 69
```python
sage: R.<x> = QQ[]
sage: f = (-2/5)*x^3 + 2*x - 1/3
sage: f.lt()
-2/5*x^3
sage: R(5).lt()
5
sage: R(0).lt()
0
sage: R(0).lt().parent() is R
True
```

**map_coefficients** *(f, new_base_ring=None)*

Return the polynomial obtained by applying f to the non-zero coefficients of self.

If f is a sage.categories.map.Map, then the resulting polynomial will be defined over the codomain of f. Otherwise, the resulting polynomial will be over the same ring as self. Set new_base_ring to override this behaviour.

**INPUT:**

- f – a callable that will be applied to the coefficients of self.
- new_base_ring (optional) – if given, the resulting polynomial will be defined over this ring.

**EXAMPLES:**

```python
sage: R.<x> = SR[]
sage: f = (1+I)*x^2 + 3*x - I
sage: f.map_coefficients(lambda z: z.conjugate())
(-I + 1)*x^2 + 3*x + I
sage: R.<x> = ZZ[]
sage: f = x^2 + 2
sage: f.map_coefficients(lambda a: a + 42)
43*x^2 + 44
sage: R.<x> = PolynomialRing(SR, sparse=True)
sage: f = (1+I)*x^(2^32) - I
sage: f.map_coefficients(lambda z: z.conjugate())
(-I + 1)*x^4294967296 + I
sage: R.<x> = PolynomialRing(ZZ, sparse=True)
sage: f = x^(2^32) + 2
sage: f.map_coefficients(lambda a: a + 42)
43*x^4294967296 + 44
```

Examples with different base ring:

```python
sage: R.<x> = ZZ[]
sage: k = GF(2)
sage: residue = lambda x: k(x)
sage: f = 4*x^2+x+3
sage: g = f.map_coefficients(residue); g
x + 1
sage: g.parent()
Univariate Polynomial Ring in x over Integer Ring
sage: g = f.map_coefficients(residue, new_base_ring = k); g
x + 1
```

(continues on next page)


```python
sage: g.parent()
Univariate Polynomial Ring in x over Finite Field of size 2 (using GF2X)
sage: residue = k.coerce_map_from(ZZ)
sage: g = f.map_coefficients(residue); g
x + 1
sage: g.parent()
Univariate Polynomial Ring in x over Finite Field of size 2 (using GF2X)
```

**mod**(other)

Remainder of division of self by other.

EXAMPLES:

```python
sage: R.<x> = ZZ[]
sage: x % (x+1)
-1
sage: (x^3 + x - 1) % (x^2 - 1)
2*x - 1
```

**monic**()

Return this polynomial divided by its leading coefficient. Does not change this polynomial.

EXAMPLES:

```python
sage: x = QQ['x'].0
sage: f = 2*x^2 + x^3 + 56*x^5
sage: f.monic()
x^5 + 1/56*x^3 + 1/28*x^2
sage: f = (1/4)*x^2 + 3*x + 1
sage: f.monic()
x^2 + 12*x + 4
```

The following happens because $f = 0$ cannot be made into a monic polynomial

```python
sage: f = 0*x
sage: f.monic()
Traceback (most recent call last):
  ... ZeroDivisionError: rational division by zero
```

Notice that the monic version of a polynomial over the integers is defined over the rationals.

```python
sage: x = ZZ['x'].0
sage: f = 3*x^19 + x^2 - 37
sage: g = f.monic(); g
x^19 + 1/3*x^2 - 37/3
sage: g.parent()
Univariate Polynomial Ring in x over Rational Field
```

**AUTHORS:**

- Naqi Jaffery (2006-01-24): examples

**monomial_coefficient**(m)

Return the coefficient in the base ring of the monomial $m$ in $self$, where $m$ must have the same parent as $self$.  

2.1. Univariate Polynomials and Polynomial Rings 71
INPUT:

• m - a monomial

OUTPUT:

Coefficient in base ring.

EXAMPLES:

```sage
P.<x> = QQ[]

The parent of the return is a member of the base ring.
sage: f = 2 * x
sage: c = f.monomial_coefficient(x); c
2
sage: c.parent()
Rational Field

sage: f = x^9 - 1/2*x^2 + 7*x + 5/11
sage: f.monomial_coefficient(x^9)
1
sage: f.monomial_coefficient(x^2)
-1/2
sage: f.monomial_coefficient(x)
7
sage: f.monomial_coefficient(x^0)
5/11
sage: f.monomial_coefficient(x^3)
0
```

```sage
monomials()

Return the list of the monomials in self in a decreasing order of their degrees.

EXAMPLES:

```sage
P.<x> = QQ[]
sage: f = x^2 + (2/3)*x + 1
sage: f.monomials()
[x^2, x, 1]
sage: f = P(3/2)
sage: f.monomials()
[1]
sage: f = P(0)
sage: f.monomials()
[]
sage: f = x
sage: f.monomials()
[x]
sage: f = - 1/2*x^2 + x^9 + 7*x + 5/11
sage: f.monomials()
[x^9, x^2, x, 1]
sage: x = var('x')
sage: K.<rho> = NumberField(x^2 + 1)
sage: R.<y> = QQ[]
sage: p = rho*y
```

(continues on next page)
\texttt{sage: p.monomials()}
\begin{verbatim}
[y]
\end{verbatim}

\texttt{multiplication\_trunc(other, n)}

Truncated multiplication

\textbf{EXAMPLES:}

\begin{verbatim}
sage: R.<x> = ZZ[]
sage: (x^10 + 5*x^5 + x^2 - 3).multiplication_trunc(x^7 - 3*x^3 + 1, 11)
x^10 + x^9 - 15*x^8 - 3*x^7 + 2*x^5 + 9*x^3 + x^2 - 3
\end{verbatim}

Check that coercion is working:

\begin{verbatim}
sage: R2 = QQ['x']
sage: x2 = R2.gen()
sage: p1 = (x^3 + 1).multiplication_trunc(x2^3 - 2, 5); p1
-x^3 - 2
sage: p2 = (x2^3 + 1).multiplication_trunc(x^3 - 2, 5); p2
-x^3 - 2
sage: parent(p1) == parent(p2) == R2
True
\end{verbatim}

\texttt{newton\_raphson(n, x0)}

Return a list of \( n \) iterative approximations to a root of this polynomial, computed using the Newton-Raphson method.

The Newton-Raphson method is an iterative root-finding algorithm. For \( f(x) \) a polynomial, as is the case here, this is essentially the same as Horner's method.

\textbf{INPUT:}

- \( n \) - an integer (=the number of iterations),
- \( x0 \) - an initial guess \( x0 \).

\textbf{OUTPUT:} A list of numbers hopefully approximating a root of \( f(x)=0 \).

If one of the iterates is a critical point of \( f \) then a \texttt{ZeroDivisionError} exception is raised.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: x = PolynomialRing(RealField(), 'x').gen()
sage: f = x^2 - 2
sage: f.newton_raphson(4, 1)
[1.50000000000000, 1.41666666666667, 1.41421568627451, 1.41421356237469]
\end{verbatim}

\textbf{AUTHORS:}

- David Joyner and William Stein (2005-11-28)

\texttt{newton\_slopes(p, lengths=False)}

Return the \( p \)-adic slopes of the Newton polygon of self, when this makes sense.

\textbf{OUTPUT:}

If \texttt{lengths} is \texttt{False}, a list of rational numbers. If \texttt{lengths} is \texttt{True}, a list of couples \((s,l)\) where \( s \) is the slope and \( l \) the length of the corresponding segment in the Newton polygon.

\textbf{EXAMPLES:}
sage: x = QQ['x'].0
sage: f = x^3 + 2
sage: f.newton_slopes(2)
[1/3, 1/3, 1/3]
sage: R.<x> = PolynomialRing(ZZ, sparse=True)
sage: p = x^5 + 6*x^2 + 4
sage: p.newton_slopes(2)
[1/2, 1/2, 1/3, 1/3, 1/3]
sage: p.newton_slopes(2, lengths=True)
[(1/2, 2), (1/3, 3)]
sage: (x^2^100 + 27).newton_slopes(3, lengths=True)
[(3/1267650600228229401496703205376, 1267650600228229401496703205376)]

ALGORITHM: Uses PARI if lengths is False.

**norm**(*p*)
Return the *p*-norm of this polynomial.

**DEFINITION:** For integer *p*, the *p*-norm of a polynomial is the *p*th root of the sum of the *p*th powers of the absolute values of the coefficients of the polynomial.

**INPUT:**

- *p* - (positive integer or +infinity) the degree of the norm

**EXAMPLES:**

sage: R.<x> = RR[]
sage: f = x^6 + x^2 + -x^4 - 2*x^3
sage: f.norm(2)
2.64575131106459
sage: (sqrt(1^2 + 1^2 + (-1)^2 + (-2)^2)).n()
2.64575131106459
sage: f.norm(1)
5.00000000000000
sage: f.norm(infinity)
2.00000000000000
sage: f.norm(-1)
Traceback (most recent call last):
  ...
ValueError: The degree of the norm must be positive

**AUTHORS:**

- Didier Deshommes
- William Stein: fix bugs, add definition, etc.

**nth_root**(*n*)
Return a *n*-th root of this polynomial.

This is computed using Newton method in the ring of power series. This method works only when the base ring is an integral domain. Moreover, for polynomial whose coefficient of lower degree is different from 1, the elements of the base ring should have a method nth_root implemented.

**EXAMPLES:**
\begin{verbatim}
sage: R.<x> = ZZ[]
sage: a = 27 * (x+3)**6 * (x+5)**3
sage: a.nth_root(3)
3*x^3 + 33*x^2 + 117*x + 135

sage: b = 25 * (x^2 + x + 1)
sage: b.nth_root(2)
Traceback (most recent call last):
  ...  ValueError: not a 2nd power

sage: R0 = QQ[]
sage: a = 1/4 * (x/7 + 3/2)^2 * (x/2 + 5/3)^4
sage: a.nth_root(2)
1/56*x^3 + 103/336*x^2 + 365/252*x + 25/12

sage: K.<sqrt2> = QuadraticField(2)
sage: R.<x> = K[]
sage: a = (x + sqrt2)^3 * ((1+sqrt2)*x - 1/sqrt2)^6
sage: b = a.nth_root(3); b
(2*sqrt2 + 3)*x^3 + (2*sqrt2 + 2)*x^2 + (-2*sqrt2 - 3/2)*x + 1/2*sqrt2
sage: b^3 == a
True

sage: R.<x> = QQbar[]
sage: p = x**3 + QQbar(2).sqrt() * x - QQbar(3).sqrt()
sage: r = (p**5).nth_root(5)
sage: r * p[0] == p * r[0]
True

sage: p = (x+1)^20 + x^20
sage: p.nth_root(20)
Traceback (most recent call last):
  ...  ValueError: not a 20th power

sage: z = GF(4).gen()
sage: R.<x> = GF(4)[]
sage: p = z*x**4 + 2*x - 1
sage: r = (p**15).nth_root(15)
sage: r * p[0] == p * r[0]
True

sage: ((x+1)**2).nth_root(2)
x + 1

sage: ((x+1)**4).nth_root(4)
x + 1

sage: ((x+1)**12).nth_root(12)
x + 1

sage: (x^4 + x^3 + 1).nth_root(2)
Traceback (most recent call last):
  ...  ValueError: not a 2nd power

sage: p = (x+1)^17 + x^17
\end{verbatim}
Here we consider a base ring without `nth_root` method. The third example with a non-trivial coefficient of lowest degree raises an error:

```python
sage: R.<x> = QQ[]
sage: R2 = R.quotient(x**2 + 1)
sage: x = R2.gen()
sage: R3.<y> = R2[]
sage: (y**2 - 2*y + 1).nth_root(2)
-y + 1
sage: (y**3).nth_root(3)
y
sage: (y**2 + x).nth_root(2)
Traceback (most recent call last):
  ... AttributeError: ... has no attribute 'nth_root'
```

### `number_of_real_roots()`

Return the number of real roots of this polynomial, counted without multiplicity.

**EXAMPLES:**

```python
sage: R.<x> = PolynomialRing(ZZ)
sage: pol = (x-1)**2 * (x-2)**2 * (x-3)
sage: pol.number_of_real_roots()
3
sage: pol = (x-1)*(x-2)*(x-3)
sage: pol2 = pol.change_ring(CC)
sage: pol2.number_of_real_roots()
3
sage: R.<x> = PolynomialRing(CC)
sage: pol = (x-1)*(x-CC(I))
sage: pol.number_of_real_roots()
1
```

### `number_of_roots_in_interval(a=None, b=None)`

Return the number of roots of this polynomial in the interval \([a,b]\), counted without multiplicity. The endpoints \(a, b\) default to \(-\infty, \infty\) (which are also valid input values).

Calls the PARI routine `pari:polsturm`.

Note that as of version 2.8, PARI includes the left endpoint of the interval (and no longer uses Sturm’s algorithm on exact inputs). `polsturm` requires a polynomial with real coefficients; in case PARI returns an
error, we try again after taking the GCD of $self$ with its complex conjugate.

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(ZZ)
sage: pol = (x-1)^2 * (x-2)^2 * (x-3)
sage: pol.number_of_roots_in_interval(1, 2)
2
sage: pol.number_of_roots_in_interval(1.01, 2)
1
sage: pol.number_of_roots_in_interval(None, 2)
2
sage: pol.number_of_roots_in_interval(1, Infinity)
3
sage: pol.number_of_roots_in_interval()
3
sage: R.<x> = PolynomialRing(CC)
sage: pol2 = pol.change_ring(CC)
sage: pol2.number_of_roots_in_interval()
3
sage: pol = (x-1)*(x-2)*(x-3)
sage: pol2 = pol.change_ring(CC)
sage: pol2.number_of_roots_in_interval(0, 2)
1
```

**number_of_terms()**

Return the number of non-zero coefficients of $self$.

Also called weight, Hamming weight or sparsity.

EXAMPLES:

```python
sage: R.<x> = ZZ[]
sage: f = x^3 - x
sage: f.number_of_terms()
2
sage: R(0).number_of_terms()
0
sage: f = (x+1)^100
sage: f.number_of_terms()
101
sage: S = GF(5)['y']
sage: S(f).number_of_terms()
5
sage: cyclotomic_polynomial(105).number_of_terms()
33
```

The method `hamming_weight()` is an alias:

```python
sage: f.hamming_weight()
101
```

**numerator()**

Return a numerator of $self$ computed as $self * self.denominator()$

Note that some subclasses may implement its own numerator function. For example, see `sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint`.

### 2.1. Univariate Polynomials and Polynomial Rings
**Warning:** This is not the numerator of the rational function defined by self, which would always be self since self is a polynomial.

**EXAMPLES:**

First we compute the numerator of a polynomial with integer coefficients, which is of course self.

```
sage: R.<x> = ZZ[]
sage: f = x^3 + 17*x + 1
sage: f.numerator()
x^3 + 17*x + 1
sage: f == f.numerator()
True
```

Next we compute the numerator of a polynomial with rational coefficients.

```
sage: R.<x> = PolynomialRing(QQ)
sage: f = (1/17)*x^19 - (2/3)*x + 1/3; f
1/17*x^19 - 2/3*x + 1/3
sage: f.numerator()
3*x^19 - 34*x + 17
sage: f == f.numerator()
False
```

We try to compute the denominator of a polynomial with coefficients in the real numbers, which is a ring whose elements do not have a denominator method.

```
sage: R.<x> = RR[]
sage: f = x + RR('0.3'); f
x + 0.300000000000000
sage: f.numerator()
x + 0.300000000000000
```

We check that the computation the numerator and denominator are valid.

```
sage: K=NumberField(symbolic_expression('x^3+2'),'a')['s,t']['x']
sage: f=K.random_element()
sage: f.numerator() / f.denominator() == f
True
sage: R=RR['x']
sage: f=R.random_element()
sage: f.numerator() / f.denominator() == f
True
```

**ord** *(p=None)*

This is the same as the valuation of self at p. See the documentation for self.valueation.

**EXAMPLES:**

```
sage: R.<x> = ZZ[]
sage: (x^2+x).ord(x+1)
1
```

**padded_list** *(n=None)*

Return list of coefficients of self up to (but not including) $q^n$. 

---

Chapter 2. Univariate Polynomials
Includes 0's in the list on the right so that the list has length \( n \).

**INPUT:**

- \( n \) - (default: None); if given, an integer that is at least 0

**EXAMPLES:**

```python
sage: x = polygen(QQ)
sage: f = 1 + x^3 + 23*x^5
sage: f.padded_list()
[1, 0, 1, 0, 23]
sage: f.padded_list(10)
[1, 0, 0, 1, 0, 23, 0, 0, 0, 0]
sage: len(f.padded_list(10))
10
sage: f.padded_list(3)
[1, 0, 0]
sage: f.padded_list(0)
[]
sage: f.padded_list(-1)
Traceback (most recent call last):
  ...
ValueError: n must be at least 0
```

**plot**(*xmin=None, xmax=None, *args, **kwds*)

Return a plot of this polynomial.

**INPUT:**

- \( \text{xmin} \) - float
- \( \text{xmax} \) - float
- *args, **kwds* - passed to either plot or point

**OUTPUT:** returns a graphic object.

**EXAMPLES:**

```python
sage: x = polygen(GF(389))
sage: plot(x^2 + 1, rgbcolor=(0,0,1))
Graphics object consisting of 1 graphics primitive
sage: x = polygen(QQ)
sage: plot(x^2 + 1, rgbcolor=(1,0,0))
Graphics object consisting of 1 graphics primitive
```

**polynomial**(*var*)

Let \( \text{var} \) be one of the variables of the parent of self. This returns self viewed as a univariate polynomial in \( \text{var} \) over the polynomial ring generated by all the other variables of the parent.

For univariate polynomials, if \( \text{var} \) is the generator of the parent ring, we return this polynomial, otherwise raise an error.

**EXAMPLES:**

```python
sage: R.<x> = QQ[]
sage: (x+1).polynomial(x)
x + 1
```
**power_trunc(n, prec)**

Truncated n-th power of this polynomial up to precision prec

**INPUT:**

- n – (non-negative integer) power to be taken
- prec – (integer) the precision

**EXAMPLES:**

```python
sage: R.<x> = ZZ[]
sage: (3*x^2 - 2*x + 1).power_trunc(5, 8)
-1800*x^7 + 1590*x^6 - 1052*x^5 + 530*x^4 - 200*x^3 + 55*x^2 - 10*x + 1
sage: ((3*x^2 - 2*x + 1)^5).truncate(8)
-1800*x^7 + 1590*x^6 - 1052*x^5 + 530*x^4 - 200*x^3 + 55*x^2 - 10*x + 1
sage: S.<y> = R[]
sage: (x+y).power_trunc(5,5)
5*x*y^4 + 10*x^2*y^3 + 10*x^3*y^2 + 5*x^4*y + x^5
sage: ((x+y)^5).truncate(5)
5*x*y^4 + 10*x^2*y^3 + 10*x^3*y^2 + 5*x^4*y + x^5
sage: R.<x> = GF(3)[]
sage: p = x^2 - x + 1
sage: q = p.power_trunc(80, 20)
sage: q
x^19 + x^18 + ... + 2*x^4 + 2*x^3 + x + 1
sage: (p^80).truncate(20) == q
True
sage: R.<x> = GF(7)[]
sage: p = (x^2 + x + 1).power_trunc(2^100, 100)
sage: p
2*x^99 + x^98 + x^95 + 2*x^94 + ... + 3*x^2 + 2*x + 1
sage: for i in range(100):
    ....:     q1 = (x^2 + x + 1).power_trunc(2^100 + i, 100)
    ....:     q2 = p * (x^2 + x + 1).power_trunc(i, 100)
    ....:     q2 = q2.truncate(100)
    ....:     assert q1 == q2, "i = {}".format(i)
```

**prec()**

Return the precision of this polynomial. This is always infinity, since polynomials are of infinite precision by definition (there is no big-oh).

**EXAMPLES:**

```python
sage: x = polygen(ZZ)
sage: (x^5 + x + 1).prec()
+Infinity
sage: x.prec()
+Infinity
```

**pseudo_quo_rem(other)**

Compute the pseudo-division of two polynomials.
INPUT:

- other – a nonzero polynomial

OUTPUT:

$Q$ and $R$ such that $l^{m-n+1} \cdot \text{self} = Q \cdot \text{other} + R$ where $m$ is the degree of this polynomial, $n$ is the degree of other, $l$ is the leading coefficient of other. The result is such that $\deg(R) < \deg(\text{other})$.

ALGORITHM:

Algorithm 3.1.2 in [Coh1993].

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(ZZ, sparse=True)
sage: p = x^4 + 6*x^3 + x^2 - x + 2
sage: q = 2*x^2 - 3*x - 1
sage: (quo, rem) = p.pseudo_quo_rem(q); quo, rem
(4*x^2 + 30*x + 51, 175*x + 67)
sage: 2^(4-2+1)*p == quo*q + rem
True
sage: S.<T> = R[]
sage: p = (-3*x^2 - x)*T^3 - 3*x*T^2 + (x^2 - x)*T + 2*x^2 + 3*x - 2
sage: q = (-x^2 - 4*x - 5)*T^2 + (6*x^2 + x + 1)*T + 2*x^2 - x
sage: quo, rem = p.pseudo_quo_rem(q); quo, rem
((3*x^4 + 13*x^3 + 19*x^2 + 5*x)*T + 18*x^4 + 12*x^3 + 16*x^2 + 16*x, ~ (-113*x^6 - 106*x^5 - 133*x^4 - 101*x^3 - 42*x^2 - 41*x)*T - 34*x^6 + 13*x^5 + 54*x^4 + 126*x^3 + 134*x^2 - 5*x - 50)
sage: (-x^2 - 4*x - 5)^(3-2+1) * p == quo*q + rem
True
```

radical()

Return the radical of self.

Over a field, this is the product of the distinct irreducible factors of self. (This is also sometimes called the “square-free part” of self, but that term is ambiguous; it is sometimes used to mean the quotient of self by its maximal square factor.)

EXAMPLES:

```python
sage: P.<x> = ZZ[]
sage: t = (x^2-x+1)^3 * (3*x-1)^2
sage: t.radical()
3*x^3 - 4*x^2 + 4*x - 1
sage: radical(12 * x^5)
6*x
```

If self has a factor of multiplicity divisible by the characteristic (see trac ticket #8736):

```python
sage: P.<x> = GF(2)[]
sage: (x^3 + x^2).radical()
x^2 + x
```

rational_reconstruct($m, n\_deg=None, d\_deg=None$)

Return a tuple of two polynomials $(n, d)$ where self * d is congruent to n modulo m and n.degree() <= n_deg and d.degree() <= d_deg.
INPUT:

- \( m \) – a univariate polynomial
- \( n_{\text{deg}} \) – (optional) an integer; the default is \( \lfloor (\deg(m) - 1)/2 \rfloor \)
- \( d_{\text{deg}} \) – (optional) an integer; the default is \( \lfloor (\deg(m) - 1)/2 \rfloor \)

ALGORITHM:

The algorithm is based on the extended Euclidean algorithm for the polynomial greatest common divisor.

EXAMPLES:

Over \( \mathbb{Q}[z] \):

```python
sage: z = PolynomialRing(QQ, 'z').gen()
sage: p = -z**16 - z**15 - z**14 + z**13 + z**12 + z**11 - z**5 - z**4 - z**3 + z**2 + z + 1
sage: m = z**21
sage: n, d = p.rational_reconstruct(m)
sage: print((n, d))
(z^4 + 2*z^3 + 3*z^2 + 2*z + 1, z^10 + z^9 + z^8 + z^7 + z^6 + z^5 + z^4 + z^3 + z^2 + z + 1)
sage: print(((p*d - n) % m).is_zero())
True
```

Over \( \mathbb{Z}[z] \):

```python
sage: z = PolynomialRing(ZZ, 'z').gen()
sage: p = -z**16 - z**15 - z**14 + z**13 + z**12 + z**11 - z**5 - z**4 - z**3 + z**2 + z + 1
sage: m = z**21
sage: n, d = p.rational_reconstruct(m)
sage: print((n, d))
(z^4 + 2*z^3 + 3*z^2 + 2*z + 1, z^10 + z^9 + z^8 + z^7 + z^6 + z^5 + z^4 + z^3 + z^2 + z + 1)
sage: print(((p*d - n) % m).is_zero())
True
```

Over an integral domain \( d \) might not be monic:

```python
sage: P = PolynomialRing(ZZ, 'x')
sage: x = P.gen()
sage: p = 7*x^5 - 10*x^4 + 16*x^3 - 32*x^2 + 128*x + 256
sage: m = x^5
sage: n, d = p.rational_reconstruct(m, 3, 2)
sage: print((n, d))
(-32*x^3 + 384*x^2 + 2304*x + 2048, 5*x + 8)
sage: print(((p*d - n) % m).is_zero())
True
sage: n, d = p.rational_reconstruct(m, 4, 0)
sage: print((n, d))
(-10*x^4 + 16*x^3 - 32*x^2 + 128*x + 256, 1)
sage: print(((p*d - n) % m).is_zero())
True
```

Over \( \mathbb{Q}(t)[z] \):
sage: P = PolynomialRing(QQ, 't')
sage: t = P.gen()
sage: Pz = PolynomialRing(P.fraction_field(), 'z')
sage: z = Pz.gen()
sage: p = (1 + t^2*z + z^4) / (1 - t*z)

sage: p = (1 + t^2*z + z^4)*(1 - t*z).inverse_mod(z^9)

sage: m = z^9
sage: n, d = p.rational_reconstruct(m)
sage: print((n, d))
(-1/t*z^4 - t*z - 1/t, z - 1/t)

sage: print(((p*d - n) % m).is_zero())
True

sage: w = PowerSeriesRing(P.fraction_field(), 'w').gen()
sage: n = -10^8*t^2*z^4 + (-t^2 + t - 1)*z^3 + (-t - 8)*z^2 + z + 2*t^2 - t
sage: d = z^4 + (2*t + 4)*z^3 + (-t + 5)*z^2 + (t^2 + 2)*z + t^2 + 2*t + 1
sage: prec = 9
sage: nc, dc = Pz((n.subs(z = w)/d.subs(z = w) + O(w^prec)).list()).rational_reconstruct(z^prec)
sage: print((nc, dc) == (n, d))
True

Over \( \mathbb{Q}(t)[z] \):

sage: P = PolynomialRing(QQ, 't')

sage: t = P.gen()

sage: z = PolynomialRing(P, 'z').gen()

sage: p = (1 + t^2*z + z^4) / (1 - t*z) mod z^9

sage: p = (1 + t^2*z + z^4) * sum((t*z)^i for i in range(9))

sage: m = z^9
sage: n, d = p.rational_reconstruct(m)

sage: print((n, d))
(-z^4 - t^2*z - 1, t*z - 1)

sage: print(((p*d - n) % m).is_zero())
True

Over \( \mathbb{Q}_5 \):

sage: x = PolynomialRing(Qp(5), 'x').gen()

sage: p = 4*x^5 + 3*x^4 + 2*x^3 + 2*x^2 + 4*x + 2
sage: m = x^6

sage: n, d = p.rational_reconstruct(m, 3, 2)

sage: print(((p*d - n) % m).is_zero())
True

Can also be used to obtain known Padé approximations:

sage: z = PowerSeriesRing(QQ, 'z').gen()

sage: P = PolynomialRing(QQ, 'x')

sage: x = P.gen()

sage: p = P(exp(z).list())

sage: m = x^5

sage: n, d = p.rational_reconstruct(m, 4, 0)

sage: print((n, d))
(continues on next page)
sage: print(((p*d - n) % m).is_zero())
True
sage: m = x^3
sage: n, d = p.rational_reconstruct(m, 1, 1)
sage: print((n,d))
(-x - 2, x - 2)
sage: print(((p*d - n) % m).is_zero())
True
sage: p = P(log(1-z).list())
sage: m = x^9
sage: n, d = p.rational_reconstruct(m, 4, 4)
sage: print((n,d))
(25/6*x^4 - 130/3*x^3 + 105*x^2 - 70*x, x^4 - 20*x^3 + 90*x^2 - 140*x + 70)
sage: print(((p*d - n) % m).is_zero())
True
sage: p = P(sqrt(1+z).list())
sage: m = x^6
sage: n, d = p.rational_reconstruct(m, 3, 2)
sage: print((n,d))
(1/6*x^3 + 3*x^2 + 8*x + 16/3, x^2 + 16/3*x + 16/3)
sage: print(((p*d - n) % m).is_zero())
True
sage: p = P(exp(2*z).list())
sage: m = x^7
sage: n, d = p.rational_reconstruct(m, 3, 3)
sage: print((n,d))
(-x^3 - 6*x^2 - 15*x - 15, x^3 - 6*x^2 + 15*x - 15)
sage: print(((p*d - n) % m).is_zero())
True

Over \(\mathbb{R}[z]\):

sage: z = PowerSeriesRing(RR, 'z').gen()
sage: P = PolynomialRing(RR,'x')
sage: x = P.gen()
sage: p = P(exp(2*z).list())
sage: m = x^7
sage: n, d = p.rational_reconstruct(m, 3, 3)
sage: print((n,d)) # absolute tolerance 1e-10
(-x^3 - 6.0*x^2 - 15.0*x - 15.0, x^3 - 6.0*x^2 + 15.0*x - 15.0)

See also:
- sage.matrix.berlekamp_massey
- sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint.rational_reconstruct()

real_roots()
Return the real roots of this polynomial, without multiplicities.
Calls self.roots(ring=RR), unless this is a polynomial with floating-point real coefficients, in which case it calls self.roots().
EXAMPLES:

```
sage: x = polygen(ZZ)
sage: (x^2 - x - 1).real_roots()
[-0.618033988749895, 1.61803398874989]
```

**reciprocal_transform**(R=1, q=1)
Transform a general polynomial into a self-reciprocal polynomial.

The input $Q$ and output $P$ satisfy the relation

$$P(x) = Q(x + q/x)x^\deg(Q)R(x).$$

In this relation, $Q$ has all roots in the real interval $[-2\sqrt{q}, 2\sqrt{q}]$ if and only if $P$ has all roots on the circle $|x| = \sqrt{q}$ and $R$ divides $x^2 - q$.

*See also:*
The inverse operation is *trace_polynomial()*.

**INPUT:**

- $R$ – polynomial
- $q$ – scalar (default: 1)

**EXAMPLES:**

```
sage: pol.<x> = PolynomialRing(Rationals())
sage: u = x^2+x-1
sage: u.reciprocal_transform()
x^4 + x^3 + x^2 + x + 1
sage: u.reciprocal_transform(R=x-1)
x^5 - 1
sage: u.reciprocal_transform(q=3)
x^4 + x^3 + 5*x^2 + 3*x + 9
```

**resultant**(other)
Return the resultant of self and other.

**INPUT:**

- other – a polynomial

**OUTPUT:** an element of the base ring of the polynomial ring

**ALGORITHM:**
Uses PARI’s *polresultant* function. For base rings that are not supported by PARI, the resultant is computed as the determinant of the Sylvester matrix.

**EXAMPLES:**

```
sage: R.<x> = QQ[]
sage: f = x^3 + x + 1; g = x^3 - x - 1
sage: r = f.resultant(g); r
-8
sage: r.parent() is QQ
True
```

We can compute resultants over univariate and multivariate polynomial rings.
### reverse

Return polynomial but with the coefficients reversed.

If an optional degree argument is given the coefficient list will be truncated or zero padded as necessary before reversing it. Assuming that the constant coefficient of `self` is nonzero, the reverse polynomial will have the specified degree.

**EXAMPLES:**

```python
sage: R.<x> = ZZ[]; S.<y> = R[]
sage: f = y^3 + x*y -3*x; f
y^3 + x*y - 3*x
sage: f.reverse()
-3*x*y^3 + x*y^2 + 1
sage: f.reverse(degree=2)
-3*x*y^2 + x*y
sage: f.reverse(degree=5)
-3*x*y^5 + x*y^4 + y^2
```

### revert_series

Return a polynomial \( f \) such that \( f(self(x)) = self(f(x)) = x \mod x^n \).

Currently, this is only implemented over some coefficient rings.

**EXAMPLES:**

```python
sage: Pol.<x> = QQ[]
sage: (x + x^3/6 + x^5/120).revert_series(6)
3/40*x^5 - 1/6*x^3 + x
sage: Pol.<x> = CBF[]
sage: (x + x^3/6 + x^5/120).revert_series(6)
([0.075000000000000 +/- ...e-17])*x^5 + (-0.166666666666667 +/- ...e-16)*x^3 + x
sage: Pol.<x> = SR[]
sage: x.revert_series(6)
Traceback (most recent call last):
  ... Not ImplementedError: only implemented for certain base rings
```
\texttt{root_field(names, check_irreducible=True)}

Return the field generated by the roots of the irreducible polynomial self. The output is either a number field, relative number field, a quotient of a polynomial ring over a field, or the fraction field of the base ring.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: R.<x> = QQ['x']
sage: f = x^3 + x + 17
sage: f.root_field('a')
Number Field in a with defining polynomial x^3 + x + 17

sage: R.<x> = QQ['x']
sage: f = x - 3
sage: f.root_field('b')
Rational Field

sage: R.<x> = ZZ['x']
sage: f = x^3 + x + 17
sage: f.root_field('b')
Number Field in b with defining polynomial x^3 + x + 17

sage: y = QQ['x'].0
sage: L.<a> = NumberField(y^3-2)
sage: R.<x> = L['x']
sage: f = x^3 + x + 17
sage: f.root_field('c')
Number Field in c with defining polynomial x^3 + x + 17 over its base field

sage: R.<x> = PolynomialRing(GF(9, 'a'))
sage: f = x^3 + x^2 + 8
sage: K.<alpha> = f.root_field(); K
Univariate Quotient Polynomial Ring in alpha over Finite Field in a of size 3^2
˓
\rightarrow with modulus x^3 + x^2 + 2
sage: alpha^2 + 1
alpha^2 + 1
sage: alpha^3 + alpha^2
1

sage: R.<x> = QQ[]
sage: f = x^2
sage: K.<alpha> = f.root_field()
Traceback (most recent call last):
... ValueError: polynomial must be irreducible
\end{verbatim}
• **algorithm** - the root-finding algorithm to use. We attempt to select a reasonable algorithm by default, but this lets the caller override our choice.

By default, this finds all the roots that lie in the base ring of the polynomial. However, the ring parameter can be used to specify a ring to look for roots in.

If the polynomial and the output ring are both exact (integers, rationals, finite fields, etc.), then the output should always be correct (or raise an exception, if that case is not yet handled).

If the output ring is approximate (floating-point real or complex numbers), then the answer will be estimated numerically, using floating-point arithmetic of at least the precision of the output ring. If the polynomial is ill-conditioned, meaning that a small change in the coefficients of the polynomial will lead to a relatively large change in the location of the roots, this may give poor results. Distinct roots may be returned as multiple roots, multiple roots may be returned as distinct roots, real roots may be lost entirely (because the numerical estimate thinks they are complex roots). Note that polynomials with multiple roots are always ill-conditioned; there’s a footnote at the end of the docstring about this.

If the output ring is a RealIntervalField or ComplexIntervalField of a given precision, then the answer will always be correct (or an exception will be raised, if a case is not implemented). Each root will be contained in one of the returned intervals, and the intervals will be disjoint. (The returned intervals may be of higher precision than the specified output ring.)

At the end of this docstring (after the examples) is a description of all the cases implemented in this function, and the algorithms used. That section also describes the possibilities for “algorithm=”, for the cases where multiple algorithms exist.

**EXAMPLES:**

```python
sage: x = QQ['x'].0
sage: f = x^3 - 1
sage: f.roots()
[(1, 1)]
sage: f.roots(ring=CC)  # note -- low order bits slightly different on ppc.
[(1.00000000000000, 1), (-0.500000000000000 - 0.86602540378443...*I, 1), (-0.500000000000000 + 0.86602540378443...*I, 1)]
sage: f = (x^3 - 1)^2
sage: f.roots()
[(1, 2)]
sage: f = -19*x + 884736
sage: f.roots()
[(884736/19, 1)]
sage: (f^20).roots()
[(884736/19, 20)]
```

```python
sage: K.<z> = CyclotomicField(3)
sage: f = K.defining_polynomial()
sage: f.roots(ring=GF(7))
[(4, 1), (2, 1)]
sage: g = f.change_ring(GF(7))
sage: g.roots()
[(4, 1), (2, 1)]
sage: g.roots(multiplicities=False)
[4, 2]
```

A new ring. In the example below, we add the special method `_roots_univariate_polynomial` to the base ring, and observe that this method is called instead to find roots of polynomials over this ring. This facility
can be used to easily extend root finding to work over new rings you introduce:

```python
sage: R.<x> = QQ[]
sage: (x^2 + 1).roots()
[]
sage: g = lambda f, *args, **kwds: f.change_ring(CDF).roots()
sage: QQ._roots_univariate_polynomial = g
sage: (x^2 + 1).roots()  # abs tol 1e-14
[(2.7755575615628914e-17 - 1.0*I, 1), (0.9999999999999997*I, 1)]
sage: del QQ._roots_univariate_polynomial
```

An example over RR, which illustrates that only the roots in RR are returned:

```python
sage: x = RR['x'].0
sage: f = x^3 - 2
sage: f.roots()
[(1.25992104989487, 1)]
sage: f.factor()
(x - 1.25992104989487)* (x^2 + 1.25992104989487*x + 1.58740105196820)
sage: x = RealField(100)['x'].0
sage: f = x^3 - 2
sage: f.roots()
[(1.2599210498948731647672106073, 1)]
sage: x = CC['x'].0
sage: f = x^3 - 2
sage: f.roots()
[(1.25992104989487, 1), (-0.629960524947437 - 1.09112363597172*I, 1), (-0.629960524947437 + 1.09112363597172*I, 1)]
sage: f.roots(algorithm='pari')
[(1.25992104989487, 1), (-0.629960524947437 - 1.09112363597172*I, 1), (-0.629960524947437 + 1.09112363597172*I, 1)]
```

Another example showing that only roots in the base ring are returned:

```python
sage: x = polygen(ZZ)
sage: f = (2*x-3) * (x-1) * (x+1)
sage: f.roots()
[(1, 1), (-1, 1)]
sage: f.roots(ring=QQ)
[(3/2, 1), (1, 1), (-1, 1)]
```

An example where we compute the roots lying in a subring of the base ring:

```python
sage: Pols.<n> = QQ[]
sage: pol = (n - 1/2)^2*(n - 1)^2*(n-2)
sage: pol.roots(ZZ)
[(2, 1), (1, 2)]
```

An example involving large numbers:

```python
sage: x = RR['x'].0
sage: f = x^2 - 1e100
sage: f.roots()
```

(continues on next page)
\[\begin{align*}
\text{sage: } f &= x^{10} - 2^6(5^5x-1)^2 \\
\text{sage: } f.\text{roots}(\text{multiplicities=False}) \\
&= [-1.6772670339941..., 0.19995479628..., 0.20004530611..., 1.5763035161844...
\end{align*}\]

\[\begin{align*}
\text{sage: } x &= \text{CC['}x'\text{'].0} \\
\text{sage: } i &= \text{CC.}0 \\
\text{sage: } f &= (x - 1)^8(x - i) \\
\text{sage: } f.\text{roots}(\text{multiplicities=False}) \\
&= [1.00000000000000, 1.00000000000000*I]
\end{align*}\]

Describing roots using radical expressions:

\[\begin{align*}
\text{sage: } x &= \text{QQ['}x'\text{'].0} \\
\text{sage: } f &= x^2 + 2 \\
\text{sage: } f.\text{roots}(\text{SR}) \\
&= [(-I*sqrt(2), 1), (I*sqrt(2), 1)]
\end{align*}\]

The roots of some polynomials cannot be described using radical expressions:

\[\begin{align*}
\text{sage: } (x^5 - x + 1).\text{roots}(\text{SR}) \\
&= []
\end{align*}\]

For some other polynomials, no roots can be found at the moment due to the way roots are computed. trac ticket #17516 addresses these defects. Until that gets implemented, one such example is the following:

\[\begin{align*}
\text{sage: } f &= x^6-300*x^5+30361*x^4-1061610*x^3+1141893*x^2-915320*x+101724 \\
\text{sage: } f.\text{roots()} \\
&= []
\end{align*}\]

A purely symbolic roots example:

\[\begin{align*}
\text{sage: } X &= \text{var('X')} \\
\text{sage: } f &= \text{expand((X-1)*(X-I)^3*(X^2 - sqrt(2)))}; f \\
X^6 - (3*I + 1)*X^5 - sqrt(2)*X^4 + (3*I - 3)*X^4 + (3*I + 1)*sqrt(2)*X^3 + (I\_ \text{___}3)*X^3 - (3*I - 3)*sqrt(2)*X^2 - I*X^2 - (I + 3)*sqrt(2)*X + I*sqrt(2) \\
\text{sage: } f.\text{roots()} \\
&= [(I, 3), (-2^(1/4), 1), (2^(1/4), 1), (1, 1)]
\end{align*}\]

The same operation, performed over a polynomial ring with symbolic coefficients:

\[\begin{align*}
\text{sage: } X &= \text{SR['}X'\text{'].0} \\
\text{sage: } f &= (X-1)*(X-I)^3*(X^2 - sqrt(2)); f \\
X^6 - (3*I - 1)*X^5 + (-sqrt(2)*X + 3*I - 3)*X^4 + (3*I + 1)*sqrt(2)*X^3 + (I\_ \text{___}3)*X^3 + (-3*I - 3)*sqrt(2) - I)*X^2 + (-I + 3)*sqrt(2)*X + I*sqrt(2) \\
\text{sage: } f.\text{roots()} \\
&= [(I, 3), (-2^(1/4), 1), (2^(1/4), 1), (1, 1)]
\end{align*}\]
A couple of examples where the base ring does not have a factorization algorithm (yet). Note that this is currently done via a rather naive enumeration, so could be very slow:

```
sage: R = Integers(6)
sage: S.<x> = R['x']
sage: p = x^2-1
sage: p.roots()
Traceback (most recent call last):
...  
NotImplementedError: root finding with multiplicities for this polynomial not implemented (try the multiplicities=False option)
sage: p.roots(multiplicities=False)
[5, 1]
sage: R = Integers(9)
sage: A = PolynomialRing(R, 'y')
sage: y = A.gen()
sage: f = 10*y^2 - y^3 - 9
sage: f.roots(multiplicities=False)
[1, 3, 6]
```

An example over the complex double field (where root finding is fast, thanks to NumPy):

```
sage: R.<x> = CDF[]
sage: f = R.cyclotomic_polynomial(5); f
x^4 + x^3 + x^2 + x + 1.0
sage: f.roots(multiplicities=False)
# abs tol 1e-9
[-0.8090169943749469 - 0.5877852522924724*I, -0.8090169943749473 + 0.*I, 0.30901699437494773 - 0.951056516295154*I, 0.30901699437494756 + 0.9510565162951525*I]
sage: [z^5 for z in f.roots(multiplicities=False)]
# abs tol 2e-14
[0.9999999999999957 - 1.2864981197413038e-15*I, 0.9999999999999976 + 3.06285495141552e-15*I, 1.0000000000000024 + 1.1331077795295987e-15*I, 0.9999999999999953 - 2.0212861992297117e-15*I]
sage: f = CDF['x']([1,2,3,4]); f
4.0*x^3 + 3.0*x^2 + 2.0*x + 1.0
sage: r = f.roots(multiplicities=False)
sage: [f(a).abs() for a in r]
# abs tol 1e-14
```

Another example over RDF:

```
sage: x = RDF['x']

sage: ((x^3 -1)).roots()  
# abs tol 4e-16
[(1.0000000000000002, 1)]
sage: ((x^3 -1)).roots(multiplicities=False)  
# abs tol 4e-16
[1.0000000000000002]
```

More examples involving the complex double field:
sage: x = CDF['x'].0
sage: i = CDF.0
sage: f = x^3 + 2*i; f
x^3 + 2.0*I
sage: f.roots()
[(-1.09112363597172... - 0.62996052494743...*I, 1), (...1.25992104989487...*I, 
 1), (1.09112363597172... - 0.62996052494743...*I, 1)]
sage: f.roots(multiplicities=False)
[-1.09112363597172... - 0.62996052494743...*I, ...1.25992104989487...*I, 1.
 1.09112363597172... - 0.62996052494743...*I]
sage: abs(f(z)) for z in f.roots(multiplicities=False)
# abs tol 1e-14
[8.95090418262362e-16, 8.728374398092689e-16, 1.0235750533041806e-15]
sage: f = i*x^3 + 2; f
I*x^3 + 2.0
sage: f.roots()
[(-1.09112363597172... + 0.62996052494743...*I, 1), (...1.25992104989487...*I, 
 1), (1.09112363597172... + 0.62996052494743...*I, 1)]
sage: abs(f(f.roots()[0][0]))
# abs tol 1e-13
1.1102230246251565e-16

Examples using real root isolation:

sage: x = polygen(ZZ)
sage: f = x^2 - x - 1
sage: f.roots()
[]
sage: f.roots(ring=RIF)
[(-0.6180339887498948482045868343657?, 1), (1.6180339887498948482045868343657?,
 1)]
sage: f.roots(ring=RIF, multiplicities=False)
[-0.6180339887498948482045868343657?, 1.6180339887498948482045868343657?]
sage: f.roots(ring=RealIntervalField(150))
[(-0.6180339887498948482045868343657381177203091798057628621354486227?, 1), (1.
 6180339887498948482045868343657381177203091798057628621354486227?, 1)]
sage: f.roots(ring=AA)
[(-0.618033988749895?, 1), (1.618033988749895?, 1)]
sage: f = f^2 * (x - 1)
sage: f.roots(ring=RIF)
[(-0.6180339887498948482045868343657?, 2), (1.0000000000000000000000000000000?,
 1), (1.6180339887498948482045868343657?, 1)]
sage: f.roots(ring=RIF, multiplicities=False)
[-0.6180339887498948482045868343657?, 1.6180339887498948482045868343657?]

Examples using complex root isolation:

sage: x = polygen(ZZ)
sage: p = x^5 - x - 1
sage: p.roots()
[]
sage: p.roots(ring=CIF)
[(-0.764884433600585? - 0.352471546031727?*I, 1), (-0.764884433600585? + 0.352471546031727?*I, 1), (0.181232444469876? - 1.
 0.181232444469876?*I, 1), (0.181232444469876? + 1.083954101317711?*I, 1)]
sage: p.roots(ring=ComplexIntervalField(200))
[(1.16730397826141868425604589985489039140082?, 1), (-0.
 ˓→764884433605847260298231877085417303289965194736756700778 - 0.
 ˓→352471546031726493197407914205810543942064802824733283770*I, 1), (-0.
 ˓→764884433605847260298231877085417303289965194736756700778 + 0.
 ˓→352471546031726493197407914205810543942064802824733283770*I, 1), (0.
 ˓→1812344446987389390180023778112063996871646618642304743774 - 1.
 ˓→08395410131771668430344929807667574273640231511565430114*I, 1), (0.
 ˓→1812344446987389390180023778112063996871646618642304743774 + 1.
 ˓→08395410131771668430344929807667574273640231511565430114*I, 1)]

sage: rts = p.roots(ring=QQbar); rts
[(1.167303978261419?, 1), (-0.7648844336005847? - 0.3524715460317263?*I, 1), (-
 ˓→0.7648844336005847? + 0.3524715460317263?*I, 1), (0.1812324444698754? - 1.
 ˓→0839541013177117?*I, 1), (0.1812324444698754? + 1.0839541013177117?*I, 1)]

sage: p.roots(ring=AA)
[(1.167303978261419?, 1)]

sage: p = (x - rts[4][0])^2 * (3*x^2 + x + 1)

sage: p.roots(ring=QQbar)
[(-0.1666666666666667? - 0.552770798392567?*I, 1), (-0.1666666666666667? + 0.
 ˓→552770798392567?*I, 1), (0.1812324444698754? + 1.0839541013177117?*I, 2)]

sage: p.roots(ring=CIF)
[(-0.1666666666666667? - 0.552770798392567?*I, 1), (-0.1666666666666667? + 0.
 ˓→552770798392567?*I, 1), (0.1812324444698754? + 1.0839541013177117?*I, 2)]

In some cases, it is possible to isolate the roots of polynomials over complex ball fields:

sage: Pol.<x> = CBF[]
sage: (x^2 + 2).roots(multiplicities=False)
[[+/- e-19] + [+/- e-19]*I,
 [+/- e-19] + [+/- e-19]*I]
sage: (x^3 - 1/2).roots(RBF, multiplicities=False)
[[0.7937005259840997 +/- e-17]]
sage: ((x - 1)^2).roots(multiplicities=False, proof=False)
doctest:...
UserWarning: roots may have been lost...
[[1.0000000000 +/- e-12] + [+/- e-12]*I,
 [1.0000000000 +/- e-12]*I]

Note that coefficients in a number field with defining polynomial $x^2 + 1$ are considered to be Gaussian rationals (with the generator mapping to $+I$), if you ask for complex roots.

sage: K.<im> = QuadraticField(-1)
sage: y = polygen(K)
sage: p = y^4 - 2 - im

sage: p.roots(ring=CC)
[(-1.2146389322441... - 0.14142505258239...*I, 1), (-0.14142505258239... + 1.
 ˓→2146389322441...*I, 1), (0.14142505258239... - 1.2146389322441...*I, 1), (1.
 ˓→2146389322441... + 0.14142505258239...*I, 1)]

sage: p = p^2 * (y^2 - 2)

sage: p.roots(ring=CIF)
[(-1.414213562373095?, 1), (1.414213562373095?, 1), (-1.214638932244183? - 0.
 ˓→141425052582394?*I, 2), (-0.141425052582394? + 1.214638932244183?*I, 2), (0.
 ˓→141425052582394? - 1.214638932244183?*I, 2), (1.214638932244183? + 0.
 ˓→141425052582394?*I, 2)]
Note that one should not use NumPy when wanting high precision output as it does not support any of the high precision types:

```sage
sage: R.<x> = RealField(200)
```

```sage
sage: f = x^2 - R(pi)
```

```sage
sage: f.roots()
```

```
[(-1.77245385090551..., 1), (1.77245385090551..., 1)]
```

```sage
sage: f.roots(algorithm='numpy')
doctest... UserWarning: NumPy does not support arbitrary precision arithmetic.
```

```
The roots found will likely have less precision than you expect.
```

```
[(-1.77245385090551..., 1), (1.77245385090551..., 1)]
```

We can also find roots over number fields:

```sage
sage: K.<z> = CyclotomicField(15)
sage: R.<x> = PolynomialRing(K)
sage: (x^2 + x + 1).roots()
```

```
[(z^5, 1), (-z^5 - 1, 1)]
```

There are many combinations of floating-point input and output types that work. (Note that some of them are quite pointless like using `algorithm='numpy'` with high-precision types.)

```sage
sage: rflds = (RR, RDF, RealField(100))
sage: cflds = (CC, CDF, ComplexField(100))
sage: def cross(a, b):
    .....:     return list(cartesian_product_iterator([a, b]))
sage: flds = cross(rflds, rflds) + cross(rflds, cflds) + cross(cflds, cflds)
sage: for (fld_in, fld_out) in flds:
    .....:     x = polygen(fld_in)
    .....:     f = x^3 - fld_in(2)
    .....:     x2 = polygen(fld_out)
    .....:     f2 = x2^3 - fld_out(2)
    .....:     for algo in (None, 'pari', 'numpy'):
    .....:         if fld_in == fld_out and algo is None:
    .....:             print('{} {}'.format(fld_in, algo))
    .....:     for rt in f.roots(ring=fld_out, multiplicities=False):
    .....:         assert(abs(f2(rt)) <= 1e-10)
```

```
Real Field with 53 bits of precision [1.259921049894874...]
Real Double Field [1.259921049894874...]
Complex Field with 53 bits of precision [1.259921049894874... - 1.09112363597172*I, -0.62996052494743... + 1.09112363597172*I]
Complex Double Field [1.259921049894874... - 1.09112363597172*I, -0.62996052494743... + 1.09112363597172*I]
Complex Field with 100 bits of precision [1.2599210498948731647672106073, -0.62996052494743... + 1.09112363597172*I]
```

Note that we can find the roots of a polynomial with algebraic coefficients:
We can handle polynomials with huge coefficients.

This number doesn’t even fit in an IEEE double-precision float, but RR and CC allow a much larger range of floating-point numbers:

```
sage: bigc = 2^1500
sage: CDF(bigc)
+infinity
sage: CC(bigc)
3.50746621104340e451
```

Polynomials using such large coefficients can’t be handled by numpy, but pari can deal with them:

```
sage: p = x + bigc
sage: p.roots(ring=RR, algorithm='numpy')
Traceback (most recent call last):
  ... LinAlgError: Array must not contain infs or NaNs
sage: p.roots(ring=RR, algorithm='pari')
[(-2.85106096489671e-452, 1)]
sage: p.roots(ring=AA)
[(-2.8510609648967059?e-452, 1)]
sage: p.roots(ring=QQbar)
[(-2.8510609648967059?e-452, 1)]
sage: p = bigc*x + 1
sage: p.roots(ring=RR)
[(-2.85106096489671e-452, 1)]
sage: p.roots(ring=AA)
[(-2.8510609648967059?e-452, 1)]
sage: p.roots(ring=QQbar)
[(-2.8510609648967059?e-452, 1)]
sage: p = x^2 - bigc
sage: p.roots(ring=RR)
[(-5.922386521532868e225, 1), (5.922386521532868e225, 1)]
sage: p.roots(ring=QQbar)
[(-5.9223865215328558?e225, 1), (5.9223865215328558?e225, 1)]
```

Check that trac ticket #30522 is fixed:
Check that trac ticket #30523 is fixed:

```python
sage: PolynomialRing(SR, names="x")("x^2 + q").roots()
[(-sqrt(-q), 1), (sqrt(-q), 1)]
```

Algorithms used:

For brevity, we will use RR to mean any RealField of any precision; similarly for RIF, CC, and CIF. Since Sage has no specific implementation of Gaussian rationals (or of number fields with embedding, at all), when we refer to Gaussian rationals below we will accept any number field with defining polynomial \( x^2 + 1 \), mapping the field generator to +I.

We call the base ring of the polynomial K, and the ring given by the ring= argument L. (If ring= is not specified, then L is the same as K.)

If K and L are floating-point (RDF, CDF, RR, or CC), then a floating-point root-finder is used. If L is RDF or CDF then we default to using NumPy’s roots(); otherwise, we use PARI’s polroots(). This choice can be overridden with algorithm='pari' or algorithm='numpy'. If the algorithm is unspecified and NumPy’s roots() algorithm fails, then we fall back to pari (numpy will fail if some coefficient is infinite, for instance).

If L is SR, then the roots will be radical expressions, computed as the solutions of a symbolic polynomial expression. At the moment this delegates to `sage.symbolic.expression.Expression.solve()` which in turn uses Maxima to find radical solutions. Some solutions may be lost in this approach. Once trac ticket #17516 gets implemented, all possible radical solutions should become available.

If L is AA or RIF, and K is ZZ, QQ, or AA, then the root isolation algorithm `sage.rings.polynomial.real_roots.real_roots()` is used. (You can call real_roots() directly to get more control than this method gives.)

If L is QQbar or CIF, and K is ZZ, QQ, AA, QQbar, or the Gaussian rationals, then the root isolation algorithm `sage.rings.polynomial.complex_roots.complex_roots()` is used. (You can call complex_roots() directly to get more control than this method gives.)

If L is AA and K is QQbar or the Gaussian rationals, then complex_roots() is used (as above) to find roots in QQbar, then these roots are filtered to select only the real roots.

If L is floating-point and K is not, then we attempt to change the polynomial ring to L (using `.change_ring()`) (or, if L is complex and K is not, to the corresponding real field). Then we use either PARI or numpy as specified above.

For all other cases where K is different than L, we attempt to use `.change_ring(L)`. When that fails but L is a subring of K, we also attempt to compute the roots over K and filter the ones belonging to L.

The next method, which is used if K is an integral domain, is to attempt to factor the polynomial. If this succeeds, then for every degree-one factor a\(x+b\), we add -b/a as a root (as long as this quotient is actually in the desired ring).

If factoring over K is not implemented (or K is not an integral domain), and K is finite, then we find the roots by enumerating all elements of K and checking whether the polynomial evaluates to zero at that value.

**Note:** We mentioned above that polynomials with multiple roots are always ill-conditioned; if your input is given to n bits of precision, you should not expect more than n/k good bits for a k-fold root. (You can get solutions that make the polynomial evaluate to a number very close to zero; basically the problem is that with a multiple root, there are many such numbers, and it’s difficult to choose between them.)
To see why this is true, consider the naive floating-point error analysis model where you just pretend that all floating-point numbers are somewhat imprecise - a little ‘fuzzy’, if you will. Then the graph of a floating-point polynomial will be a fuzzy line. Consider the graph of \((x - 1)^3\); this will be a fuzzy line with a horizontal tangent at \(x = 1, y = 0\). If the fuzziness extends up and down by about \(j\), then it will extend left and right by about \(\text{cube_root}(j)\).

**shift**\((n)\)

Return this polynomial multiplied by the power \(x^n\). If \(n\) is negative, terms below \(x^n\) will be discarded. Does not change this polynomial (since polynomials are immutable).

**EXAMPLES:**

```sage
sage: R.<x> = QQ[]
sage: p = x^2 + 2*x + 4
sage: p.shift(0)
x^2 + 2*x + 4
sage: p.shift(-1)
x + 2
sage: p.shift(-5)
0
sage: p.shift(2)
x^4 + 2*x^3 + 4*x^2
```

One can also use the infix shift operator:

```sage
sage: f = x^3 + x
sage: f >> 2
x
sage: f << 2
x^5 + x^3
```

**AUTHORS:**

- David Harvey (2006-08-06)

**specialization**\((D=None, phi=None)\)

Specialization of this polynomial.

Given a family of polynomials defined over a polynomial ring. A specialization is a particular member of that family. The specialization can be specified either by a dictionary or a SpecializationMorphism.

**INPUT:**

- \(D\) – dictionary (optional)
- \(phi\) – SpecializationMorphism (optional)

**OUTPUT:** a new polynomial

**EXAMPLES:**

```sage
sage: R.<c> = PolynomialRing(ZZ)
sage: S.<z> = PolynomialRing(R)
sage: F = c*z^2 + c^2
sage: F.specialization({c:2})
2*z^2 + 4
```
sage: A.<c> = QQ[]
sage: R.<x> = Frac(A)[]
sage: X = (1 + x/c).specialization({c:20})
sage: X
1/20*x + 1
sage: X.parent()
Univariate Polynomial Ring in x over Rational Field

\textbf{splitting_field}(names=\texttt{None}, map=\texttt{False}, **\texttt{kwds})

Compute the absolute splitting field of a given polynomial.

\textbf{INPUT}:

\begin{itemize}
  \item names – (default: \texttt{None}) a variable name for the splitting field.
  \item map – (default: \texttt{False}) also return an embedding of \texttt{self} into the resulting field.
  \item kwds – additional keywords depending on the type. Currently, only number fields are implemented.
\end{itemize}

\textbf{OUTPUT}:

If \texttt{map} is \texttt{False}, the splitting field as an absolute field. If \texttt{map} is \texttt{True}, a tuple \texttt{(K, phi)} where \texttt{phi} is an embedding of the base field of \texttt{self} in \texttt{K}.

\textbf{EXAMPLES}:

\begin{verbatim}
sage: R.<x> = PolynomialRing(ZZ)
sage: K.<a> = (x^3 + 2).splitting_field(); K
Number Field in a with defining polynomial x^6 + 3*x^5 + 6*x^4 + 11*x^3 + 12*x^2 - 3*x + 1
sage: K.<a> = (x^3 - 3*x + 1).splitting_field(); K
Number Field in a with defining polynomial x^3 - 3*x + 1
\end{verbatim}

Relative situation:

\begin{verbatim}
sage: R.<x> = PolynomialRing(QQ)
sage: K.<a> = NumberField(x^3 + 2)
sage: S.<t> = PolynomialRing(K)
sage: L.<b> = (t^2 - a).splitting_field()
sage: L
Number Field in b with defining polynomial t^6 + 2
\end{verbatim}

With \texttt{map=\texttt{True}}, we also get the embedding of the base field into the splitting field:

\begin{verbatim}
sage: L.<b>, phi = (t^2 - a).splitting_field(map=True)
sage: phi
Ring morphism:
  From: Number Field in a with defining polynomial x^3 + 2
  To: Number Field in b with defining polynomial t^6 + 2
  Defn: a |--> b^2
\end{verbatim}

An example over a finite field:

\begin{verbatim}
sage: P.<x> = PolynomialRing(GF(7))
sage: t = x^2 + 1
\end{verbatim}
sage: t.splitting_field('b')
Finite Field in b of size 7^2

sage: P.<x> = PolynomialRing(GF(7^3, 'a'))
sage: t = x^2 + 1
sage: t.splitting_field('b', map=True)
(Finite Field in b of size 7^6,
 Ring morphism:
  From: Finite Field in a of size 7^3
  To:   Finite Field in b of size 7^6
  Defn: a |--> 2*b^4 + 6*b^3 + 2*b^2 + 3*b + 2)

If the extension is trivial and the generators have the same name, the map will be the identity:

sage: t = 24*x^13 + 2*x^12 + 14
sage: t.splitting_field('a', map=True)
(Finite Field in a of size 7^3,
 Identity endomorphism of Finite Field in a of size 7^3)

sage: t = x^56 - 14*x^3
sage: t.splitting_field('b', map=True)
(Finite Field in b of size 7^3,
 Ring morphism:
  From: Finite Field in a of size 7^3
  To:   Finite Field in b of size 7^3
  Defn: a |--> b)

See also:

sage.rings.number_field.splitting_field.splitting_field() for more examples over number fields

square()
Return the square of this polynomial.

Todo:

• This is just a placeholder; for now it just uses ordinary multiplication. But generally speaking, squaring is faster than ordinary multiplication, and it’s frequently used, so subclasses may choose to provide a specialised squaring routine.

• Perhaps this even belongs at a lower level? RingElement or something?

AUTHORS:

• David Harvey (2006-09-09)

EXAMPLES:

sage: R.<x> = QQ[]
sage: f = x^3 + 1
sage: f.square()
x^6 + 2*x^3 + 1
sage: f*f
x^6 + 2*x^3 + 1
squarefree_decomposition()
Return the square-free decomposition of this polynomial. This is a partial factorization into square-free, coprime polynomials.

EXAMPLES:

```python
sage: x = polygen(QQ)
sage: p = 37 * (x-1)^3 * (x-2)^3 * (x-1/3)^7 * (x-3/7)
sage: p.squarefree_decomposition()
(37*x - 111/7) * (x^2 - 3*x + 2)^3 * (x - 1/3)^7
sage: p = 37 * (x-2/3)^2
sage: p.squarefree_decomposition()
(37) * (x - 2/3)^2
sage: x = polygen(GF(3))
sage: x.squarefree_decomposition()
x
sage: f = QQbar['x'](1)
sage: f.squarefree_decomposition()
1
```

subresultants(other)
Return the nonzero subresultant polynomials of self and other.

INPUT:

• other – a polynomial

OUTPUT: a list of polynomials in the same ring as self

EXAMPLES:

```python
sage: R.<x> = ZZ[]
sage: f = x^8 + x^6 -3*x^4 -3*x^3 +8*x^2 +2*x -5
sage: g = 3*x^6 +5*x^4 -4*x^2 -9*x +21
sage: f.subresultants(g)
[260708,
 9326*x - 12300,
169*x^2 + 325*x - 637,
65*x^2 + 125*x - 245,
25*x^4 - 5*x^2 + 15,
15*x^4 - 3*x^2 + 9]
```

ALGORITHM:
We use the schoolbook algorithm with Lazard’s optimization described in [Duc1998]

REFERENCES:
Wikipedia article Polynomial_greatest_common_divisor#Subresultants

subs(*x, **kwds)
Identical to self(*x).

See the docstring for self.__call__.

EXAMPLES:

```python
sage: R.<x> = QQ[]
sage: f = x^3 + x - 3
```
```python
sage: f.subs(x=5)
127
sage: f.subs(5)
127
sage: f.subs({x:2})
7
sage: f.subs({})
x^3 + x - 3
sage: f.subs({'x':2})
Traceback (most recent call last):
  ...TypeError: keys do not match self's parent
```

**`substitute(*x, **kwds)`**

Identical to `self(*x)`.

See the docstring for `self.__call__`.

**EXAMPLES:**

```python
sage: R.<x> = QQ[]
sage: f = x^3 + x - 3
sage: f.subs(x=5)
127
sage: f.subs(5)
127
sage: f.subs({x:2})
7
sage: f.subs({})
x^3 + x - 3
sage: f.subs({'x':2})
Traceback (most recent call last):
  ...TypeError: keys do not match self's parent
```

**`sylvester_matrix(right, variable=None)`**

Return the Sylvester matrix of `self` and `right`.

Note that the Sylvester matrix is not defined if one of the polynomials is zero.

**INPUT:**

- `right`: a polynomial in the same ring as `self`.
- `variable`: optional, included for compatibility with the multivariate case only. The variable of the polynomials.

**EXAMPLES:**

```python
sage: R.<x> = PolynomialRing(ZZ)
sage: f = (6*x + 47)*(7*x^2 - 2*x + 38)
sage: g = (6*x + 47)*(3*x^3 + 2*x + 1)
sage: M = f.sylvester_matrix(g)
sage: M
[ 42 317 134 1786 0 0 0]
[ 0 42 317 134 1786 0 0]
```

(continues on next page)
If the polynomials share a non-constant common factor then the determinant of the Sylvester matrix will be zero:

```
sage: M.determinant()
0
```

If self and right are polynomials of positive degree, the determinant of the Sylvester matrix is the resultant of the polynomials:

```
sage: h1 = R._random_nonzero_element()
sage: h2 = R._random_nonzero_element()
sage: M1 = h1.sylvester_matrix(h2)
sage: M1.determinant() == h1.resultant(h2)
True
```

The rank of the Sylvester matrix is related to the degree of the gcd of self and right:

```
sage: f.gcd(g).degree() == f.degree() + g.degree() - M.rank()
True
sage: h1.gcd(h2).degree() == h1.degree() + h2.degree() - M1.rank()
True
```

**symmetric_power***(k, monic=False)**

Return the polynomial whose roots are products of k-th distinct roots of this.

**EXAMPLES:**

```
sage: x = polygen(QQ)
sage: f = x^4-x+2
sage: [f.symmetric_power(k) for k in range(5)]
[x - 1, x^4 - x + 2, x^6 - 2*x^4 - x^3 - 4*x^2 + 8, x^4 - x^3 + 8, x - 2]
sage: f = x^5-2*x+2
sage: [f.symmetric_power(k) for k in range(6)]
[x - 1, x^5 - 2*x + 2, x^10 + 2*x^8 - 4*x^6 - 8*x^5 - 8*x^4 - 8*x^3 + 16, x^10 + 4*x^7 - 8*x^6 + 16*x^5 - 16*x^4 + 32*x^2 + 64, x^5 + 2*x^4 - 16, x + 2]
sage: R.<a,b,c,d> = ZZ[]
sage: x = polygen(R)
sage: f = (x-a)*(x-b)*(x-c)*(x-d)
sage: [f.symmetric_power(k).factor() for k in range(5)]
[x - 1, (-x + d) * (-x + c) * (-x + b) * (-x + a),
```

(continues on next page)
(continued from previous page)

\[
(x - c*d) * (x - b*d) * (x - a*d) * (x - b*c) * (x - a*c) * (x - a*b),
(x - b*c*d) * (x - a*c*d) * (x - a*b*d) * (x - a*b*c),
x - a*b*c*d
\]

\textbf{trace\_polynomial()}

Compute the trace polynomial and cofactor.

The input \(P\) and output \(Q\) satisfy the relation

\[
P(x) = Q(x + q/x)x^{\deg(Q)} R(x).
\]

In this relation, \(Q\) has all roots in the real interval \([-2\sqrt{q}, 2\sqrt{q}]\) if and only if \(P\) has all roots on the circle \(|x| = \sqrt{q}\) and \(R\) divides \(x^2 - q\). We thus require that the base ring of this polynomial have a coercion to the real numbers.

\textbf{See also:}

The inverse operation is \texttt{reciprocal\_transform()}.  

\textbf{OUTPUT:}

\begin{itemize}
  \item \texttt{Q} – trace polynomial
  \item \texttt{R} – cofactor
  \item \texttt{q} – scaling factor
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
sage: pol.<x> = PolynomialRing(Rationals())
sage: u = x^5 - 1; u.trace_polynomial()
(x^2 + x - 1, x - 1, 1)
sage: u = x^4 + x^3 + 5*x^2 + 3*x + 9
sage: u.trace_polynomial()
(x^2 + x - 1, 1, 3)
sage: u = x^4 + a*x^3 + 3*x^2 + 2*a*x + 4
sage: u.trace_polynomial()
(x^2 + a*x - 1, 1, 2)
sage: (u*(x^2-2)).trace_polynomial()
(x^2 + a*x - 1, x^2 - 2, 2)
sage: (u*(x^2-2)^2).trace_polynomial()
(x^2 + a*x - 1, x^2 - 2, 2)
sage: (u*(x^2-2)^3).trace_polynomial()
(x^2 + a*x - 1, x^2 - 2, 2)
sage: u = x^4 + a*x^3 + 3*x^2 + 4*a*x + 16
sage: u.trace_polynomial()
(x^2 + a*x - 5, 1, 4)
sage: (u*(x-2)).trace_polynomial()
(x^2 + a*x - 5, x - 2, 4)
sage: (u*(x+2)).trace_polynomial()
(x^2 + a*x - 5, x + 2, 4)
\end{verbatim}

\textbf{truncate(n)}

Return the polynomial of degree \(\leq n\) which is equivalent to self modulo \(x^n\).
EXAMPLES:

```python
sage: R.<x> = ZZ[]; S.<y> = PolynomialRing(R, sparse=True)
sage: f = y^3 + x*y - 3*x; f
y^3 + x*y - 3*x
sage: f.truncate(2)
x*y - 3*x
sage: f.truncate(1)
-3*x
sage: f.truncate(0)
0
```

valuation($p=None$)

If $f = a_r x^r + a_{r+1} x^{r+1} + \cdots$, with $a_r$ nonzero, then the valuation of $f$ is $r$. The valuation of the zero polynomial is $\infty$.

If a prime (or non-prime) $p$ is given, then the valuation is the largest power of $p$ which divides self.

The valuation at $\infty$ is -self.degree().

EXAMPLES:

```python
sage: P.<x> = ZZ[]
sage: (x^2+x).valuation() 1
sage: (x^2+x).valuation(x+1) 1
sage: (x^2+1).valuation() 0
sage: (x^3+1).valuation(infinity) -3
sage: P(0).valuation() +Infinity
```

variable_name()

Return name of variable used in this polynomial as a string.

OUTPUT: string

EXAMPLES:

```python
sage: R.<t> = QQ[]
sage: f = t^3 + 3/2*t + 5
sage: f.variable_name() 't'
```

variables()

Return the tuple of variables occurring in this polynomial.

EXAMPLES:

```python
sage: R.<x> = QQ[]
sage: x.variables() (x,)
```

A constant polynomial has no variables.
\texttt{sage: R(2).variables()}

\texttt{()}

\texttt{\texttt{xgcd}(\texttt{other})}

Return an extended gcd of this polynomial and \texttt{other}.

INPUT:

\begin{itemize}
\item \texttt{other} – a polynomial in the same ring as this polynomial
\end{itemize}

OUTPUT:

A tuple \((r, s, t)\) where \(r\) is a greatest common divisor of this polynomial and \texttt{other}, and \(s\) and \(t\) are such that \(r = s \times \texttt{self} + t \times \texttt{other}\) holds.

\textbf{Note:} The actual algorithm for computing the extended gcd depends on the base ring underlying the polynomial ring. If the base ring defines a method \texttt{\_xgcd\_univariate\_polynomial}, then this method will be called (see examples below).

\textbf{EXAMPLES:}

\begin{verbatim}
sage: R.<x> = QQbar[]
sage: (2*x^2).gcd(2*x)
x
sage: R.zero().gcd(0)
0
sage: (2*x).gcd(0)
x
\end{verbatim}

One can easily add \texttt{xgcd} functionality to new rings by providing a method \texttt{\_xgcd\_univariate\_polynomial}:

\begin{verbatim}
sage: R.<x> = QQ[]
sage: S.<y> = R[]
sage: h1 = y*x
sage: h2 = y^2*x^2
sage: h1.xgcd(h2)
Traceback (most recent call last):
  ...  
NotImplementedError: Univariate Polynomial Ring in x over Rational Field does not provide an xgcd implementation for univariate polynomials
sage: T.<x,y> = QQ[]
sage: def poor_xgcd(f,g):
    ...  ret = S(T(f).gcd(g))
    ...  if ret == f: return ret,S.one(),S.zero()
    ...  if ret == g: return ret,S.zero(),S.one()
    ...  raise NotImplementedError
sage: R._xgcd_univariate Polynomial = poor_xgcd
sage: h1.xgcd(h2)
(x*y, 1, 0)
sage: del R._xgcd_univariate Polynomial
\end{verbatim}

class sage.rings.polynomial.polynomial_element.PolynomialBaseringInjection
Bases: sage.categories.morphism.Morphism

2.1. Univariate Polynomials and Polynomial Rings
This class is used for conversion from a ring to a polynomial over that ring.

It calls the \_new\_constant\_poly method on the generator, which should be optimized for a particular polynomial type.

Technically, it should be a method of the polynomial ring, but few polynomial rings are cython classes, and so, as a method of a cython polynomial class, it is faster.

**EXAMPLES:**

We demonstrate that most polynomial ring classes use polynomial base injection maps for coercion. They are supposed to be the fastest maps for that purpose. See trac ticket #9944.

```
sage: R.<x> = Qp(3)[]
sage: R.coerce_map_from(R.base_ring())
Polynomial base injection morphism:
  From: 3-adic Field with capped relative precision 20
  To:  Univariate Polynomial Ring in x over 3-adic Field with capped relative...
    \--precision 20
sage: R.<x,y> = Qp(3)[]
sage: R.coerce_map_from(R.base_ring())
Polynomial base injection morphism:
  From: 3-adic Field with capped relative precision 20
  To:  Multivariate Polynomial Ring in x, y over 3-adic Field with capped relative...
    \--precision 20
sage: R.<x,y> = QQ[

```

By trac ticket #9944, there are now only very few exceptions:

```
sage: PolynomialRing(QQ,names=[]).coerce_map_from(QQ)
Call morphism:
  From: Rational Field
  To:  Multivariate Polynomial Ring in no variables over Rational Field
sage: R.<x> = QQ[

```

**is\_injective()**

Return whether this morphism is injective.

**EXAMPLES:**

```
sage: R.<x> = ZZ[
sage: S.<y> = R[
sage: S.coerce_map_from(R).is_injective()
True

```

Check that trac ticket #23203 has been resolved:

```
sage: R.is_subring(S) # indirect doctest
True
```
**is_surjective()**
Return whether this morphism is surjective.

EXAMPLES:
```
sage: R.<x> = ZZ
sage: R.coerce_map_from(ZZ).is_surjective()
False
```

**section()**

**class** `sage.rings.polynomial.polynomial_element.Polynomial_generic_dense`
Bases: `sage.rings.polynomial.polynomial_element.Polynomial`
A generic dense polynomial.

EXAMPLES:
```
sage: f = QQ['x','y'].random_element()
sage: loads(f.dumps()) == f
True
```

**constant_coefficient()**
Return the constant coefficient of this polynomial.

**OUTPUT:** element of base ring

EXAMPLES:
```
sage: R.<t> = QQ
sage: S.<x> = R
sage: f = x*t + x + t
sage: f.constant_coefficient()
t
```

**degree(gen=None)**

EXAMPLES:
```
sage: R.<x> = RDF
sage: f = (1+2*x^7)^5
sage: f.degree()
35
```

**is_term()**
Return True if this polynomial is a nonzero element of the base ring times a power of the variable.

EXAMPLES:
```
sage: R.<x> = SR
sage: R(0).is_term()
False
sage: R(1).is_term()
True
sage: (3*x^5).is_term()
True
sage: (1+3*x^5).is_term()
False
```
**list**(*copy=True*)

Return a new copy of the list of the underlying elements of self.

**EXAMPLES:**

```python
sage: R.<x> = GF(17)[]
sage: f = (1+2*x)^3 + 3*x; f
8*x^3 + 12*x^2 + 9*x + 1
sage: f.list()
[1, 9, 12, 8]
```

**quo_rem**(other)

Return the quotient and remainder of the Euclidean division of self and other.

Raises a **ZeroDivisionError** if other is zero. Raises an **ArithmeticError** if the division is not exact.

**AUTHORS:**

• Kwankyu Lee (2013-06-02)
• Bruno Grenet (2014-07-13)

**EXAMPLES:**

```python
sage: P.<x> = QQ[]
sage: R.<y> = P[]
sage: f = R.random_element(10)
sage: g = y^5+R.random_element(4)
sage: q,r = f.quo_rem(g)
sage: f == q*g + r
True
sage: g = x*y^5
sage: f.quo_rem(g)
Traceback (most recent call last):
  ... ArithmeticError: division non exact (consider coercing to polynomials over the fraction field)
sage: g = 0
sage: f.quo_rem(g)
Traceback (most recent call last):
  ... ZeroDivisionError: division by zero polynomial
```

**shift**(*n*)

Return this polynomial multiplied by the power \(x^n\).

If \(n\) is negative, terms below \(x^n\) will be discarded. Does not change this polynomial.

**EXAMPLES:**

```python
sage: R.<x> = PolynomialRing(PolynomialRing(QQ,'y'), 'x')
sage: p = x^2 + 2*x + 4
sage: type(p)
<type 'sage.rings.polynomial.polynomial_element.Polynomial_generic_dense'>
sage: p.shift(0)
x^2 + 2*x + 4
sage: p.shift(-1)
x + 2
```
AUTHORS:
  • David Harvey (2006-08-06)

**truncate**(*n*)
Return the polynomial of degree `< n` which is equivalent to self modulo \( x^n \).

**EXAMPLES:**

```python
sage: S.<q> = QQ['t']['q']
sage: f = (1+q^10+q^11+q^12).truncate(11); f
q^10 + 1
sage: f = (1+q^10+q^100).truncate(50); f
q^10 + 1
sage: f.degree()
10
sage: f = (1+q^10+q^100).truncate(500); f
q^100 + q^10 + 1
```

**class** `sage.rings.polynomial.polynomial_element.Polynomial_generic_dense_inexact`
Bases: `sage.rings.polynomial.polynomial_element.Polynomial_generic_dense`
A dense polynomial over an inexact ring.

**AUTHOR:**
  • Xavier Caruso (2013-03)

**degree**(secure=False)

**INPUT:**
  • secure – a boolean (default: False)

**OUTPUT:**
The degree of self.

If `secure` is `True` and the degree of this polynomial is not determined (because the leading coefficient is indistinguishable from 0), an error is raised

If `secure` is `False`, the returned value is the largest \( n \) so that the coefficient of \( x^n \) does not compare equal to 0.

**EXAMPLES:**

```python
sage: K = Qp(3,10)
sage: R.<T> = K[]
sage: f = T + 2; f
(1 + O(3^10))*T + 2 + O(3^10)
sage: f.degree()
1
sage: (f-T).degree()
0
sage: (f-T).degree(secure=True)
Traceback (most recent call last):
...
PrecisionError: the leading coefficient is indistinguishable from 0

```sage
sage: x = O(3^5)
sage: li = [3^i * x for i in range(0,5)]; li
[0(3^5), 0(3^6), 0(3^7), 0(3^8), 0(3^9)]
sage: f = R(li); f
0(3^9)*T^4 + 0(3^8)*T^3 + 0(3^7)*T^2 + 0(3^6)*T + 0(3^5)
sage: f.degree()
-1
sage: f.degree(secure=True)
Traceback (most recent call last):
  ...  PrecisionError: the leading coefficient is indistinguishable from 0
```

AUTHOR:
• Xavier Caruso (2013-03)

```python
prec_degree()
```

Return the largest $n$ so that precision information is stored about the coefficient of $x^n$.

Always greater than or equal to degree.

EXAMPLES:

```sage
sage: K = Qp(3,10)
sage: R.<T> = K[]
sage: f = T + 2; f
(1 + O(3^10))*T + 2 + O(3^10)
sage: f.degree()
1
sage: f.prec_degree()
1

sage: g = f - T; g
O(3^10)*T + 2 + O(3^10)
sage: g.degree()
0
sage: g.prec_degree()
1
```

AUTHOR:
• Xavier Caruso (2013-03)

```python
sage.rings.polynomial.polynomial_element.generic_power_trunc(p, n, prec)
```

Generic truncated power algorithm

INPUT:
• $p$ - a polynomial
• $n$ - an integer (of type `sage.rings.integer.Integer`
• $prec$ - a precision (should fit into a C long)

```python
sage.rings.polynomial.polynomial_element.is_Polynomial()
```

Return True if $f$ is of type univariate polynomial.
INPUT:

- f – an object

EXAMPLES:

```
sage: from sage.rings.polynomial.polynomial_element import is_Polynomial
sage: R.<x> = ZZ[]
sage: is_Polynomial(x^3 + x + 1)
True
sage: S.<y> = R[

sage: f = y^3 + x*y - 3*x; f
y^3 + x*y - 3*x
sage: is_Polynomial(f)
True
```

However this function does not return True for genuine multivariate polynomial type objects or symbolic polynomials, since those are not of the same data type as univariate polynomials:

```
sage: R.<x,y> = QQ[

sage: f = y^3 + x*y - 3*x; f
y^3 + x*y - 3*x
sage: is_Polynomial(f)
False
sage: var('x,y')
(x, y)
sage: f = y^3 + x*y - 3*x; f
y^3 + x*y - 3*x
sage: is_Polynomial(f)
False
```

```
sage.rings.polynomial.polynomial_element.make_generic_polynomial(parent, coeffs)
sage.rings.polynomial.polynomial_element.universal_discriminant(n)
```

Return the discriminant of the ‘universal’ univariate polynomial \(a_nx^n + \cdots + a_1x + a_0\) in \(\mathbb{Z}[a_0, \ldots, a_n][x]\).

INPUT:

- n - degree of the polynomial

OUTPUT:

The discriminant as a polynomial in \(n+1\) variables over \(\mathbb{Z}\). The result will be cached, so subsequent computations of discriminants of the same degree will be faster.

EXAMPLES:

```
sage: from sage.rings.polynomial.polynomial_element import universal_discriminant
sage: universal_discriminant(1)
1
sage: universal_discriminant(2)
a1^2 - 4*a0*a2
sage: universal_discriminant(3)
a1^2*a2^2 - 4*a0*a2^3 - 4*a1*a3 + 18*a0*a1*a2*a3 - 27*a0^2*a3^2
sage: universal_discriminant(4).degrees()
(3, 4, 4, 3)
```
See also:

* `Polynomial.discriminant()`

## 2.1.4 Univariate Polynomials over domains and fields

AUTHORS:

- William Stein: first version
- Martin Albrecht: Added singular coercion.
- David Harvey: split off polynomial_integer_dense_ntl.pyx (2007-09)
- Robert Bradshaw: split off polynomial_modn_dense_ntl.pyx (2007-09)

```python
class sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_cdv(parent, is_gen=False, construct=False):

Bases: sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_domain

A generic class for polynomials over complete discrete valuation domains and fields.

AUTHOR:

- Xavier Caruso (2013-03)

*factor_of_slope*(slope=\texttt{None})

**INPUT:**

- slope – a rational number (default: the first slope in the Newton polygon of \texttt{self})

**OUTPUT:**

The factor of \texttt{self} corresponding to the slope \texttt{slope} (i.e. the unique monic divisor of \texttt{self} whose slope is \texttt{slope} and degree is the length of \texttt{slope} in the Newton polygon).

**EXAMPLES:**

```python
sage: K = Qp(5)
sage: R.<x> = K[]
sage: K = Qp(5)
sage: R.<t> = K[]
sage: f = 5 + 3*t + t^4 + 25*t^10
sage: f.newton_slopes()
[1, 0, 0, 0, -1/3, -1/3, -1/3, -1/3, -1/3, -1/3]

sage: g = f.factor_of_slope(0)
sage: g.newton_slopes()
[0, 0, 0]

sage: (f % g).is_zero()
True

sage: h = f.factor_of_slope()
sage: h.newton_slopes()
[1]

sage: (f % h).is_zero()
True
```
If slope is not a slope of self, the corresponding factor is 1:

```python
sage: f.factor_of_slope(-1)
1 + O(5^20)
```

AUTHOR:

- Xavier Caruso (2013-03-20)

**hensel_lift**

Lift $a$ to a root of this polynomial (using Newton iteration).

If $a$ is not close enough to a root (so that Newton iteration does not converge), an error is raised.

EXAMPLES:

```python
sage: K = Qp(5, 10)
sage: P.<x> = PolynomialRing(K)
sage: f = x^2 + 1
sage: root = f.hensel_lift(2); root
2 + 5 + 2*5^2 + 5^3 + 3*5^4 + 4*5^5 + 2*5^6 + 3*5^7 + 3*5^9 + O(5^10)
sage: f(root)
O(5^10)
sage: g = (x^2 + 1)*(x - 7)
sage: g.hensel_lift(2)  # here, 2 is a multiple root modulo p
Traceback (most recent call last):
... ValueError: a is not close enough to a root of this polynomial
```

AUTHOR:

- Xavier Caruso (2013-03-23)

**newton_polygon**

Returns a list of vertices of the Newton polygon of this polynomial.

**Note:** If some coefficients have not enough precision an error is raised.

EXAMPLES:

```python
sage: K = Qp(5)
sage: R.<t> = K[

sage: f = 5 + 3*t + t^4 + 25*t^10
sage: f.newton_polygon()
Finite Newton polygon with 4 vertices: ((0, 1), (1, 0), (4, 0), (10, 2))

sage: g = f + K(0,0)*t^4; g
(5^2 + O(5^22))*t^10 + O(5^0)*t^4 + (3 + O(5^20))*t + 5 + O(5^21)

sage: g.newton_polygon()
Traceback (most recent call last):
... PrecisionError: The coefficient of t^4 has not enough precision
```

AUTHOR:

- Xavier Caruso (2013-03-20)
newton_slopes(repetition=True)

Returns a list of the Newton slopes of this polynomial.

These are the valuations of the roots of this polynomial.

If repetition is True, each slope is repeated a number of times equal to its multiplicity. Otherwise it appears only one time.

EXAMPLES:

```python
sage: K = Qp(5)
sage: R.<t> = K[]
sage: f = 5 + 3*t + t^4 + 25*t^10
sage: f.newton_polygon()
Finite Newton polygon with 4 vertices: (0, 1), (1, 0), (4, 0), (10, 2)
sage: f.newton_slopes()
[1, 0, 0, 0, -1/3, -1/3, -1/3, -1/3, -1/3, -1/3]
sage: f.newton_slopes(repetition=False)
[1, 0, -1/3]
```

AUTHOR:

• Xavier Caruso (2013-03-20)

slope_factorization()

Return a factorization of self into a product of factors corresponding to each slope in the Newton polygon.

EXAMPLES:

```python
sage: K = Qp(5)
sage: R.<x> = K[]
sage: K = Qp(5)
sage: R.<t> = K[]
sage: f = 5 + 3*t + t^4 + 25*t^10
sage: f.newton_slopes()
[1, 0, 0, 0, -1/3, -1/3, -1/3, -1/3, -1/3, -1/3]
sage: F = f.slope_factorization()
sage: F.prod() == f
True
sage: for (f,_) in F:
    print(f.newton_slopes())
[-1/3, -1/3, -1/3, -1/3, -1/3]
[0, 0, 0]
[1]
```

AUTHOR:

• Xavier Caruso (2013-03-20)
class sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_cdvr(
    parent, 
    is_gen=False, 
    construct=False) 

Bases: sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_cdv

class sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_dense_cdv 

    rings.polynomial.polynomial_element_generic.Polynomial_generic_cdv 

class sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_dense_cdvf 

Bases: sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_dense_cdv, 
    sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_cdvf 

class sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_dense_cdvr 

Bases: sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_dense_cdv, 
    sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_cdvr 

class sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_dense_field(
    parent, 
    x=None, 
    check=True, 
    is_gen=False, 
    construct=False) 

    polynomial.polynomial_element_generic.Polynomial_generic_field 

class sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_domain 

    IntegralDomainElement

is_unit() 
Return True if this polynomial is a unit.

EXERCISE (Atiyah-McDonald, Ch 1): Let $A[x]$ be a polynomial ring in one variable. Then $f = \sum a_i x^i \in 
A[x]$ is a unit if and only if $a_0$ is a unit and $a_1, \ldots, a_n$ are nilpotent.

EXAMPLES:

```
sage: R.<z> = PolynomialRing(ZZ, sparse=True)
sage: (2 + z^3).is_unit()
False
sage: f = -1 + 3*z^3; f
3*z^3 - 1
sage: f.is_unit()
False
sage: R(-3).is_unit()
False
sage: R(-1).is_unit()
True
sage: R(0).is_unit()
False
```
class sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_field(
    parent, is_gen=False, construct=False)
Bases: sage.rings.polynomial.polynomial_singular_interface.Polynomial_singular_repr,
    sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_domain,
    sage.structure.element.EuclideanDomainElement

quo_rem(other)

Returns a tuple (quotient, remainder) where self = quotient * other + remainder.

EXAMPLES:

```
sage: R.<y> = PolynomialRing(QQ)
sage: K.<t> = NumberField(y^2 - 2)
sage: P.<x> = PolynomialRing(K)
sage: x.quo_rem(K(1))
(x, 0)
sage: x.xgcd(K(1))
(1, 0, 1)
```

class sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse(
    parent, x=None, check=True, is_gen=False, construct=False)
Bases: sage.rings.polynomial.polynomial_element.Polynomial

A generic sparse polynomial.

The Polynomial_generic_sparse class defines functionality for sparse polynomials over any base ring. A sparse polynomial is represented using a dictionary which maps each exponent to the corresponding coefficient. The coefficients must never be zero.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(PolynomialRing(QQ, 'y'), sparse=True)
sage: f = x^3 - x + 17
sage: type(f)
<class 'sage.rings.polynomial.polynomial_ring.PolynomialRing_integral_domain_with_category.element_class'>
sage: loads(f.dumps()) == f
True
```

A more extensive example:

```
sage: A.<T> = PolynomialRing(Integers(5),sparse=True) ; f = T^2+1 ; B = A.quo(f)
sage: C.<s> = PolynomialRing(B)
sage: C
Univariate Polynomial Ring in s over Univariate Quotient Polynomial Ring in Tbar over Ring of integers modulo 5 with modulus T^2 + 1
sage: s + T
s + Tbar
```

(continues on next page)
\[ \texttt{sage: } (s + T)^2 \]
\[ s^2 + 2Tbar*s + 4 \]

**coefficients**(\textit{sparse=True})

Return the coefficients of the monomials appearing in \texttt{self}.

**EXAMPLES:**

\[
\begin{align*}
\texttt{sage: } & \text{R.<w> = PolynomialRing(Integers(8), sparse=True) } \\
\texttt{sage: } & f = 5 + w^{1997} - w^{10000}; f \\
& 7^*w^{10000} + w^{1997} + 5 \\
\texttt{sage: } & f.coefficients() \\
& [5, 1, 7]
\end{align*}
\]

\[ \texttt{degree}(\textit{gen=None}) \]

Return the degree of this sparse polynomial.

**EXAMPLES:**

\[
\begin{align*}
\texttt{sage: } & \text{R.<z> = PolynomialRing(ZZ, sparse=True) } \\
\texttt{sage: } & f = 13*z^{50000} + 15*z^2 + 17*z \\
\texttt{sage: } & f.degree() \\
& 50000
\end{align*}
\]

\[ \texttt{dict}() \]

Return a new copy of the dict of the underlying elements of \texttt{self}.

**EXAMPLES:**

\[
\begin{align*}
\texttt{sage: } & \text{R.<w> = PolynomialRing(Integers(8), sparse=True) } \\
\texttt{sage: } & f = 5 + w^{1997} - w^{10000}; f \\
& 7^*w^{10000} + w^{1997} + 5 \\
\texttt{sage: } & d = f.dict(); d \\
& \{0: 5, 1997: 1, 10000: 7\} \\
\texttt{sage: } & d[0] = 10 \\
\texttt{sage: } & f.dict() \\
& \{0: 10, 1997: 1, 10000: 7\}
\end{align*}
\]

\[ \texttt{exponents}() \]

Return the exponents of the monomials appearing in \texttt{self}.

**EXAMPLES:**

\[
\begin{align*}
\texttt{sage: } & \text{R.<w> = PolynomialRing(Integers(8), sparse=True) } \\
\texttt{sage: } & f = 5 + w^{1997} - w^{10000}; f \\
& 7^*w^{10000} + w^{1997} + 5 \\
\texttt{sage: } & f.exponents() \\
& [0, 1997, 10000]
\end{align*}
\]

\[ \texttt{gcd}(\textit{other}, \textit{algorithm=None}) \]

Return the gcd of this polynomial and \texttt{other}.

**INPUT:**

- \texttt{other} – a polynomial defined over the same ring as this polynomial.

**ALGORITHM:**
Two algorithms are provided:

- **generic**: Uses the generic implementation, which depends on the base ring being a UFD or a field.
- **dense**: The polynomials are converted to the dense representation, their gcd is computed and is converted back to the sparse representation.

Default is `dense` for polynomials over `ZZ` and `generic` in the other cases.

**EXAMPLES:**

```python
sage: R.<x> = PolynomialRing(ZZ, sparse=True)
sage: p = x^6 + 7*x^5 + 8*x^4 + 6*x^3 + 2*x^2 + x + 2
sage: q = 2*x^4 - x^3 - 2*x^2 - 4*x - 1
sage: gcd(p,q)
x^2 + x + 1
sage: gcd(p, q, algorithm = "dense")
x^2 + x + 1
sage: gcd(p, q, algorithm = "generic")
x^2 + x + 1
sage: gcd(p, q, algorithm = "foobar")
Traceback (most recent call last):
  ... ValueError: Unknown algorithm 'foobar'
```

**integral**(var=None)

Return the integral of this polynomial.

By default, the integration variable is the variable of the polynomial.

Otherwise, the integration variable is the optional parameter `var`

**Note:** The integral is always chosen so that the constant term is 0.

**EXAMPLES:**

```python
sage: R.<x> = PolynomialRing(ZZ, sparse=True)
sage: (1 + 3*x^10 - 2*x^100).integral()
  -2/101*x^101 + 3/11*x^11 + x
```

**list**(copy=True)

Return a new copy of the list of the underlying elements of `self`.

**EXAMPLES:**

```python
sage: R.<z> = PolynomialRing(Integers(100), sparse=True)
sage: f = 13*z^5 + 15*z^2 + 17*z
sage: f.list()
[0, 17, 15, 0, 0, 13]
```

**number_of_terms**()

Return the number of nonzero terms.

**EXAMPLES:**

```python
sage: R.<x> = PolynomialRing(ZZ, sparse=True)
sage: p = x^100 - 3*x^10 + 12
```
**quo_rem**(other)

Returns the quotient and remainder of the Euclidean division of *self* and *other*.

Raises `ZerodivisionError` if *other* is zero. Raises `ArithmeticError` if *other* has a nonunit leading coefficient.

**EXAMPLES:**

```python
sage: P.<x> = PolynomialRing(ZZ, sparse=True)
sage: R.<y> = PolynomialRing(P, sparse=True)
sage: f = R.random_element(10)
sage: g = y^5 + R.random_element(4)
sage: q, r = f.quo_rem(g)
sage: f == q*g + r and r.degree() < g.degree()
True
sage: g = x*y^5
sage: f.quo_rem(g)
Traceback (most recent call last):
...
ArithmeticError: Division non exact (consider coercing to polynomials over the fraction field)
sage: g = 0
sage: f.quo_rem(g)
Traceback (most recent call last):
...
ZeroDivisionError: Division by zero polynomial
```

**reverse**(degree=None)

Return this polynomial but with the coefficients reversed.

If an optional degree argument is given the coefficient list will be truncated or zero padded as necessary and the reverse polynomial will have the specified degree.

**EXAMPLES:**

```python
sage: R.<x> = PolynomialRing(ZZ, sparse=True)
sage: p = x^4 + 2*x^2^100
sage: p.reverse()
x^1267650600228229401496703205372 + 2
sage: p.reverse(10)
x^6
```

**shift**(n)

Returns this polynomial multiplied by the power $x^n$.

If *n* is negative, terms below $x^n$ will be discarded. Does not change this polynomial.

**EXAMPLES:**
AUTHOR: - David Harvey (2006-08-06)

**truncate(n)**

Return the polynomial of degree $< n$ equal to $\textit{self}$ modulo $x^n$.

**valuation()**

Return the valuation of $\textit{self}$.

```python
sage: R.<x> = PolynomialRing(ZZ, sparse=True)
sage: p = x^100000 + 2*x + 4
sage: type(p)
<class 'sage.rings.polynomial.polynomial_ring.PolynomialRing_integral_domain_with_category.element_class'>
sage: p.shift(0)
x^100000 + 2*x + 4
sage: p.shift(-1)
x^99999 + 2
sage: p.shift(-100002)
0
sage: p.shift(2)
x^100002 + 2*x^3 + 4*x^2

sage: (x^11 + x^10 + 1).truncate(11)
x^10 + 1
sage: (x^2^500 + x^2^100 + 1).truncate(2^101)
x^1267650600228229401496763205376 + 1

sage: f = w^1997 - w^10000
sage: f.valuation()
1997
sage: R(19).valuation()
0
sage: R(0).valuation()
+Infinity
```

**class** `sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse_cdv`(*parent*, *x=None*, *check=True*, *is_gen=False*, *construct=False*)

**Bases:**

- `sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse_cdv`
- `sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse`
- `sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_cdv`
class sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse_cdvf(
    parent,
    x=None,
    check=True,
    is_gen=False,
    construct=False)

Bases: sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse_cdv,
    sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_cdvf

class sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse_cdvr(
    parent,
    x=None,
    check=True,
    is_gen=False,
    construct=False)

Bases: sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse_cdv,
    sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_cdvr

class sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse_field(
    parent,
    x=None,
    check=True,
    is_gen=False,
    construct=False)

Bases: sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse,
    sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_field

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(Frac(RR['t']), sparse=True)
sage: f = x^3 - x + 17
sage: type(f)
<class 'sage.rings.polynomial.polynomial_ring.PolynomialRing_field_with_category.element_class'>
sage: loads(f.dumps()) == f
True
```

2.1.5 Univariate Polynomials over GF(2) via NTL’s GF2X

AUTHOR: - Martin Albrecht (2008-10) initial implementation

sage.rings.polynomial.polynomial_gf2x.GF2X_BuildIrred_list(n)

Return the list of coefficients of the lexicographically smallest irreducible polynomial of degree \(n\) over the field of 2 elements.

EXAMPLES:

```python
sage: from sage.rings.polynomial.polynomial_gf2x import GF2X_BuildIrred_list
sage: GF2X_BuildIrred_list(2)
[1, 1, 1]
sage: GF2X_BuildIrred_list(3)
[1, 1, 0, 1]
sage: GF2X_BuildIrred_list(4)
[1, 1, 0, 0, 1]
```

(continues on next page)
sage: GF(2)['x'](GF2X_BuildIrred_list(33))
x^33 + x^6 + x^3 + x + 1

sage.rings.polynomial.polynomial_gf2x.GF2X_BuildRandomIrred_list(n)
Return the list of coefficients of an irreducible polynomial of degree \( n \) of minimal weight over the field of 2 elements.

EXAMPLES:

```python
sage: from sage.rings.polynomial.polynomial_gf2x import GF2X_BuildRandomIrred_list
sage: GF2X_BuildRandomIrred_list(2)
[1, 1, 1]
sage: GF2X_BuildRandomIrred_list(3) in [[1, 1, 0, 1], [1, 0, 1, 1]]
True
```

sage.rings.polynomial.polynomial_gf2x.GF2X_BuildSparseIrred_list(n)
Return the list of coefficients of an irreducible polynomial of degree \( n \) of minimal weight over the field of 2 elements.

EXAMPLES:

```python
sage: from sage.rings.polynomial.polynomial_gf2x import GF2X_BuildIrred_list, GF2X_BuildSparseIrred_list
sage: all([GF2X_BuildSparseIrred_list(n) == GF2X_BuildIrred_list(n) for n in range(1,33)])
True
sage: GF(2)['x'](GF2X_BuildSparseIrred_list(33))
x^33 + x^10 + 1
```

class sage.rings.polynomial.polynomial_gf2x.Polynomial_GF2X
Bases: sage.rings.polynomial.polynomial_gf2x.Polynomial_template

Univariate Polynomials over GF(2) via NTL's GF2X.

EXAMPLES:

```python
sage: P.<x> = GF(2)[]
sage: x^3 + x^2 + 1
x^3 + x^2 + 1
```

is_irreducible()
Return whether this polynomial is irreducible over \( \mathbb{F}_2 \).

EXAMPLES:

```python
sage: R.<x> = GF(2)[]
sage: (x^2 + 1).is_irreducible()
False
sage: (x^3 + x + 1).is_irreducible()
True
```

Test that caching works:

```python
sage: R.<x> = GF(2)[]
sage: f = x^2 + 1
(continues on next page)```
sage: f.is_irreducible()
False
sage: f.is_irreducible.cache
False

modular_composition\((g, h, algorithm=None)\)
Compute \(f(g) \pmod{h}\).


INPUT:

- \(g\) – a polynomial
- \(h\) – a polynomial
- \(algorithm\) – either ‘native’ or ‘ntl’ (default: ‘native’)

EXAMPLES:

```python
sage: P.<x> = GF(2)[]
sage: r = 279
sage: f = x^r + x +1
sage: g = x^r
sage: g.modular_composition(g, f) == g(g) % f
True
sage: P.<x> = GF(2)[]
sage: f = x^29 + x^24 + x^22 + x^20 + x^16 + x^15 + x^14 + x^10 + x^9 + x^8 + x^7 + x^6 + x^5 + x^2
sage: g = x^31 + x^30 + x^28 + x^26 + x^24 + x^21 + x^19 + x^18 + x^11 + x^10 + x^9 + x^8 + x^5 + x^2 + 1
sage: h = x^30 + x^28 + x^26 + x^25 + x^24 + x^22 + x^21 + x^18 + x^17 + x^15 + x^13 + x^12 + x^11 + x^10 + x^9 + x^4
sage: f.modular_composition(g,h) == f(g) % h
True
```

AUTHORS:

- Paul Zimmermann (2008-10) initial implementation
- Martin Albrecht (2008-10) performance improvements

**class** sage.rings.polynomial.polynomial_gf2x.Polynomial_template
**Bases:** sage.rings.polynomial.polynomial_element.Polynomial

Template for interfacing to external C / C++ libraries for implementations of polynomials.

AUTHORS:

- Robert Bradshaw (2008-10): original idea for templating
- Martin Albrecht (2008-10): initial implementation

This file implements a simple templating engine for linking univariate polynomials to their C/C++ library implementations. It requires a ‘linkage’ file which implements the `element_` functions (see sage.libsntl.ntl_GF2X_linkage for an example). Both parts are then plugged together by inclusion of the linkage file when inheriting from this class. See sage.rings.polynomial.polynomial_gf2x for an example.
We illustrate the generic glueing using univariate polynomials over $\text{GF}(2)$.

**Note:** Implementations using this template MUST implement coercion from base ring elements and `get_unsafe()`. See `Polynomial_GF2X` for an example.

### degree()

**EXAMPLES:**

```
sage: P.<x> = GF(2)[]
sage: x.degree()
1
sage: P(1).degree()
0
sage: P(0).degree()
-1
```

### gcd(other)

Return the greatest common divisor of self and other.

**EXAMPLES:**

```
sage: P.<x> = GF(2)[]
sage: f = x*(x+1)
sage: f.gcd(x+1)
x + 1
sage: f.gcd(x^2)
x
```

### get_cparent()

### is_gen()

**EXAMPLES:**

```
sage: P.<x> = GF(2)[]
sage: x.is_gen()
True
sage: (x+1).is_gen()
False
```

### is_one()

**EXAMPLES:**

```
sage: P.<x> = GF(2)[]
sage: P(1).is_one()
True
```

### is_zero()

**EXAMPLES:**

```
sage: P.<x> = GF(2)[]
sage: x.is_zero()
False
```

### list(copy=True)

**EXAMPLES:**
sage: P.<x> = GF(2)[]
sage: x.list()
[0, 1]
sage: list(x)
[0, 1]

**quo_rem(right)**

EXAMPLES:

```
sage: P.<x> = GF(2)[]
sage: f = x^2 + x + 1
sage: f.quo_rem(x + 1)
(x, 1)
```

**shift(n)**

EXAMPLES:

```
sage: P.<x> = GF(2)[]
sage: f = x^3 + x^2 + 1
sage: f.shift(1)
x^4 + x^3 + x
sage: f.shift(-1)
x^2 + x
```

**truncate(n)**

Returns this polynomial mod \(x^n\).

EXAMPLES:

```
sage: R.<x> =GF(2)[]
sage: f = sum(x^n for n in range(10)); f
x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1
sage: f.truncate(6)
x^5 + x^4 + x^3 + x^2 + x + 1
```

If the precision is higher than the degree of the polynomial then the polynomial itself is returned:

```
sage: f.truncate(10) is f
True
```

If the precision is negative, the zero polynomial is returned:

```
sage: f.truncate(-1)
0
```

**xgcd(other)**

Computes extended gcd of self and other.

EXAMPLES:

```
sage: P.<x> = GF(7)[]
sage: f = x*(x+1)
sage: f.xgcd(x+1)
(x + 1, 0, 1)
```
sage: f.xgcd(x^2)
(x, 1, 6)

sage.rings.polynomial.polynomial_gf2x.make_element(parent, args)

## 2.1.6 Univariate polynomials over number fields.

**AUTHOR:**


**EXAMPLES:**

Define a polynomial over an absolute number field and perform basic operations with them:

```python
sage: N.<a> = NumberField(x^2-2)
sage: K.<x> = N[]
sage: f = x - a
sage: g = x^3 - 2*a + 1
sage: f*(x + a)
x^2 - 2
sage: f + g
x^3 + x - 3*a + 1
sage: g // f
x^2 + a*x + 2
sage: g % f
1
sage: factor(x^3 - 2*a*x^2 - 2*x + 4*a)
(x - 2*a) * (x - a) * (x + a)
sage: gcd(f, x - a)
x - a
```

Polynomials are aware of embeddings of the underlying field:

```python
sage: x = var('x')
sage: Q7 = Qp(7)
sage: r1 = Q7(3 + 7 + 2*7^2 + 6*7^3 + 7^4 + 2*7^5 + 7^6 + 2*7^7 + 4*7^8 +
6*7^9 + 6*7^10 + 2*7^11 + 7^12 + 7^13 + 2*7^15 + 7^16 + 7^17 +
4*7^18 + 6*7^19)
sage: N.<b> = NumberField(x^2-2, embedding = r1)
sage: K.<t> = N[]
sage: f = t^3-2*t+1
sage: f(r1)
1 + O(7^20)
```

We can also construct polynomials over relative number fields:

```python
sage: N.<i, s2> = QQ[I, sqrt(2)]
sage: K.<x> = N[]
sage: f = x - s2
sage: g = x^3 - 2*i*x^2 + s2*x
sage: f*(x + s2)
x^2 - 2
```

(continues on next page)
\begin{verbatim}
sage: f + g
x^3 - 2*I*x^2 + (sqrt2 + 1)*x - sqrt2
sage: g // f
x^2 + (-2*I + sqrt2)*x - 2*sqrt2*I + sqrt2 + 2
sage: g % f
-4*I + 2*sqrt2 + 2
sage: factor(i*x^4 - 2*i*x^2 + 9*i)
(I) * (x - I + sqrt2) * (x + I - sqrt2) * (x - I - sqrt2) * (x + I + sqrt2)
sage: gcd(f, x-i)
1

class sage.rings.polynomial.polynomial_number_field.Polynomial_absolute_number_field_dense

Bases: sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_dense_field

Class of dense univariate polynomials over an absolute number field.

\texttt{gcd(\textit{other})}
\hspace{1cm} Compute the monic gcd of two univariate polynomials using PARI.

\textbf{INPUT:}
\begin{itemize}
  \item \textit{other} – a polynomial with the same parent as \textit{self}.
\end{itemize}

\textbf{OUTPUT:}
\begin{itemize}
  \item The monic gcd of \textit{self} and \textit{other}.
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
sage: N.<a> = NumberField(x^3-1/2, 'a')
sage: R.<x> = N['x']
sage: f = (5/4*a^2 - 2*a + 4)*x^2 + (5*a^2 - 81/5*a - 17/2)*x + 4/5*a^2 + 24*a + 6
sage: g = (5/4*a^2 - 2*a + 4)*x^2 + (-11*a^2 + 79/5*a - 7/2)*x - 4/5*a^2 - 24*a - 6
sage: gcd(f, g**2)
r - 60808/96625*a^2 - 69936/96625*a - 149212/96625
sage: R = QQ[I]['x']
sage: f = R.random_element(2)
sage: g = f + 1
sage: h = R.random_element(2).monic()
sage: f *=h
sage: g *=h
sage: gcd(f, g) - h
0
sage: f.gcd(g) - h
0
\end{verbatim}
\end{verbatim}

2.1. Univariate Polynomials and Polynomial Rings 127
class sage.rings.polynomial.polynomial_number_field.Polynomial_relative_number_field_dense(parent, x=None, check=True, is_gen=False, construct=False)

Bases: sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_dense_field

Class of dense univariate polynomials over a relative number field.

gcd(other)

Compute the monic gcd of two polynomials.

Currently, the method checks corner cases in which one of the polynomials is zero or a constant. Then, computes an absolute extension and performs the computations there.

INPUT:

• other – a polynomial with the same parent as self.

OUTPUT:

• The monic gcd of self and other.

See Polynomial_absolute_number_field_dense.gcd() for more details.

EXAMPLES:

```python
sage: N = QQ[sqrt(2), sqrt(3)]
sage: s2, s3 = N.gens()
sage: x = polygen(N)
sage: f = x^4 - 5*x^2 + 6
sage: g = x^3 + (-2*s2 + s3)*x^2 + (-2*s3*s2 + 2)*x + 2*s3
sage: gcd(f, g)
x^2 + (-sqrt2 + sqrt3)*x - sqrt3*sqrt2
sage: f.gcd(g)
x^2 + (-sqrt2 + sqrt3)*x - sqrt3*sqrt2
```

### 2.1.7 Dense univariate polynomials over \( \mathbb{Z} \), implemented using FLINT

AUTHORS:

• David Harvey: rewrote to talk to NTL directly, instead of via ntl.pyx (2007-09); a lot of this was based on Joel Mohler’s recent rewrite of the NTL wrapper
• David Harvey: split off from polynomial_element_generic.py (2007-09)
• Burcin Erocal: rewrote to use FLINT (2008-06-16)

class sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint

Bases: sage.rings.polynomial.polynomial_element.Polynomial

A dense polynomial over the integers, implemented via FLINT.

```python
_add_(right)

Returns self plus right.

EXAMPLES:
```
sage: R.<x> = PolynomialRing(ZZ)
sage: f = 2*x + 1
sage: g = -3*x^2 + 6
sage: f + g
-3*x^2 + 2*x + 7

_sub_(right)
Return self minus right.

EXAMPLES:

sage: R.<x> = PolynomialRing(ZZ)
sage: f = 2*x + 1
sage: g = -3*x^2 + 6
sage: f - g
3*x^2 + 2*x - 5

_lmul_(right)
Returns self multiplied by right, where right is a scalar (integer).

EXAMPLES:

sage: R.<x> = PolynomialRing(ZZ)
sage: x^2
3*x
sage: (2*x^2 + 4)*3
6*x^2 + 12

_rmul_(right)
Returns self multiplied by right, where right is a scalar (integer).

EXAMPLES:

sage: R.<x> = PolynomialRing(ZZ)
sage: 3*x
3*x
sage: 3*(2*x^2 + 4)
6*x^2 + 12

_mul_(right)
Returns self multiplied by right.

EXAMPLES:

sage: R.<x> = PolynomialRing(ZZ)
sage: (x - 2)*(x^2 - 8*x + 16)
x^3 - 10*x^2 + 32*x - 32

_mul_trunc_(right, n)
Truncated multiplication

See also:

_mul_( ) for standard multiplication

EXAMPLES:
sage: x = polygen(ZZ)
sage: p1 = 1 + x + x^2 + x^4
sage: p2 = -2 + 3*x^2 + 5*x^4
sage: p1._mul_trunc_(p2, 4)
3*x^3 + x^2 - 2*x - 2
sage: (p1*p2).truncate(4)
3*x^3 + x^2 - 2*x - 2
sage: p1._mul_trunc_(p2, 6)
5*x^5 + 6*x^4 + 3*x^3 + x^2 - 2*x - 2

**content()**

Return the greatest common divisor of the coefficients of this polynomial. The sign is the sign of the leading coefficient. The content of the zero polynomial is zero.

**EXAMPLES:**

```python
sage: R.<x> = PolynomialRing(ZZ)
sage: (2*x^2 - 4*x^4 + 14*x^7).content()
2
sage: x.content()
1
sage: R(1).content()
1
sage: R(0).content()
0
```

**degree(gen=None)**

Return the degree of this polynomial.

The zero polynomial has degree -1.

**EXAMPLES:**

```python
sage: R.<x> = ZZ[]
sage: x.degree()
1
sage: (x^2).degree()
2
sage: R(1).degree()
0
sage: R(0).degree()
-1
```

**disc(proof=True)**

Return the discriminant of self, which is by definition

\((-1)^{m(m-1)/2} \text{resultant}(a, a')/\text{lc}(a),\)

where \(m = \text{deg}(a)\), and \(\text{lc}(a)\) is the leading coefficient of \(a\). If \(proof\) is False (the default is True), then this function may use a randomized strategy that errors with probability no more than \(2^{-80}\).

**EXAMPLES:**

```python
sage: R.<x> = ZZ[]
sage: f = 3*x^3 + 2*x + 1
```
\textbf{discriminant}(\textit{proof=True})

Return the discriminant of self, which is by definition

\[ (-1)^{m(m-1)/2} \text{resultant}(a, a') / \text{lc}(a), \]

where \( m = \deg(a) \), and \( \text{lc}(a) \) is the leading coefficient of \( a \). If \textit{proof} is False (the default is True), then this function may use a randomized strategy that errors with probability no more than \( 2^{-80} \).

\textbf{EXAMPLES:}

\begin{verbatim}
sage: R.<x> = ZZ[]
sage: f = 3*x^3 + 2*x + 1
sage: f.discriminant()
sage: -339
sage: f.discriminant(proof=False)
sage: -339
\end{verbatim}

\textbf{factor()}

This function overrides the generic polynomial factorization to make a somewhat intelligent decision to use Pari or NTL based on some benchmarking.

Note: This function factors the content of the polynomial, which can take very long if it’s a really big integer. If you do not need the content factored, divide it out of your polynomial before calling this function.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: R.<x> = ZZ[]
sage: f = x^4 - 1
sage: f.factor()
(x - 1) * (x + 1) * (x^2 + 1)
sage: f = 1 - x
sage: f.factor()
(-1) * (x - 1)
sage: f.factor().unit()
-1
sage: f = -30*x; f.factor()
(-1) * 2 * 3 * 5 * x
\end{verbatim}

\textbf{factor_mod}(p)

Return the factorization of self modulo the prime \( p \).

\textbf{INPUT:}

- \( p \) – prime

\textbf{OUTPUT:}

factorization of self reduced modulo \( p \).

\textbf{EXAMPLES:}
sage: R.<x> = ZZ['x']
sage: f = -3*x*(x-2)*(x-9) + x
sage: f.factor_mod(3)
x
sage: f = -3*x*(x-2)*(x-9)
sage: f.factor_mod(3)
Traceback (most recent call last):
  ... ArithmeticError: factorization of 0 is not defined
sage: f = 2*x*(x-2)*(x-9)
sage: f.factor_mod(7)
(2) * x * (x + 5)^2

factor_padic(p, prec=10)
Return \(p\)-adic factorization of self to given precision.

INPUT:

• \(p\) – prime
• \(prec\) – integer; the precision

OUTPUT:

• factorization of \(self\) over the completion at \(p\).

EXAMPLES:

sage: R.<x> = PolynomialRing(ZZ)
sage: f = x^2 + 1
sage: f.factor_padic(5, 4)
((1 + O(5^4))*x + 2 + 5 + 2*5^2 + 5^3 + O(5^4)) * ((1 + O(5^4))*x + 3 + 3*5 +
→2*5^2 + 3*5^3 + O(5^4))

A more difficult example:

sage: f = 100 * (5*x + 1)^2 * (x + 5)^2
sage: f.factor_padic(5, 10)
(4 + O(5^10)) * (5 + O(5^11))^2 * ((1 + O(5^10))*x + 5 + O(5^10))^2 * ((5 + O(5^10))^2 + 1 + O(5^10))^2

gcd(right)
Return the GCD of self and right. The leading coefficient need not be 1.

EXAMPLES:

sage: R.<x> = PolynomialRing(ZZ)
sage: f = (6*x + 47)*(7*x^2 - 2*x + 38)
sage: g = (6*x + 47)*(3*x^3 + 2*x + 1)
sage: f.gcd(g)
6*x + 47

inverse_series_trunc(prec)
Return a polynomial approximation of precision \(prec\) of the inverse series of this polynomial.

EXAMPLES:
```python
sage: x = polygen(ZZ)
sage: p = 1+x+2*x^2
sic: q5 = p.inverse_series_trunc(5)
sic: q5
-x^4 + 3*x^3 - x^2 - x + 1
sic: p*q5
-2*x^6 + 5*x^5 + 1
sage: (x-1).inverse_series_trunc(5)
-x^4 - x^3 - x^2 - x - 1
sage: q100 = p.inverse_series_trunc(100)
sic: (q100 * p).truncate(100)
1
```

**is_one()**

Returns True if self is equal to one.

**EXAMPLES:**

```python
sage: R.<x> = ZZ[]
sic: R(0).is_one()
False
sage: R(1).is_one()
True
sage: x.is_one()
False
```

**is_zero()**

Returns True if self is equal to zero.

**EXAMPLES:**

```python
sage: R.<x> = ZZ[]
sic: R(0).is_zero()
True
sage: R(1).is_zero()
False
sage: x.is_zero()
False
```

**lcm(right)**

Return the LCM of self and right.

**EXAMPLES:**

```python
sage: R.<x> = PolynomialRing(ZZ)
sage: f = (6*x + 47)*(7*x^2 - 2*x + 38)
sage: g = (6*x + 47)*(3*x^3 + 2*x + 1)
sage: h = f.lcm(g); h
126*x^6 + 951*x^5 + 486*x^4 + 6034*x^3 + 585*x^2 + 3706*x + 1786
sage: h == (6*x + 47)*(7*x^2 - 2*x + 38)*(3*x^3 + 2*x + 1)
True
```

**list(copy=True)**

Return a new copy of the list of the underlying elements of self.

2.1. Univariate Polynomials and Polynomial Rings 133
EXAMPLES:

```python
sage: x = PolynomialRing(ZZ,'x').0
sage: f = x^3 + 3*x - 17
sage: f.list()
[-17, 3, 0, 1]
sage: f = PolynomialRing(ZZ,'x')(0)
sage: f.list()
[]
```

**pseudo_divrem** *(B)*

Write \( A = \text{self} \). This function computes polynomials \( Q \) and \( R \) and an integer \( d \) such that

\[
\text{lead}(B)^d A = BQ + R
\]

where \( R \) has degree less than that of \( B \).

**INPUT:**

- \( B \) – a polynomial over \( \mathbb{Z} \)

**OUTPUT:**

- \( Q, R \) – polynomials
- \( d \) – nonnegative integer

**EXAMPLES:**

```python
sage: R.<x> = ZZ['x']
sage: A = R(range(10))
sage: B = 3*R([-1, 0, 1])
sage: Q, R, d = A.pseudo_divrem(B)
sage: Q, R, d
(9*x^7 + 8*x^6 + 16*x^5 + 14*x^4 + 21*x^3 + 18*x^2 + 24*x + 20, 75*x + 60, 1)
sage: B.leading_coefficient()^d * A == B*Q + R
True
```

**quo_rem** *(right)*

Attempts to divide self by right, and return a quotient and remainder.

**EXAMPLES:**

```python
sage: R.<x> = PolynomialRing(ZZ)
sage: f = R(range(10)); g = R([-1, 0, 1])
sage: q, r = f.quo_rem(g)
sage: q, r
(9*x^7 + 8*x^6 + 16*x^5 + 14*x^4 + 21*x^3 + 18*x^2 + 24*x + 20, 25*x + 20)
sage: q*g + r == f
True
sage: f = x^2
sage: f.quo_rem(0)
Traceback (most recent call last):
... ZeroDivisionError: division by zero polynomial
sage: f = (x^2 + 3) * (2*x - 1)
```

(continues on next page)
sage: f.quo_rem(2*x - 1)
(x^2 + 3, 0)
sage: f = x^2
sage: f.quo_rem(2*x - 1)
(0, x^2)

real_root_intervals()
Retruns isolating intervals for the real roots of this polynomial.

EXAMPLES: We compute the roots of the characteristic polynomial of some Salem numbers:

sage: R.<x> = PolynomialRing(ZZ)
sage: f = 1 - x^2 - x^3 - x^4 + x^6
sage: f.real_root_intervals()
[((1/2, 3/4), 1), ((1, 3/2), 1)]

resultant(other, proof=True)
Returns the resultant of self and other, which must lie in the same polynomial ring.

If proof = False (the default is proof=True), then this function may use a randomized strategy that
effects with probability no more than 2^{-80}.

INPUT:
• other – a polynomial

OUTPUT:
an element of the base ring of the polynomial ring

EXAMPLES:

sage: x = PolynomialRing(ZZ,'x').0
sage: f = x^3 + x + 1; g = x^3 - x - 1
sage: r = f.resultant(g); r
-8
sage: r.parent() is ZZ
True

reverse(degree=None)
Return a polynomial with the coefficients of this polynomial reversed.
If an optional degree argument is given the coefficient list will be truncated or zero padded as necessary
before computing the reverse.

EXAMPLES:

sage: R.<x> = ZZ[]
sage: p = R([1,2,3,4]); p
4*x^3 + 3*x^2 + 2*x + 1
sage: p.reverse()
x^3 + 2*x^2 + 3*x + 4
sage: p.reverse(degree=6)
x^6 + 2*x^5 + 3*x^4 + 4*x^3
sage: p.reverse(degree=2)
x^2 + 2*x + 3
revert_series(n)
Return a polynomial f such that \( f(self(x)) = self(f(x)) = x \mod x^n \).

EXAMPLES:

```
sage: R.<t> = ZZ[]
sage: f = t - t^3 + t^5
sage: f.revert_series(6)
2*t^5 + t^3 + t
sage: f.revert_series(-1)
Traceback (most recent call last):
... ValueError: argument n must be a non-negative integer, got -1
sage: g = - t^3 + t^5
sage: g.revert_series(6)
Traceback (most recent call last):
... ValueError: self must have constant coefficient 0 and a unit for coefficient t^1
```

squarefree_decomposition()
Return the square-free decomposition of self. This is a partial factorization of self into square-free, relatively prime polynomials.

This is a wrapper for the NTL function SquareFreeDecomp.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(ZZ)
sage: p = (x-1)^2 * (x-2)^2 * (x-3)^3 * (x-4)
sage: p.squarefree_decomposition()
(x - 4) * (x^2 - 3*x + 2)^2 * (x - 3)^3
sage: p = 37 * (x-1)^2 * (x-2)^2 * (x-3)^3 * (x-4)
sage: p.squarefree_decomposition()
(37) * (x - 4) * (x^2 - 3*x + 2)^2 * (x - 3)^3
```

xgcd(right)
Return a triple \((g, s, t)\) such that \( g = s * self + t * right \) and such that \( g \) is the \( \gcd \) of \( self \) and \( right \) up to a divisor of the resultant of \( self \) and \( other \).

As integer polynomials do not form a principal ideal domain, it is not always possible given \( a \) and \( b \) to find a pair \( s, t \) such that \( \gcd(a, b) = sa + tb \). Take \( a = x + 2 \) and \( b = x + 4 \) as an example for which the \( \gcd \) is 1 but the best you can achieve in the Bezout identity is 2.

If \( self \) and \( right \) are coprime as polynomials over the rationals, then \( g \) is guaranteed to be the resultant of \( self \) and \( right \), as a constant polynomial.

EXAMPLES:

```
sage: P.<x> = PolynomialRing(ZZ)
sage: (x+2).xgcd(x+4)
(2, -1, 1)
sage: (x+2).resultant(x+4)
2
sage: (x+2).gcd(x+4)
```

(continues on next page)
2.1.8 Dense univariate polynomials over \( \mathbb{Z} \), implemented using NTL.

AUTHORS:

- David Harvey: split off from polynomial_element_generic.py (2007-09)
- David Harvey: rewrote to talk to NTL directly, instead of via ntl.pyx (2007-09); a lot of this was based on Joel Mohler’s recent rewrite of the NTL wrapper

Sage includes two implementations of dense univariate polynomials over \( \mathbb{Z} \); this file contains the implementation based on NTL, but there is also an implementation based on FLINT in \texttt{sage.rings.polynomial.polynomial_integer_dense_flint}.

The FLINT implementation is preferred (FLINT’s arithmetic operations are generally faster), so it is the default; to use the NTL implementation, you can do:

```
sage: K.<x> = PolynomialRing(ZZ, implementation='NTL')
sage: K
Univariate Polynomial Ring in x over Integer Ring (using NTL)
```

class \texttt{sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl}

Bases: \texttt{sage.rings.polynomial.polynomial_element.Polynomial}

A dense polynomial over the integers, implemented via NTL.

\texttt{content()}

Return the greatest common divisor of the coefficients of this polynomial. The sign is the sign of the leading coefficient. The content of the zero polynomial is zero.

EXAMPLES:
sage: R.<x> = PolynomialRing(ZZ, implementation='NTL')
sage: (2*x^2 - 4*x^4 + 14*x^7).content()  
2  
sage: (2*x^2 - 4*x^4 - 14*x^7).content()  
-2  
sage: x.content()  
1  
sage: R(1).content()  
1  
sage: R(0).content()  
0

**degree**(gen=None)
Return the degree of this polynomial. The zero polynomial has degree -1.

**EXAMPLES:**

```
sage: R.<x> = PolynomialRing(ZZ, implementation='NTL')  
sage: x.degree()  
1  
sage: (x^2).degree()  
2  
sage: R(1).degree()  
0  
sage: R(0).degree()  
-1
```

discriminant**(proof=True)**
Return the discriminant of self, which is by definition

\[
(-1)^{m(m-1)/2} \text{resultant}(a, a') / \text{lcm}(a),
\]

where \( m = \text{deg}(a) \), and lcm(a) is the leading coefficient of a. If proof is False (the default is True), then this function may use a randomized strategy that errors with probability no more than \( 2^{-80} \).

**EXAMPLES:**

```
sage: f = ntl.ZZX([1,2,0,3])  
sage: f.discriminant()  
-339  
sage: f.discriminant(proof=False)  
-339
```

**factor**()
This function overrides the generic polynomial factorization to make a somewhat intelligent decision to use Pari or NTL based on some benchmarking.

Note: This function factors the content of the polynomial, which can take very long if it’s a really big integer. If you do not need the content factored, divide it out of your polynomial before calling this function.

**EXAMPLES:**

```
sage: R.<x>=ZZ[]  
sage: f=x^4-1  
sage: f.factor()  
(continues on next page)```
\[(x - 1) \cdot (x + 1) \cdot (x^2 + 1)\]

```
sage: f=1-x
sage: f.factor()
(-1) * (x - 1)
sage: f.factor().unit()
-1
sage: f = -30*x; f.factor()
(-1) * 2 * 3 * 5 * x
```

**factor_mod**(\(p\))

Return the factorization of self modulo the prime \(p\).

INPUT:

\* \(p\) – prime

OUTPUT: factorization of self reduced modulo \(p\).

EXAMPLES:

```
sage: R.<x> = PolynomialRing(ZZ, 'x', implementation='NTL')
sage: f = -3*x*(x-2)*(x-9) + x
sage: f.factor_mod(3)
x
sage: f = -3*x*(x-2)*(x-9)
sage: f.factor_mod(3)
Traceback (most recent call last):
  ...
ArithmeticError: factorization of 0 is not defined
```

```
sage: f = 2*x*(x-2)*(x-9)
sage: f.factor_mod(7)
(2) * x * (x + 5)^2
```

**factor_padic**(\(p, prec=10\))

Return \(p\)-adic factorization of self to given precision.

INPUT:

\* \(p\) – prime

\* \(prec\) – integer; the precision

OUTPUT:

\* factorization of self over the completion at \(p\).

EXAMPLES:

```
sage: R.<x> = PolynomialRing(ZZ, implementation='NTL')
sage: f = x^2 + 1
sage: f.factor_padic(5, 4)
((1 + O(5^4))*x + 2 + 5 + 2*5^2 + 5^3 + O(5^4)) * ((1 + O(5^4))*x + 3 + 3*5 + O(5^4))
```

A more difficult example:
**gcd(right)**
Return the GCD of self and right. The leading coefficient need not be 1.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(ZZ, implementation='NTL')
sage: f = (6*x + 47)*(7*x^2 - 2*x + 38)
sage: g = (6*x + 47)*(3*x^3 + 2*x + 1)
sage: f.gcd(g)
6*x + 47
```

**lcm(right)**
Return the LCM of self and right.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(ZZ, implementation='NTL')
sage: f = (6*x + 47)*(7*x^2 - 2*x + 38)
sage: g = (6*x + 47)*(3*x^3 + 2*x + 1)
sage: h = f.lcm(g); h
126*x^6 + 951*x^5 + 486*x^4 + 6034*x^3 + 585*x^2 + 3706*x + 1786
sage: h == (6*x + 47)*(7*x^2 - 2*x + 38)*(3*x^3 + 2*x + 1)
True
```

**list(copy=True)**
Return a new copy of the list of the underlying elements of self.

EXAMPLES:

```
sage: x = PolynomialRing(ZZ,'x', implementation='NTL').0
sage: f = x^3 + 3*x - 17
sage: f.list()
[-17, 3, 0, 1]
sage: f = PolynomialRing(ZZ,'x', implementation='NTL')(0)
sage: f.list()
[]
```

**quo_rem(right)**
Attempts to divide self by right, and return a quotient and remainder.

If right is monic, then it returns \((q, r)\) where \(self = q \times right + r\) and \(\text{deg}(r) < \text{deg}(right)\).

If right is not monic, then it returns \((q, 0)\) where \(q = \text{self}/\text{right}\) if right exactly divides self, otherwise it raises an exception.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(ZZ, implementation='NTL')
sage: f = R(range(10)); g = R([-1, 0, 1])
sage: q, r = f.quo_rem(g)
sage: q, r
```
(9\times x^7 + 8\times x^6 + 16\times x^5 + 14\times x^4 + 21\times x^3 + 18\times x^2 + 24\times x + 20, 25\times x + 20)
sage: q\cdot g + r == f
True

sage: 0//(2\times x)
0

sage: f = x^2
sage: f.quo_rem(0)
Traceback (most recent call last):
  ... ArithmeticError: division by zero polynomial

sage: f = (x^2 + 3) \times (2\times x - 1)
sage: f.quo_rem(2\times x - 1)
(x^2 + 3, 0)

sage: f = x^2
sage: f.quo_rem(2\times x - 1)
Traceback (most recent call last):
  ... ArithmeticError: division not exact in Z[x] (consider coercing to Q[x] first)

real_root_intervals()

Returns isolating intervals for the real roots of this polynomial.

EXAMPLES: We compute the roots of the characteristic polynomial of some Salem numbers:

sage: R.<x> = PolynomialRing(ZZ, implementation='NTL')
sage: f = 1 - x^2 - x^3 - x^4 + x^6
sage: f.real_root_intervals()
[((1/2, 3/4), 1), ((1, 3/2), 1)]

resultant(other, proof=True)

Returns the resultant of self and other, which must lie in the same polynomial ring.

If proof = False (the default is proof=True), then this function may use a randomized strategy that errors
with probability no more than 2^{-80}.

INPUT:
  * other -- a polynomial

OUTPUT:
 an element of the base ring of the polynomial ring

EXAMPLES:

sage: x = PolynomialRing(ZZ,'x',implementation='NTL').0
sage: f = x^3 + x + 1; g = x^3 - x - 1
sage: r = f.resultant(g); r
-8
sage: r.parent() is ZZ
True
squarefree_decomposition()

Return the square-free decomposition of self. This is a partial factorization of self into square-free, relatively prime polynomials.

This is a wrapper for the NTL function SquareFreeDecomp.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(ZZ, implementation='NTL')
sage: p = 37 * (x-1)^2 * (x-2)^2 * (x-3)^3 * (x-4)
sage: p.squarefree_decomposition()
(37) * (x - 4) * (x^2 - 3*x + 2)^2 * (x - 3)^3
```

xgcd(right)

This function can’t in general return \((g, s, t)\) as above, since they need not exist. Instead, over the integers, we first multiply \(g\) by a divisor of the resultant of \(a/g\) and \(b/g\), up to sign, and return \(g, u, v\) such that \(g = s*self + s*right\). But note that this \(g\) may be a multiple of the gcd.

If self and right are coprime as polynomials over the rationals, then \(g\) is guaranteed to be the resultant of self and right, as a constant polynomial.

EXAMPLES:

```
sage: P.<x> = PolynomialRing(ZZ, implementation='NTL')
sage: F = (x^2 + 2)*x^3; G = (x^2+2)*(x-3)
sage: g, u, v = F.xgcd(G)
sage: g, u, v
(27*x^2 + 54, 1, -x^2 - 3*x - 9)
sage: u*F + v*G
27*x^2 + 54
sage: x.xgcd(P(0))
(x, 1, 0)
sage: f = P(0)
sage: f.xgcd(x)
(x, 0, 1)
sage: F = (x-3)^3; G = (x-15)^2
sage: g, u, v = F.xgcd(G)
sage: g, u, v
(2985984, -432*x + 8208, 432*x^2 + 864*x + 14256)
sage: u*F + v*G
2985984
```

2.1.9 Univariate polynomials over \(\mathbb{Q}\) implemented via FLINT

AUTHOR:

• Sebastian Pancratz

class sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint

Bases: sage.rings.polynomial.polynomial_element.Polynomial

Univariate polynomials over the rationals, implemented via FLINT.

Internally, we represent rational polynomial as the quotient of an integer polynomial and a positive denominator which is coprime to the content of the numerator.

 additive(right)

Return the sum of two rational polynomials.
EXAMPLES:

```
 sage: R.<t> = QQ[]
sage: f = 2/3 + t + 2*t^3
sage: g = -1 + t/3 - 10/11*t^4
sage: f + g
-10/11*t^4 + 2*t^3 + 4/3*t - 1/3
```

**_sub_(right)**

Return the difference of two rational polynomials.

EXAMPLES:

```
 sage: R.<t> = QQ[]
sage: f = -10/11*t^4 + 2*t^3 + 4/3*t - 1/3
sage: g = 2*t^3
sage: f - g
-10/11*t^4 + 4/3*t - 1/3
```

**_lmul_(right)**

Return self * right, where right is a rational number.

EXAMPLES:

```
 sage: R.<t> = QQ[]
sage: f = 3/2*t^3 - t + 1/3
sage: f * 6
# indirect doctest
9*t^3 - 6*t + 2
```

**_rmul_(left)**

Return left * self, where left is a rational number.

EXAMPLES:

```
 sage: R.<t> = QQ[]
sage: f = 3/2*t^3 - t + 1/3
sage: 6 * f
# indirect doctest
9*t^3 - 6*t + 2
```

**_mul_(right)**

Return the product of self and right.

EXAMPLES:

```
 sage: R.<t> = QQ[]
sage: f = -1 + 3*t/2 - t^3
sage: g = 2/3 + 7/3*t + 3*t^2
sage: f * g
# indirect doctest
-3*t^5 - 7/3*t^4 + 23/6*t^3 + 1/2*t^2 - 4/3*t - 2/3
```

**_mul_trunc_(right, n)**

Truncated multiplication.

EXAMPLES:

```
 sage: x = polygen(QQ)
sage: p1 = 1/2 - 3*x + 2/7*x**3
```

(continues on next page)
sage: p2 = x + 2/5*x^5 + x^7
sage: p1._mul_trunc_(p2, 5)
2/7*x^4 - 3*x^2 + 1/2*x
sage: (p1*p2).truncate(5)
2/7*x^4 - 3*x^2 + 1/2*x
sage: p1._mul_trunc_(p2, 1)
0
sage: p1._mul_trunc_(p2, 0)
Traceback (most recent call last):
  ... ValueError: n must be > 0

ALGORITHM:

Call the FLINT method fmpq_poly_mullow.

degree()

Return the degree of self.

By convention, the degree of the zero polynomial is -1.

EXAMPLES:

sage: R.<t> = QQ[]
sage: f = 1 + t + t^2/2 + t^3/3 + t^4/4
sage: f.degree()
4
sage: g = R(0)
sage: g.degree()
-1

denominator()

Return the denominator of self.

EXAMPLES:

sage: R.<t> = QQ[]
sage: f = (3 * t^3 + 1) / -3
sage: f.denominator()
3

disc()

Return the discriminant of this polynomial.

The discriminant $R_n$ is defined as

$$R_n = a_n^{2n-2} \prod_{1 \leq i < j \leq n} (r_i - r_j)^2,$$

where $n$ is the degree of this polynomial, $a_n$ is the leading coefficient and the roots over $\mathbb{Q}$ are $r_1, \ldots, r_n$.

The discriminant of constant polynomials is defined to be 0.

OUTPUT:

- Discriminant, an element of the base ring of the polynomial ring
Note: Note the identity $R_n(f) := (-1)^{\frac{n}{2}}(n-1)/2)R(f, f')a_n^2n - k - 2)$, where $n$ is the degree of this polynomial, $a_n$ is the leading coefficient, $f'$ is the derivative of $f$, and $k$ is the degree of $f'$. Calls \texttt{resultant()}. ALGORITHM: Use PARI. EXAMPLES: In the case of elliptic curves in special form, the discriminant is easy to calculate:

```python
sage: R.<t> = QQ[]
sage: f = t^3 + t + 1
sage: d = f.discriminant(); d
-31
sage: d.parent() is QQ
True
sage: EllipticCurve([1, 1]).discriminant() / 16
-31
```

```python
sage: R.<t> = QQ[]
sage: f = 2*t^3 + t + 1
sage: d = f.discriminant(); d
-116
```

```python
sage: R.<t> = QQ[]
sage: f = t^3 + 3*t - 17
sage: f.discriminant()
-7911
```

discriminant()

Return the discriminant of this polynomial. The discriminant $R_n$ is defined as

$$R_n = a_{n}^{2n-2} \prod_{1 \leq i < j \leq n} (r_i - r_j)^2,$$

where $n$ is the degree of this polynomial, $a_n$ is the leading coefficient and the roots over $\mathbb{Q}$ are $r_1, \ldots, r_n$. The discriminant of constant polynomials is defined to be 0.

OUTPUT:

- Discriminant, an element of the base ring of the polynomial ring

Note: Note the identity $R_n(f) := (-1)^{\frac{n}{2}}(n-1)/2)R(f, f')a_n^2n - k - 2)$, where $n$ is the degree of this polynomial, $a_n$ is the leading coefficient, $f'$ is the derivative of $f$, and $k$ is the degree of $f'$. Calls \texttt{resultant()}. ALGORITHM: Use PARI. EXAMPLES:
In the case of elliptic curves in special form, the discriminant is easy to calculate:

```python
sage: R.<t> = QQ[]
sage: f = t^3 + t + 1
sage: d = f.discriminant(); d
-31
sage: d.parent() is QQ
True
sage: EllipticCurve([1, 1]).discriminant() / 16
-31
```

```python
sage: R.<t> = QQ[]
sage: f = 2*t^3 + t + 1
sage: d = f.discriminant(); d
-116
```

```python
sage: R.<t> = QQ[]
sage: f = t^3 + 3*t - 17
sage: f.discriminant()
-7911
```

**factor_mod(p)**

Return the factorization of self modulo the prime p.

Assumes that the degree of this polynomial is at least one, and raises a `ValueError` otherwise.

**INPUT:**

- p - Prime number

**OUTPUT:**

- Factorization of this polynomial modulo p

**EXAMPLES:**

```python
sage: R.<x> = QQ[]
sage: (x^5 + 17*x^3 + x+ 3).factor_mod(3)
x * (x^2 + 1)^2
sage: (x^5 + 2).factor_mod(5)
(x + 2)^5
```

Variable names that are reserved in PARI, such as `zeta`, are supported (see trac ticket #20631):

```python
sage: R.<zeta> = QQ[]
sage: (zeta^2 + zeta + 1).factor_mod(7)
(zeta + 3) * (zeta + 5)
```

**factor_padic(p, prec=10)**

Return the $p$-adic factorization of this polynomial to the given precision.

**INPUT:**

- p - Prime number
- prec - Integer; the precision

**OUTPUT:**
factorization of \texttt{self} viewed as a $p$-adic polynomial

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = x^3 - 2
sage: f.factor_padic(2)
(1 + O(2^10))*x^3 + O(2^10)*x^2 + 2 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 + ...
                   \rightarrow 2^7 + 2^8 + 2^9 + O(2^{10})
sage: f.factor_padic(3)
(1 + O(3^10))*x^3 + O(3^10)*x^2 + 0(3^{10})*x + 1 + 2*3 + 2*3^2 + 2*3^3 + 2*3^4 + ...
                   \rightarrow 2*3^4 + 2*3^5 + 2*3^6 + 2*3^7 + 2*3^8 + 2*3^9 + O(3^{10})
sage: f.factor_padic(5)
((1 + O(5^{10}))*x + 2 + 4*5 + 4*5^2 + 2*5^3 + 5*4 + 3*5^5 + 4*5^7 + 2*5^8 + 5*9 + ...
                   \rightarrow 0(5^{10})) * ((1 + O(5^{10}))*x + 2*5 + O(5^{10}))*x + 4 + 5 + 2*5^2 + 4*5^3 + 4*5^4 + 3*5^5 + 3*5^6 + ...
                   \rightarrow 4*5^7 + 4*5^9 + O(5^{10}))
```

The input polynomial is considered to have “infinite” precision, therefore the $p$-adic factorization of the polynomial is not the same as first coercing to $\mathbb{Q}_p$ and then factoring (see also trac ticket #15422):

```
sage: f = x^2 - 3^6
sage: f.factor_padic(3, 5)
((1 + O(3^5))*x + 3^3 + O(3^5)) * ((1 + O(3^5))*x + 2*3^3 + 2*3^4 + O(3^5))
sage: f.change_ring(Qp(3,5)).factor()
Traceback (most recent call last):
  ...  PrecisionError: p-adic factorization not well-defined since the discriminant is ...
                   \rightarrow zero up to the requestion p-adic precision
```

A more difficult example:

```
sage: f = 100 * (5*x + 1)^2 * (x + 5)^2
sage: f.factor_padic(5, 10)
(4*5^4 + O(5^{14})) * ((1 + O(5^{9}))*x + 5^-1 + O(5^{9}))*x^2 + ((1 + O(5^{10}))*x + 5 + ...
                   \rightarrow 0(5^{10}))*x^2
```

Try some bogus inputs:

```
sage: f.factor_padic(3,-1)
Traceback (most recent call last):
  ...  ValueError: prec_cap must be non-negative
sage: f.factor_padic(6,10)
Traceback (most recent call last):
  ...  ValueError: p must be prime
sage: f.factor_padic(’hello’, ’world’)  
Traceback (most recent call last):
  ...  TypeError: unable to convert ’hello’ to an integer
```

\texttt{galois\_group}(pari\_group=False, algorithm=’pari’)  
Return the Galois group of this polynomial as a permutation group.

INPUT:

\section*{2.1. Univariate Polynomials and Polynomial Rings}
• **self** - Irreducible polynomial

• **pari_group** - bool (default: False); if True instead return the Galois group as a PARI group. This has a useful label in it, and may be slightly faster since it doesn’t require looking up a group in Gap. To get a permutation group from a PARI group P, type `PermutationGroup(P)`.

• **algorithm** - 'pari', 'gap', 'kash', 'magma' (default: 'pari', for degrees is at most 11; 'gap', for degrees from 12 to 15; 'kash', for degrees from 16 or more).

**OUTPUT:**

• Galois group

**ALGORITHM:**

The Galois group is computed using PARI in C library mode, or possibly GAP, KASH, or MAGMA.

**Note:** The PARI documentation contains the following warning: The method used is that of resolvent polynomials and is sensitive to the current precision. The precision is updated internally but, in very rare cases, a wrong result may be returned if the initial precision was not sufficient.

GAP needs an optional transitive group library installed, from `database_gap spkg`.

MAGMA does not return a provably correct result. Please see the MAGMA documentation for how to obtain a provably correct result.

**EXAMPLES:**

```python
sage: R.<x> = QQ[]
sage: f = x^4 - 17*x^3 - 2*x + 1
sage: G = f.galois_group(); G
Transitive group number 5 of degree 4
sage: G.gens()
[(1,2), (1,2,3,4)]
sage: G.order()
24
```

It is potentially useful to instead obtain the corresponding PARI group, which is little more than a 4-tuple. See the PARI manual for the exact details. (Note that the third entry in the tuple is in the new standard ordering.)

```python
sage: f = x^4 - 17*x^3 - 2*x + 1
sage: G = f.galois_group(pari_group=True); G
PARI group [24, -1, 5, "S4"] of degree 4
sage: PermutationGroup(G)
Transitive group number 5 of degree 4
```

You can use KASH or GAP to compute Galois groups as well. The advantage is that KASH (resp. GAP) can compute Galois groups of fields up to degree 23 (resp. 15), whereas PARI only goes to degree 11. (In my not-so-thorough experiments PARI is faster than KASH.)

```python
sage: f = x^4 - 17*x^3 - 2*x + 1
sage: f.galois_group(algorithm='kash')  # optional - kash
Transitive group number 5 of degree 4
sage: f = x^4 - 17*x^3 - 2*x + 1
sage: f.galois_group(algorithm='gap')  # optional - database_gap
```

(continues on next page)
Transitive group number 5 of degree 4
\[
sage: f = x^{13} - 17x^3 - 2x + 1
\]
\[
sage: f.galois_group(algorithm='gap')  # optional - database_gap
\]
Transitive group number 9 of degree 13
\[
sage: f = x^9 - 2x^8 - x^7 + 2x^6 + 4x^4 - 2x^3 - x^2 - x + 1
\]
\[
sage: f.galois_group(algorithm='gap')  # optional - database_gap
\]
Transitive group number 183 of degree 12
\[
sage: f.galois_group(algorithm='magma')  # optional - magma
\]
Transitive group number 5 of degree 4
galois_group_davenport_smith_test(num_trials=50, assume_irreducible=False)

Use the Davenport-Smith test to attempt to certify that \(f\) has Galois group \(A_n\) or \(S_n\).

Return 1 if the Galois group is certified as \(S_n\), 2 if \(A_n\), or 0 if no conclusion is reached.

By default, we first check that \(f\) is irreducible. For extra efficiency, one can override this by specifying \texttt{assume_irreducible = True}; this yields undefined results if \(f\) is not irreducible.

A corresponding function in Magma is \texttt{IsEasySnAn}.

EXAMPLES:

\[
\begin{align*}
sage: & P.<x> = QQ[] \\
sage: & u = x^7 + x + 1 \\
sage: & u.galois_group_davenport_smith_test() \\
& 1 \\
sage: & u = x^7 - x^4 - x^3 + 3x^2 - 1 \\
sage: & u.galois_group_davenport_smith_test() \\
& 2 \\
sage: & u = x^7 - 2 \\
sage: & u.galois_group_davenport_smith_test() \\
& 0
\end{align*}
\]
gcd(right)

Return the (monic) greatest common divisor of self and right.

Corner cases: if self and right are both zero, returns zero. If only one of them is zero, returns the other polynomial, up to normalisation.

EXAMPLES:

\[
\begin{align*}
sage: & R.<t> = QQ[] \\
sage: & f = -2 + 3t/2 + 4t^2/7 - t^3 \\
sage: & g = 1/2 + 4t + 2t^4/3 \\
sage: & f.gcd(g) \\
& 1 \\
sage: & f = (-3t + 1/2) * f \\
sage: & g = (3t + 1/2) * (4t^2/3 - 1) * g \\
sage: & f.gcd(g) \\
& t - 1/6
\end{align*}
\]
hensel_lift(p, e)

Assuming that this polynomial factors modulo \(p\) into distinct monic factors, computes the Hensel lifts of these factors modulo \(p^e\). We assume that \(self\) has integer coefficients.

Return an empty list if this polynomial has degree less than one.
INPUT:
- p - Prime number; coercable to Integer
- e - Exponent; coercable to Integer

OUTPUT:
- Hensel lifts; list of polynomials over $\mathbb{Z}/p^e\mathbb{Z}$

EXAMPLES:
```
sage: R.<x> = QQ[]
sage: R((x-1)*(x+1)).hensel_lift(7, 2)
[x + 1, x + 48]
```

If the input polynomial $f$ is not monic, we get a factorization of $f/lc(f)$:
```
sage: R(2*x^2 - 2).hensel_lift(7, 2)
[x + 1, x + 48]
```

**inverse_series_trunc**(prec)
Return a polynomial approximation of precision `prec` of the inverse series of this polynomial.

EXAMPLES:
```
sage: x = polygen(QQ)
sage: p = 2 + x - 3/5*x^2
sage: q5 = p.inverse_series_trunc(5)
sage: q5
151/800*x^4 - 17/80*x^3 + 11/40*x^2 - 1/4*x + 1/2
sage: q5 * p
-453/4000*x^6 + 253/800*x^5 + 1
sage: q100 = p.inverse_series_trunc(100)
sage: (q100 * p).truncate(100)
1
```

**is_irreducible**()
Return whether this polynomial is irreducible.

This method computes the primitive part as an element of $\mathbb{Z}[t]$ and calls the method `is_irreducible` for elements of that polynomial ring.

By definition, over any integral domain, an element $r$ is irreducible if and only if it is non-zero, not a unit and whenever $r = ab$ then $a$ or $b$ is a unit.

EXAMPLES:
```
sage: R.<t> = QQ[]
sage: (t^2 + 2).is_irreducible()
True
sage: (t^2 - 1).is_irreducible()
False
```

**is_one**()
Return whether or not this polynomial is one.

EXAMPLES:
```
sage: R.<x> = QQ[]
sage: R([0,1]).is_one()
False
sage: R([1]).is_one()
True
sage: R([0]).is_one()
False
sage: R([-1]).is_one()
False
sage: R([1,1]).is_one()
False
```

**is_zero()**

Return whether or not self is the zero polynomial.

**EXAMPLES:**

```
sage: R.<t> = QQ[]
sage: f = 1 - t + 1/2*t^2 - 1/3*t^3
sage: f.is_zero()
False
sage: R(0).is_zero()
True
```

**lcm(right)**

Return the monic (or zero) least common multiple of self and right.

Corner cases: if either of self and right are zero, returns zero. This behaviour is ensures that the relation \(\text{lcm}(a,b) \equiv a \cdot b\) holds up to multiplication by rationals.

**EXAMPLES:**

```
sage: R.<t> = QQ[]
sage: f = -2 + 3*t/2 + 4*t^2/7 - t^3
sage: g = 1/2 + 4*t + 2*t^4/3
sage: f.lcm(g)
t^7 - 4/7*t^6 - 3/2*t^5 + 8*t^4 - 75/28*t^3 - 66/7*t^2 + 87/8*t + 3/2
sage: f.lcm(g) * f.gcd(g) // (f * g)
-3/2
```

**list(copy=True)**

Return a list with the coefficients of self.

**EXAMPLES:**

```
sage: R.<t> = QQ[]
sage: f = 1 + t + t^2/2 + t^3/3 + t^4/4
sage: f.list()
[1, 1, 1/2, 1/3, 1/4]
sage: g = R(0)
sage: g.list()
[]
```

**numerator()**

Return the numerator of self.

2.1. Univariate Polynomials and Polynomial Rings 151
Representing self as the quotient of an integer polynomial and a positive integer denominator (coprime to the content of the polynomial), returns the integer polynomial.

EXAMPLES:
```
sage: R.<t> = QQ[]
sage: f = (3 * t^3 + 1) / -3
sage: f.numerator()
-3*t^3 - 1
```

**quo_rem(right)**
Return the quotient and remainder of the Euclidean division of self and right.

Raises a ZerodivisionError if right is zero.

EXAMPLES:
```
sage: R.<t> = QQ[]
sage: g = R.random_element(1000)
sage: q, r = f.quo_rem(g)
sage: f == q*g + r
True
```

**real_root_intervals()**
Return isolating intervals for the real roots of self.

EXAMPLES:
```
We compute the roots of the characteristic polynomial of some Salem numbers:

sage: R.<t> = QQ[]
sage: f = 1 - t^2 - t^3 - t^4 + t^6
sage: f.real_root_intervals()
[[(1/2, 3/4), 1), ((1, 3/2), 1]]
```

**resultant(right)**
Return the resultant of self and right.

Enumerating the roots over \( \mathbb{Q} \) as \( r_1, \ldots, r_m \) and \( s_1, \ldots, s_n \) and letting \( x \) and \( y \) denote the leading coefficients of \( f \) and \( g \), the resultant of the two polynomials is defined by

\[
x^{\deg g} y^{\deg f} \prod_{i,j} (r_i - s_j).
\]

Corner cases: if one of the polynomials is zero, the resultant is zero. Note that otherwise if one of the polynomials is constant, the last term in the above is the empty product.

EXAMPLES:
```
sage: R.<t> = QQ[]
sage: f = (t - 2/3) * (t + 4/5) * (t - 1)
sage: g = (t - 1/3) * (t + 1/2) * (t + 1)
sage: f.resultant(g)
119/1350
sage: h = (t - 1/3) * (t + 1/2) * (t - 1)

sage: f.resultant(h)
0
```
**reverse**(degree=\texttt{None})
Reverse the coefficients of this polynomial (thought of as a polynomial of degree degree).

**INPUT:**
- degree (\texttt{None} or integral value that fits in an unsigned long, default: degree of self) - if specified, truncate or zero pad the list of coefficients to this degree before reversing it.

**EXAMPLES:**
We first consider the simplest case, where we reverse all coefficients of a polynomial and obtain a polynomial of the same degree:

```python
sage: R.<t> = QQ[]
sage: f = 1 + t + t^2 / 2 + t^3 / 3 + t^4 / 4
sage: f.reverse()
t^4 + t^3 + 1/2*t^2 + 1/3*t + 1/4
```

Next, an example where the returned polynomial has lower degree because the original polynomial has low coefficients equal to zero:

```python
sage: R.<t> = QQ[]
sage: f = 3/4*t^2 + 6*t^7
sage: f.reverse()
3/4*t^5 + 6
```

The next example illustrates the passing of a value for degree less than the length of self, notationally resulting in truncation prior to reversing:

```python
sage: R.<t> = QQ[]
sage: f = 1 + t + t^2 / 2 + t^3 / 3 + t^4 / 4
sage: f.reverse(2)
t^2 + t + 1/2
```

Now we illustrate the passing of a value for degree greater than the length of self, notationally resulting in zero padding at the top end prior to reversing:

```python
sage: R.<t> = QQ[]
sage: f = 1 + t + t^2 / 2 + t^3 / 3
sage: f.reverse(4)
t^4 + t^3 + 1/2*t^2 + 1/3*t
```

**revert_series**(\texttt{n})
Return a polynomial \( f \) such that \( f(self(x)) = self(f(x)) = x \mod x^n \).

**EXAMPLES:**

```python
sage: R.<t> = QQ[]
sage: f = t - t^3/6 + t^5/120
sage: f.revert_series(6)
3/40*t^5 + 1/60*t^3 + t

sage: f.revert_series(-1)
Traceback (most recent call last):
  ... ValueError: argument \texttt{n} must be a non-negative integer, got -1

sage: g = - t^3/3 + t^5/5
```

(continues on next page)
\textbf{truncate}(n)

Return self truncated modulo \( t^n \).

INPUT:

\begin{itemize}
  \item \texttt{n} - The power of \( t \) modulo which self is truncated
\end{itemize}

EXAMPLES:

\begin{verbatim}
 sage: R.<t> = QQ[]
 sage: f = 1 - t + 1/2*t^2 - 1/3*t^3
 sage: f.truncate(0)
 0
 sage: f.truncate(2)
-t + 1
\end{verbatim}

\textbf{xgcd}(\texttt{right})

Return polynomials \( d, s, \) and \( t \) such that \( d = s \times \text{self} + t \times \text{right} \), where \( d \) is the (monic) greatest common divisor of \text{self} and \text{right}. The choice of \( s \) and \( t \) is not specified any further.

Corner cases: if \text{self} and \text{right} are zero, returns zero polynomials. Otherwise, if only \text{self} is zero, returns \((d, s, t) = (\text{right}, 0, 1)\) up to normalisation, and similarly if only \text{right} is zero.

EXAMPLES:

\begin{verbatim}
 sage: R.<t> = QQ[]
 sage: f = 2/3 + 3/4 * t - t^2
 sage: g = -3 + 1/7 * t
 sage: f.xgcd(g)
 (1, -12/5095, -84/5095*t - 1701/5095)
\end{verbatim}

\section{2.10 Dense univariate polynomials over \( \mathbb{Z}/n\mathbb{Z} \), implemented using FLINT}

This module gives a fast implementation of \((\mathbb{Z}/n\mathbb{Z})[x]\) whenever \( n \) is at most \texttt{sys.maxsize}. We use it by default in preference to NTL when the modulus is small, falling back to NTL if the modulus is too large, as in the example below.

EXAMPLES:

\begin{verbatim}
 sage: R.<a> = PolynomialRing(Integers(100))
 sage: type(a)
 <type 'sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint'>
 sage: R.<a> = PolynomialRing(Integers(5*2^64))
 sage: type(a)
 <type 'sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_modn_ntl_ZZ'>
 sage: R.<a> = PolynomialRing(Integers(5*2^64), implementation="FLINT")
 Traceback (most recent call last):
   ...
 ValueError: FLINT does not support modulus 9223372036854775808
\end{verbatim}

AUTHORS:
class sage.rings.polynomial.polynomial_zmod_flint.Polynomial_template

Bases: sage.rings.polynomial.polynomial_element.Polynomial

Template for interfacing to external C / C++ libraries for implementations of polynomials.

AUTHORS:

- Robert Bradshaw (2008-10): original idea for templating
- Martin Albrecht (2008-10): initial implementation

This file implements a simple templating engine for linking univariate polynomials to their C/C++ library implementations. It requires a ‘linkage’ file which implements the celement_ functions (see sage.libs.ntl.ntl_GF2X_linkage for an example). Both parts are then plugged together by inclusion of the linkage file when inheriting from this class. See sage.rings.polynomial.polynomial_gf2x for an example.

We illustrate the generic glueing using univariate polynomials over GF(2).

Note: Implementations using this template MUST implement coercion from base ring elements and get_unsafe(). See Polynomial_GF2X for an example.

degree()

EXAMPLES:

```sage
sage: P.<x> = GF(2)[]
sage: x.degree()
1
sage: P(1).degree()
0
sage: P(0).degree()
-1
```

gcd(other)

Return the greatest common divisor of self and other.

EXAMPLES:

```sage
sage: P.<x> = GF(2)[]
sage: f = x*(x+1)
sage: f.gcd(x+1)
x + 1
sage: f.gcd(x^2)
x
```

get_cparent()

is_gen()

EXAMPLES:

```sage
sage: P.<x> = GF(2)[]
sage: x.is_gen()
True
sage: (x+1).is_gen()
False
```
is_one()
EXAMPLES:

```
sage: P.<x> = GF(2)[]
sage: P(1).is_one()
True
```

is_zero()
EXAMPLES:

```
sage: P.<x> = GF(2)[]
sage: x.is_zero()
False
```

list(
copy=True)
EXAMPLES:

```
sage: P.<x> = GF(2)[]
sage: x.list()
[0, 1]
sage: list(x)
[0, 1]
```

quo_rem(right)
EXAMPLES:

```
sage: P.<x> = GF(2)[]
sage: f = x^2 + x + 1
sage: f.quo_rem(x + 1)
(x, 1)
```

shift(n)
EXAMPLES:

```
sage: P.<x> = GF(2)[]
sage: f = x^3 + x^2 + 1
sage: f.shift(1)
x^4 + x^3 + x
sage: f.shift(-1)
x^2 + x
```

truncate(n)
Returns this polynomial mod $x^n$.
EXAMPLES:

```
sage: R.<x> =GF(2)[]
sage: f = sum(x^n for n in range(10)); f
x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1
sage: f.truncate(6)
x^5 + x^4 + x^3 + x^2 + x + 1
```

If the precision is higher than the degree of the polynomial then the polynomial itself is returned:

```
sage: f.truncate(10) is f
True
```
If the precision is negative, the zero polynomial is returned:

```
sage: f.truncate(-1)
```

\texttt{xgcd(other)}

Computes extended gcd of self and other.

\textbf{EXAMPLES:}

```
sage: P.<x> = GF(7)[]
sage: f = x*(x+1)
sage: f.xgcd(x+1)
(x + 1, 0, 1)
sage: f.xgcd(x^2)
(x, 1, 6)
```

\texttt{class sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint}

\texttt{Bases: sage.rings.polynomial.polynomial_zmod_flint.Polynomial_template}

Polynomial on $\mathbb{Z}/n\mathbb{Z}$ implemented via FLINT.

\texttt{__add__(right)}

\textbf{EXAMPLES:}

```
sage: P.<x> = GF(2)[]
sage: x + 1
x + 1
```

\texttt{__sub__(right)}

\textbf{EXAMPLES:}

```
sage: P.<x> = GF(2)[]
sage: x - 1
x + 1
```

\texttt{__lmul__(left)}

\textbf{EXAMPLES:}

```
sage: P.<x> = GF(2)[]
sage: t = x^2 + x + 1
sage: 0*t
0
sage: 1*t
x^2 + x + 1
sage: R.<y> = GF(5)[]
sage: u = y^2 + y + 1
sage: 3*u
3*y^2 + 3*y + 3
sage: 5*u
0
sage: (2^81)*u
2*y^2 + 2*y + 2
sage: (-2^81)*u
3*y^2 + 3*y + 3
```
\begin{align*}
\text{sage: } P.\langle x \rangle &= \text{GF}(2)[] \\
\text{sage: } t &= x^2 + x + 1 \\
\text{sage: } t^0 &= 0 \\
\text{sage: } t^1 &= x^2 + x + 1 \\
\text{sage: } R.\langle y \rangle &= \text{GF}(5)[] \\
\text{sage: } u &= y^2 + y + 1 \\
\text{sage: } u^3 &= 3y^2 + 3y + 3 \\
\text{sage: } u^5 &= 0
\end{align*}

\texttt{\_rmul\_} (\texttt{right})
Multiply self on the right by a scalar.

\textbf{EXAMPLES:}
\begin{align*}
\text{sage: } R.\langle x \rangle &= \mathbb{Z}[x] \\
\text{sage: } f &= (x^3 + x + 5) \\
\text{sage: } f.\_\text{rmul\_}(7) &= 7x^3 + 7x + 35 \\
\text{sage: } f^7 &= 7x^3 + 7x + 35
\end{align*}

\texttt{\_mul\_} (\texttt{right})

\textbf{EXAMPLES:}
\begin{align*}
\text{sage: } P.\langle x \rangle &= \text{GF}(2)[] \\
\text{sage: } x^2(x+1) &= x^2 + x
\end{align*}

\texttt{\_mul\_trunc\_} (\texttt{right}, \texttt{n})
Return the product of this polynomial and other truncated to the given length \(n\).

This function is usually more efficient than simply doing the multiplication and then truncating. The function is tuned for length \(n\) about half the length of a full product.

\textbf{EXAMPLES:}
\begin{align*}
\text{sage: } P.\langle a \rangle &\text{= GF}(7)[] \\
\text{sage: } a &= P(\text{range}(10)); b = P(\text{range}(5, 15)) \\
\text{sage: } a.\_\text{mul\_trunc\_}(b, 5) &= 4a^4 + 6a^3 + 2a^2 + 5a
\end{align*}

\texttt{factor()}
Returns the factorization of the polynomial.

\textbf{EXAMPLES:}
\begin{align*}
\text{sage: } R.\langle x \rangle &= \text{GF}(5)[x] \\
\text{sage: } (x^2 + 1)\_.\text{factor}() &= (x + 2) \ast (x + 3)
\end{align*}
is_irreducible()

Return whether this polynomial is irreducible.

EXAMPLES:

```
sage: R.<x> = GF(5)[]
sage: (x^2 + 1).is_irreducible()
False
sage: (x^3 + x + 1).is_irreducible()
True
```

Not implemented when the base ring is not a field:

```
sage: S.<s> = Zmod(10)[]
sage: (s^2).is_irreducible()
Traceback (most recent call last):
...  
NotImplementedError: checking irreducibility of polynomials over rings with composite characteristic is not implemented
```

monic()

Return this polynomial divided by its leading coefficient.

Raises ValueError if the leading coefficient is not invertible in the base ring.

EXAMPLES:

```
sage: R.<x> = GF(5)[]
sage: (2*x^2+1).monic()
x^2 + 3
```

rational_reconstruct(m, n_deg=0, d_deg=0)

Construct a rational function n/d such that \( p \cdot d \) is equivalent to \( n \) modulo \( m \) where \( p \) is this polynomial.

EXAMPLES:

```
sage: P.<x> = GF(5)[]
sage: p = 4*x^5 + 3*x^4 + 2*x^3 + 2*x^2 + 4*x + 2
sage: n, d = p.rational_reconstruct(x^9, 4, 4); n, d
(3*x^4 + 2*x^3 + x^2 + 2*x, x^4 + 3*x^3 + x^2 + x)
sage: (p*d % x^9) == n
True
```

resultant(other)

Returns the resultant of self and other, which must lie in the same polynomial ring.

INPUT:

- other – a polynomial

OUTPUT: an element of the base ring of the polynomial ring

EXAMPLES:

```
sage: R.<x> = GF(19)[]
sage: f = x^3 + x + 1; g = x^3 - x - 1
sage: r = f.resultant(g); r
11
```

(continues on next page)
sage: r.parent()  is GF(19)
True

The following example shows that trac ticket #11782 has been fixed:

```
sage: R, x = ZZ.quo(9)["x"]
sage: f = 2*x^3 + x^2 + x;  g = 6*x^2 + 2*x + 1
sage: f.resultant(g)
5
```

**reverse**(degree=None)

Return a polynomial with the coefficients of this polynomial reversed.

If an optional degree argument is given the coefficient list will be truncated or zero padded as necessary before computing the reverse.

**EXAMPLES:**

```
sage: R.<x> = GF(5)[]
sage: p = R([1,2,3,4]); p
4*x^3 + 3*x^2 + 2*x + 1
sage: p.reverse()
x^3 + 2*x^2 + 3*x + 4
sage: p.reverse(degree=6)
x^6 + 2*x^5 + 3*x^4 + 4*x^3
sage: p.reverse(degree=2)
x^2 + 2*x + 3
sage: R.<x> = GF(101)[]
sage: f = x^3 - x + 2; f
x^3 + 100*x + 2
sage: f.reverse()
2*x^3 + 100*x^2 + 1
sage: f.reverse() == f(1/x) * x^f.degree()
True
```

Note that if $f$ has zero constant coefficient, its reverse will have lower degree.

```
sage: f = x^3 + 2*x
sage: f.reverse()
2*x^2 + 1
```

In this case, reverse is not an involution unless we explicitly specify a degree.

```
sage: f
x^3 + 2*x
sage: f.reverse().reverse()
x^2 + 2
sage: f.reverse(5).reverse(5)
x^3 + 2*x
```

**revert_series**(n)

Return a polynomial $f$ such that $f(self(x)) = self(f(x)) = x \bmod x^n$.

**EXAMPLES:**
```python
sage: R.<t> = GF(5)[]
sage: f = t + 2*t^2 - t^3 - 3*t^4
sage: f.revert_series(5)
3*t^4 + 4*t^3 + 3*t^2 + t
sage: f.revert_series(-1)
Traceback (most recent call last):
  ... ValueError: argument n must be a non-negative integer, got -1
sage: g = - t^3 + t^5
sage: g.revert_series(6)
Traceback (most recent call last):
  ... ValueError: self must have constant coefficient 0 and a unit for coefficient t^1
sage: g = t + 2*t^2 - t^3 -3*t^4 + t^5
sage: g.revert_series(6)
Traceback (most recent call last):
  ... ValueError: the integers 1 up to n=5 are required to be invertible over the base field
```

```python
small_roots(*args, **kwds)
See sage.rings.polynomial.polynomial_modn_dense_ntl.small_roots() for the documentation of this function.

EXAMPLES:
```
sage: N = 10001
sage: K = Zmod(10001)
sage: P.<x> = PolynomialRing(K)
sage: f = x^3 + 10*x^2 + 5000*x - 222
sage: f.small_roots()
[4]
```

```python
squarefree_decomposition()
Returns the squarefree decomposition of this polynomial.

EXAMPLES:
```
sage: R.<x> = GF(5)[]
sage: ((x+1)*(x^2+1)^2*x^3).squarefree_decomposition()
(x + 1) * (x^2 + 1)^2 * x^3
```
```
2.1.11 Dense univariate polynomials over $\mathbb{Z}/n\mathbb{Z}$, implemented using NTL

This implementation is generally slower than the FLINT implementation in `polynomial_zmod_flint`, so we use FLINT by default when the modulus is small enough; but NTL does not require that $n$ be int-sized, so we use it as default when $n$ is too large for FLINT.

Note that the classes `Polynomial_dense_modn_ntl_zz` and `Polynomial_dense_modn_ntl_ZZ` are different; the former is limited to moduli less than a certain bound, while the latter supports arbitrarily large moduli.

AUTHORS:

- Robert Bradshaw: Split off from `polynomial_element_generic.py` (2007-09)
- Robert Bradshaw: Major rewrite to use NTL directly (2007-09)

class sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_mod_n
Bases: sage.rings.polynomial.polynomial_element.Polynomial

A dense polynomial over the integers modulo $n$, where $n$ is composite, with the underlying arithmetic done using NTL.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(Integers(16), implementation='NTL')
sage: f = x^3 - x + 17
sage: f^2
x^6 + 14*x^4 + 2*x^3 + x^2 + 14*x + 1
sage: loads(f.dumps()) == f
True
sage: R.<x> = PolynomialRing(Integers(100), implementation='NTL')
sage: p = 3*x
sage: q = 7*x
sage: p+q
10*x
sage: R.<x> = PolynomialRing(Integers(8), implementation='NTL')
sage: parent(p)
Univariate Polynomial Ring in x over Ring of integers modulo 100 (using NTL)
sage: p + q
10*x
sage: R({10:-1})
7*x^10
```

def degree(gen=None)
Return the degree of this polynomial.

The zero polynomial has degree -1.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(Integers(100), implementation='NTL')
sage: (x^3 + 3*x - 17).degree()
3
sage: R.zero().degree()
-1
```

int_list()
list\(\text{(copy=\text{True})}\)
Return a new copy of the list of the underlying elements of self.

EXAMPLES:

```
sage: _.<x> = PolynomialRing(Integers(100), implementation='NTL')
sage: f = x^3 + 3*x - 17
sage: f.list()
[83, 3, 0, 1]
```

\texttt{ntl\_\text{\texttt{ZZ}}\_\text{\texttt{p}}\text{\texttt{X}}()}
Return underlying NTL representation of this polynomial. Additional "\text{\texttt{bonus}}" functionality is available through this function.

\textbf{Warning:} You must call \texttt{ntl.set_modulus(ntl.ZZ(n))} before doing arithmetic with this object!

\texttt{ntl\_set\_directly\(\text{(v)}\)}
Set the value of this polynomial directly from a vector or string.

Polynomials over the integers modulo n are stored internally using NTL's \texttt{ZZ\_pX} class. Use this function to set the value of this polynomial using the NTL constructor, which is potentially \texttt{very fast}. The input v is either a vector of ints or a string of the form \([\ n1\ n2\ n3\ \ldots\ ]\) where the ni are integers and there are no commas between them. The optimal input format is the string format, since that's what NTL uses by default.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(Integers(12345678901234567890), implementation='NTL')
sage: from sage.rings.polynomial.polynomial_modn_dense_ntl import Polynomial_‐\rightarrow\text{\texttt{dense\_mod\_n}}\text{\texttt{as}\ poly\_modn\_dense}
sage: poly_modn_dense(R, ([1,-2,3]))
3*x^2 + 98*x + 1
sage: f = poly_modn_dense(R, 0)
sage: f.ntl_set_directly([1,-2,3])
sage: f
3*x^2 + 98*x + 1
sage: f.ntl_set_directly('[1 -2 3 4]')
sage: f
4*x^3 + 3*x^2 + 98*x + 1
```

\texttt{quo\_rem\(\text{(right)}\)}
Returns a tuple (quotient, remainder) where self = quotient*other + remainder.

\texttt{shift\(\(n\))}
Returns this polynomial multiplied by the power \(x^n\). If \(n\) is negative, terms below \(x^n\) will be discarded.

Does not change this polynomial.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(Integers(12345678901234567890), implementation='NTL')
sage: p = x^2 + 2*x + 4
sage: p.shift(0)
x^2 + 2*x + 4
sage: p.shift(-1)
(continues on next page)```
AUTHOR:

- David Harvey (2006-08-06)

small_roots(*args, **kwds)

See sage.rings.polynomial.polynomial_modn_dense_ntl.small_roots() for the documentation of this function.

EXAMPLES:

```python
sage: N = 10001
sage: K = Zmod(10001)
sage: P.<x> = PolynomialRing(K, implementation='NTL')
sage: f = x^3 + 10*x^2 + 5000*x - 222
sage: f.small_roots()
[4]
```

class sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_mod_p

Bases: sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_mod_n

A dense polynomial over the integers modulo p, where p is prime.

discriminant()

EXAMPLES:

```python
sage: _.<x> = PolynomialRing(GF(19),implementation='NTL')
sage: f = x^3 + 3*x - 17
sage: f.discriminant()
12
```

gcd(right)

Return the greatest common divisor of this polynomial and other, as a monic polynomial.

INPUT:

- other – a polynomial defined over the same ring as self

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(GF(3),implementation="NTL")
sage: f,g = x + 2, x^2 - 1
sage: f.gcd(g)
x + 2
```

resultant(other)

Returns the resultant of self and other, which must lie in the same polynomial ring.

INPUT:

- other – a polynomial

OUTPUT: an element of the base ring of the polynomial ring
EXAMPLES:

```python
sage: R.<x> = PolynomialRing(GF(19),implementation='NTL')
sage: f = x^3 + x + 1; g = x^3 - x - 1
sage: r = f.resultant(g); r
11
sage: r.parent() is GF(19)
True
```

xgcd(other)

Compute the extended gcd of this element and other.

INPUT:

• other – an element in the same polynomial ring

OUTPUT:

A tuple \((r, s, t)\) of elements in the polynomial ring such that \(r = s*\text{self} + t*\text{other}\).

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(GF(3),implementation='NTL')
sage: x.xgcd(x)
(x, 0, 1)
sage: (x^2 - 1).xgcd(x - 1)
(x + 2, 0, 1)
sage: R.zero().xgcd(R.one())
(1, 0, 1)
sage: (x^3 - 1).xgcd((x - 1)^2)
(x^2 + x + 1, 0, 1)
sage: ((x - 1)*(x + 1)).xgcd(x*(x - 1))
(x + 2, 1, 2)
```

class sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_modn_ntl_ZZ

Bases: sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_mod_n

degree()

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(Integers(14^34), implementation='NTL')
sage: f = x^4 - x - 1
sage: f.degree()
4
sage: f = 14^43*x + 1
sage: f.degree()
0
```

is_gen()

list(copy=True)

quo_rem(right)

Returns \(q\) and \(r\), with the degree of \(r\) less than the degree of \(right\), such that \(q*right + r = self\).

EXAMPLES:
```python
sage: R.<x> = PolynomialRing(Integers(10^30), implementation='NTL')
sage: f = x^5+1; g = (x+1)^2
sage: q, r = f.quo_rem(g)
sage: q
x^3 + 999999999999999999999999999998*x^2 + 3*x + 999999999999999999999999999996
sage: r
5*x + 5
sage: q*g + r
x^5 + 1
```

**reverse**(degree=None)

Return the reverse of the input polynomial thought as a polynomial of degree degree.

If \( f \) is a degree-\( d \) polynomial, its reverse is \( x^d f(1/x) \).

**INPUT:**

- degree (None or an integer) - if specified, truncate or zero pad the list of coefficients to this degree before reversing it.

**EXAMPLES:**

```python
sage: R.<x> = PolynomialRing(Integers(12^29), implementation='NTL')
sage: f = x^4 + 2*x + 5
sage: f.reverse()
5*x^4 + 2*x^3 + 1
sage: f = x^3 + x
sage: f.reverse()
x^2 + 1
sage: f.reverse(1)
1
sage: f.reverse(5)
x^4 + x^2
```

**shift**(\( n \))

Shift self to left by \( n \), which is multiplication by \( x^n \), truncating if \( n \) is negative.

**EXAMPLES:**

```python
sage: R.<x> = PolynomialRing(Integers(12^30), implementation='NTL')
sage: f = x^7 + x + 1
sage: f.shift(1)
x^8 + x^2 + x
sage: f.shift(-1)
x^6 + 1
sage: f.shift(10).shift(-10) == f
True
```

**truncate**(\( n \))

Returns this polynomial mod \( x^n \).

**EXAMPLES:**

```python
sage: R.<x> = PolynomialRing(Integers(15^30), implementation='NTL')
sage: f = sum(x^n for n in range(10)); f
x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1
```
sage: f.truncate(6)
x^5 + x^4 + x^3 + x^2 + x + 1

valuation()
Returns the valuation of self, that is, the power of the lowest non-zero monomial of self.

EXAMPLES:

sage: R.<x> = PolynomialRing(Integers(10^50), implementation='NTL')
sage: x.valuation()
1
sage: f = x-3; f.valuation()
0
sage: f = x^99; f.valuation()
99
sage: f = x-x; f.valuation()
+Infinity

class sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_modn_ntl_zz
Bases: sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_mod_n

Polynomial on \( \mathbb{Z}/n\mathbb{Z} \) implemented via NTL.

_add_(_right)
_sub_(_right)
_lmul_(c)
_rmul_(c)
_mul_(_right)
_mul_trunc_ (right, n)

Return the product of self and right truncated to the given length n

EXAMPLES:

sage: R.<x> = PolynomialRing(Integers(100), implementation="NTL")
sage: f = x - 2
sage: g = x^2 - 8*x + 16
sage: f*g
x^3 + 90*x^2 + 32*x + 68
sage: f._mul_trunc_(g, 42)
x^3 + 90*x^2 + 32*x + 68
sage: f._mul_trunc_(g, 3)
90*x^2 + 32*x + 68
sage: f._mul_trunc_(g, 2)
32*x + 68
sage: f._mul_trunc_(g, 1)
68
sage: f._mul_trunc_(g, 0)
0
sage: f = x^2 - 8*x + 16
sage: f._mul_trunc_(f, 2)
44*x + 56
degree()
EXAMPLES:

```
sage: R.<x> = PolynomialRing(Integers(77), implementation='NTL')
sage: f = x^4 - x - 1
sage: f.degree()
4
sage: f = 77*x + 1
sage: f.degree()
0
```

int_list()
Returns the coefficients of self as efficiently as possible as a list of python ints.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(Integers(100), implementation='NTL')
sage: from sage.rings.polynomial.polynomial_modn_dense_ntl import Polynomial_˓→dense_mod_n as poly_modn_dense
sage: f = poly_modn_dense(R,[5,0,0,1])
sage: f.int_list()
[5, 0, 0, 1]
sage: [type(a) for a in f.int_list()]
[<... 'int'>, <... 'int'>, <... 'int'>, <... 'int'>]
```

is_gen()

ntl_set_directly(v)

quo_rem(right)
Returns q and r, with the degree of r less than the degree of right, such that q * right + r = self.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(Integers(125), implementation='NTL')
sage: f = x^5+1; g = (x+1)^2
sage: q, r = f.quo_rem(g)
sage: q
x^3 + 123*x^2 + 3*x + 121
sage: r
5*x + 5
sage: q*g + r
x^5 + 1
```

reverse(degree=None)
Return the reverse of the input polynomial thought as a polynomial of degree degree.

If f is a degree-d polynomial, its reverse is \( x^d f(1/x) \).

INPUT:

- degree (None or an integer) - if specified, truncate or zero pad the list of coefficients to this degree before reversing it.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(Integers(77), implementation='NTL')
sage: f = x^4 - x - 1
```
sage: f.reverse()
76*x^4 + 76*x^3 + 1
sage: f.reverse(2)
76*x^2 + 76*x
sage: f.reverse(5)
76*x^5 + 76*x^4 + x
sage: g = x^3 - x
sage: g.reverse()
76*x^2 + 1

shift(n)
Shift self to left by n, which is multiplication by x^n, truncating if n is negative.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(Integers(77), implementation='NTL')
sage: f = x^7 + x + 1
sage: f.shift(1)
x^8 + x^2 + x
sage: f.shift(-1)
x^6 + 1
sage: f.shift(10).shift(-10) == f
True
```

truncate(n)
Returns this polynomial mod x^n.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(Integers(77), implementation='NTL')
sage: f = sum(x^n for n in range(10)); f
x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1
sage: f.truncate(6)
x^5 + x^4 + x^3 + x^2 + x + 1
```

valuation()
Returns the valuation of self, that is, the power of the lowest non-zero monomial of self.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(Integers(10), implementation='NTL')
sage: x.valuation()
1
sage: f = x-3; f.valuation()
0
sage: f = x^99; f.valuation()
99
sage: f = x-x; f.valuation()
+Infinity
```

Let N be the characteristic of the base ring this polynomial is defined over: N = self.base_ring().
characteristic(). This method returns small roots of this polynomial modulo some factor \( b \) of \( N \) with the constraint that \( b \geq N^\beta \). Small in this context means that if \( x \) is a root of \( f \) modulo \( b \) then \( |x| < X \). This \( X \) is either provided by the user or the maximum \( X \) is chosen such that this algorithm terminates in polynomial time. If \( X \) is chosen automatically it is \( X = \text{ceil}(1/2N^\beta^2/\delta - \epsilon) \). The algorithm may also return some roots which are larger than \( X \). ‘This algorithm’ in this context means Coppersmith’s algorithm for finding small roots using the LLL algorithm. The implementation of this algorithm follows Alexander May’s PhD thesis referenced below.

**INPUT:**
- \( X \) – an absolute bound for the root (default: see above)
- \( \text{beta} \) – compute a root mod \( b \) where \( b \) is a factor of \( N \) and \( b \geq N^\beta \). (Default: 1.0, so \( b = N \).)
- \( \epsilon \) – the parameter \( \epsilon \) described above. (Default: \( \beta/8 \))
- **\( \text{kwd} \)**s – passed through to method \( \text{Matrix_integers.dense.LLL()} \).

**EXAMPLES:**
First consider a small example:

```python
sage: N = 10001
sage: K = Zmod(10001)
sage: P.<x> = PolynomialRing(K, implementation='NTL')
sage: f = x^3 + 10*x^2 + 5000*x - 222
```

This polynomial has no roots without modular reduction (i.e. over \( \mathbb{Z} \)):

```python
sage: f.change_ring(ZZ).roots()
[]
```

To compute its roots we need to factor the modulus \( N \) and use the Chinese remainder theorem:

```python
sage: p, q = N.prime_divisors()
sage: f.change_ring(GF(p)).roots()
[(4, 1)]
sage: f.change_ring(GF(q)).roots()
[(4, 1)]
sage: crt(4, 4, p, q)
4
```

This root is quite small compared to \( N \), so we can attempt to recover it without factoring \( N \) using Coppersmith’s small root method:

```python
sage: f.small_roots()
[4]
```

An application of this method is to consider RSA. We are using 512-bit RSA with public exponent \( e = 3 \) to encrypt a 56-bit DES key. Because it would be easy to attack this setting if no padding was used we pad the key \( K \) with 1s to get a large number:

```python
sage: Nbits, Kbits = 512, 56
sage: e = 3
```

We choose two primes of size 256-bit each:
We choose a random key:

```
sage: K = ZZ.random_element(0, 2^Kbits)
```

and pad it with 512-56=456 1s:

```
sage: Kdigits = K.digits(2)
sage: M = [0]*Kbits + [1]*(Nbits-Kbits)
sage: for i in range(len(Kdigits)): M[i] = Kdigits[i]
sage: M = ZZ(M, 2)
```

Now we encrypt the resulting message:

```
sage: C = ZmodN(M)^e
```

To recover \(K\) we consider the following polynomial modulo \(N\):

```
sage: P.<x> = PolynomialRing(ZmodN, implementation='NTL')
sage: f = (2^Nbits - 2^Kbits + x)^e - C
```

and recover its small roots:

```
sage: Kbar = f.small_roots()[0]
sage: K == Kbar
True
```

The same algorithm can be used to factor \(N = pq\) if partial knowledge about \(q\) is available. This example is from the Magma handbook:

First, we set up \(p\), \(q\) and \(N\):

```
sage: length = 512
sage: hidden = 110
sage: p = next_prime(2^int(round(length/2)))
sage: q = next_prime( round(pi.n()*p) )
sage: N = p*q
```

Now we disturb the low 110 bits of \(q\):

```
sage: qbar = q + ZZ.random_element(0,2^hidden-1)
```

And try to recover \(q\) from it:

```
sage: F.<x> = PolynomialRing(Zmod(N), implementation='NTL')
sage: f = x - qbar
```

We know that the error is \(\leq 2^{\text{hidden}} - 1\) and that the modulus we are looking for is \(\geq \sqrt{N}\):
sage: from sage.misc.verbose import setVerbose
sage: setVerbose(2)

sage: d = f.small_roots(X=2^hidden-1, beta=0.5)[0]  # time random
verbose 2 (<module>) m = 4
verbose 2 (<module>) t = 4
verbose 2 (<module>) X = 1298074214633706907132624082305023
verbose 1 (<module>) LLL of 8x8 matrix (algorithm fpLLL:wrapper)
verbose 1 (<module>) LLL finished (time = 0.006998)
sage: q == qbar - d
True

REFERENCES:


### 2.1.12 Dense univariate polynomials over \( \mathbb{R} \), implemented using MPFR

```python
class sage.rings.polynomial.polynomial_real_mpfr_dense.PolynomialRealDense
Bases: sage.rings.polynomial.polynomial_element.Polynomial

change_ring(R)
```

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.polynomial_real_mpfr_dense import PolynomialRealDense
sage: f = PolynomialRealDense(RR['x'], [-2, 0, 1.5])
sage: f
3.00000000000000*x^2 + 2.00000000000000*x + 1.00000000000000
sage: f.change_ring(QQ)
3/2*x^2 + 2
sage: f.change_ring(RealField(10))
1.5*x^2 - 2.0
sage: f.change_ring(RealField(100))
1.5000000000000000000000000000*x^2 - 2.0000000000000000000000000000
```

**degree()**

Return the degree of the polynomial.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.polynomial_real_mpfr_dense import PolynomialRealDense
sage: f = PolynomialRealDense(RR['x'], [1, 2, 3]); f
3.00000000000000*x^2 + 2.00000000000000*x + 1.00000000000000
sage: f.degree()
2
```

**integral()**

**EXAMPLES:**
```python
sage: from sage.rings.polynomial.polynomial_real_mpfr_dense import PolynomialRealDense
sage: f = PolynomialRealDense(RR['x'], [3, pi, 1])
sage: f.integral()
0.333333333333333*x^3 + 1.57079632679490*x^2 + 3.00000000000000*x
```

**list** *(copy=True)*

EXAMPLES:

```python
sage: from sage.rings.polynomial.polynomial_real_mpfr_dense import PolynomialRealDense
sage: f = PolynomialRealDense(RR['x'], [1, 0, -2]); f
-2.00000000000000*x^2 + 1.00000000000000
sage: f.list()
[1.00000000000000, 0.000000000000000, -2.00000000000000]
```

**quo_rem**(other)

Return the quotient with remainder of self by other.

EXAMPLES:

```python
sage: from sage.rings.polynomial.polynomial_real_mpfr_dense import PolynomialRealDense
sage: f = PolynomialRealDense(RR['x'], [-2, 0, 1])
sage: g = PolynomialRealDense(RR['x'], [5, 1])
sage: q, r = f.quo_rem(g)
sage: q
x - 5.00000000000000
sage: r
23.0000000000000
sage: q*g + r == f
True
sage: fg = f*g
sage: fg.quo_rem(f)
(x + 5.00000000000000, 0)
sage: fg.quo_rem(g)
(x^2 - 2.00000000000000, 0)
sage: f = PolynomialRealDense(RR['x'], range(5))
sage: g = PolynomialRealDense(RR['x'], [pi,3000,4])
sage: q, r = f.quo_rem(g)
sage: g*q + r == f
True
```

**reverse**(degree=None)

Return reverse of the input polynomial thought as a polynomial of degree degree.

If f is a degree-\(d\) polynomial, its reverse is \(x^d f(1/x)\).

**INPUT:**

- degree (None or an integer) - if specified, truncate or zero pad the list of coefficients to this degree before reversing it.

**EXAMPLES:**
sage: f = RR['x']([-3, pi, 0, 1])
sage: f.reverse()
-3.00000000000000*x^3 + 3.14159265358979*x^2 + 1.00000000000000
sage: f.reverse(2)
-3.00000000000000*x^2 + 3.14159265358979*x
sage: f.reverse(5)
-3.00000000000000*x^5 + 3.14159265358979*x^4 + x^2

**shift** (*n*)

Returns this polynomial multiplied by the power \(x^n\). If \(n\) is negative, terms below \(x^n\) will be discarded. Does not change this polynomial.

**EXAMPLES:**

sage: from sage.rings.polynomial.polynomial_real_mpfr_dense import PolynomialRealDense
sage: f = PolynomialRealDense(RR['x'], [1, 2, 3]); f
3.00000000000000*x^2 + 2.00000000000000*x + 1.00000000000000
sage: f.shift(10)
3.00000000000000*x^12 + 2.00000000000000*x^11 + x^10
sage: f.shift(-1)
3.00000000000000*x + 2.00000000000000
sage: f.shift(-10)
0

**truncate** (*n*)

Returns the polynomial of degree < \(n\) which is equivalent to self modulo \(x^n\).

**EXAMPLES:**

sage: from sage.rings.polynomial.polynomial_real_mpfr_dense import PolynomialRealDense
sage: f = PolynomialRealDense(RealField(10)['x'], [1, 2, 4, 8])
sage: f.truncate(3)
4.0*x^2 + 2.0*x + 1.0
sage: f.truncate(100)
8.0*x^3 + 4.0*x^2 + 2.0*x + 1.0
sage: f.truncate(1)
1.0
sage: f.truncate(0)
0

**truncate_abs** (*bound*)

Truncate all high order coefficients below bound.

**EXAMPLES:**

sage: from sage.rings.polynomial.polynomial_real_mpfr_dense import PolynomialRealDense
sage: f = PolynomialRealDense(RealField(10)['x'], [10^-k for k in range(10)])
sage: f
1.0e-9*x^9 + 1.0e-8*x^8 + 1.0e-7*x^7 + 1.0e-6*x^6 + 0.000010*x^5 + 0.00010*x^4 + 0.0010*x^3 + 0.010*x^2 + 0.10*x + 1.0
sage: f.truncate_abs(0.5e-6)
1.0e-6*x^6 + 0.000010*x^5 + 0.00010*x^4 + 0.0010*x^3 + 0.010*x^2 + 0.10*x + 1.0

(continues on next page)
sage: f.truncate_abs(10.0)
0
sage: f.truncate_abs(1e-100) == f
True

sage.rings.polynomial.polynomial_real_mpfr_dense.make_PolynomialRealDense(parent, data)

EXAMPLES:

sage: from sage.rings.polynomial.polynomial_real_mpfr_dense import make_
˓→PolynomialRealDense
sage: make_PolynomialRealDense(RR['x'], [1,2,3])
3.00000000000000*x^2 + 2.00000000000000*x + 1.00000000000000

2.1.13 Polynomial Interfaces to Singular

AUTHORS:

• Martin Albrecht <malb@informatik.uni-bremen.de> (2006-04-21)
• Robert Bradshaw: Re-factor to avoid multiple inheritance vs. Cython (2007-09)
• Syed Ahmad Lavasani: Added function field to _singular_init_ (2011-12-16) Added non-prime finite fields to _singular_init_ (2012-1-22)

class sage.rings.polynomial.polynomial_singular_interface.PolynomialRing_singular_repr
Bases: object

Implements methods to convert polynomial rings to Singular.

This class is a base class for all univariate and multivariate polynomial rings which support conversion from and to Singular rings.

class sage.rings.polynomial.polynomial_singular_interface.Polynomial_singular_repr
Bases: object

Implements coercion of polynomials to Singular polynomials.

This class is a base class for all (univariate and multivariate) polynomial classes which support conversion from and to Singular polynomials.

Due to the incompatibility of Python extension classes and multiple inheritance, this just defers to module-level functions.

sage.rings.polynomial.polynomial_singular_interface.can_convert_to_singular(R)
Returns True if this ring’s base field or ring can be represented in Singular, and the polynomial ring has at least one generator. If this is True then this polynomial ring can be represented in Singular.

The following base rings are supported: finite fields, rationals, number fields, and real and complex fields.

EXAMPLES:

sage: from sage.rings.polynomial.polynomial_singular_interface import can_convert_˓→to_singular
sage: can_convert_to_singular(PolynomialRing(QQ, names=['x']))
True
sage: can_convert_to_singular(PolynomialRing(ZZ, names=['x']))
True

(continues on next page)
2.1.14 Base class for generic $p$-adic polynomials

This provides common functionality for all $p$-adic polynomials, such as printing and factoring.

AUTHORS:

- Jeroen Demeyer (2013-11-22): initial version, split off from other files, made Polynomial_padic the common base class for all $p$-adic polynomials.

```python
sage: K = Zp(13,7)
sage: R.<t> = K[]
sage: f = 13^7*t^3 + K(169,4)*t - 13^4
sage: f.content()
13^2 + O(13^9)
sage: R(0).content()
0
sage: f = R(K(0,3)); f
0(13^3)
sage: f.content()
0(13^3)

sage: P.<x> = ZZ[]
sage: f = x + 2
sage: f.content()
1
sage: fp = f.change_ring(pAdicRing(2, 10))
sage: fp
(1 + 0(2^10))*x + 2 + 0(2^11)
sage: fp.content()
1 + 0(2^10)
sage: (2*fp).content()
2 + 0(2^11)
```
Over a field it would be sufficient to return only zero or one, as the content is only defined up to multiplication with a unit. However, we return $\pi^k$ where $k$ is the minimal valuation of any coefficient:

```
sage: K = Qp(13,7)
sage: R.<t> = K[]
sage: f = 13^7*t^3 + K(169,4)*t - 13^-4
sage: f.content()
13^-4 + O(13^3)
sage: f = R.zero()
sage: f.content()
0
sage: f = R(K(0,3))
sage: f.content()
O(13^3)
sage: f = 13*t^3 + K(0,1)*t
sage: f.content()
13 + O(13^8)
```

**factor()**

Return the factorization of this polynomial.

**EXAMPLES:**

```
sage: R.<t> = PolynomialRing(Qp(3,3,print_mode='terse',print_pos=False))
sage: pol = t^8 - 1
sage: for p,e in pol.factor():
    ....:     print("{} {}",format(e, p))
1 (1 + O(3^3))*t + 1 + O(3^3)
1 (1 + O(3^3))*t - 1 + O(3^3)
1 (1 + O(3^3))*t^2 + (5 + O(3^3))*t - 1 + O(3^3)
1 (1 + O(3^3))*t^2 + (-5 + O(3^3))*t - 1 + O(3^3)
1 (1 + O(3^3))*t^2 + O(3^3)*t + 1 + O(3^3)
sage: R.<t> = PolynomialRing(Qp(5,6,print_mode='terse',print_pos=False))
sage: pol = 100 * (5*t - 1) * (t - 5)
sage: pol
(500 + O(5^9))*t^2 + (-2600 + O(5^8))*t + 500 + O(5^9)
sage: pol.factor()
(500 + O(5^9)) * ((1 + O(5^5))*t - 1/5 + O(5^5)) * ((1 + O(5^6))*t - 5 + O(5^6))
sage: pol.factor().value()
(500 + O(5^8))*t^2 + (-2600 + O(5^8))*t + 500 + O(5^8)
```

The same factorization over $\mathbb{Z}_p$. In this case, the “unit” part is a $p$-adic unit and the power of $p$ is considered to be a factor:

```
sage: R.<t> = PolynomialRing(Zp(5,6,print_mode='terse',print_pos=False))
sage: pol = 100 * (5*t - 1) * (t - 5)
sage: pol
(500 + O(5^9))*t^2 + (-2600 + O(5^8))*t + 500 + O(5^9)
sage: pol.factor()
(4 + O(5^6)) * (5 + 0(5^7))^2 * ((1 + O(5^6))*t - 5 + O(5^6)) * ((5 + O(5^6))*t -
˓→ 1 + O(5^6))
sage: pol.factor().value()
(500 + O(5^8))*t^2 + (-2600 + O(5^8))*t + 500 + O(5^8)
```

In the following example, the discriminant is zero, so the $p$-adic factorization is not well defined:
An example of factoring a constant polynomial (see trac ticket #26669):

```
sage: R.<x> = Qp(5)
sage: R(2).factor()
2 + O(5^20)
```

More examples over \( \mathbb{Z}_p \): 

```
sage: R.<w> = PolynomialRing(Zp(5, prec=6, type = 'capped-abs', print_mode = 'val-unit'))
sage: f = w^5-1
sage: f.factor()
((1 + O(5^6))*w + 3124 + O(5^6)) * ((1 + O(5^6))*w^4 + (12501 + O(5^6))*w^3 +
(9376 + O(5^6))*w^2 + (6251 + O(5^6))*w + 3126 + O(5^6))
```

See trac ticket #4038:

```
sage: E = EllipticCurve('37a1')
sage: K = Qp(7,10)
sage: EK = E.base_extend(K)
sage: g = EK.division_polynomial_0(3)
sage: g.factor()
(3 + O(7^10)) * ((1 + 0(7^10))*x + 1 + 2*7 + 4*7^2 + 2*7^3 + 5*7^4 + 7*5 + 5*7^6 +
3*7^7 + 5*7^8 + 3*7^9 + 0(7^10)) * ((1 + 0(7^10))*x^3 + (6 + 4*7 + 2*7^2 +
4*7^3 + 7*4 + 5*7^5 + 7*6 + 3*7^7 + 7*8 + 3*7^9 + 0(7^10))*x^2 + (6 + 3*7 +
5*7^2 + 2*7^4 + 7*5 + 7*6 + 2*7^8 + 3*7^9 + 0(7^10))*x + 2 + 5*7 + 4*7^2 +
2*7^3 + 6*7^4 + 3*7^5 + 7*6 + 4*7^7 + 0(7^10))
```

**root_field**(names, check_irreducible=True, **kwds)

Return the \( p \)-adic extension field generated by the roots of the irreducible polynomial self.

INPUT:

- **names** – name of the generator of the extension
- **check_irreducible** – check whether the polynomial is irreducible
- **kwds** – see sage.ring.padics.padic_generic.pAdicGeneric.extension()

EXAMPLES:

```
sage: E.<x> = Qp(3,print_mode='digits')
sage: f = x^2 - 3
sage: f.root_field('x')
3-adic Eisenstein Extension Field in x defined by x^2 - 3
```
```python
sage: R.<x> = Qp(5,5,print_mode='digits')[]
sage: f = x^2 - 3
sage: f.root_field('x', print_mode='bars')
5-adic Unramified Extension Field in x defined by x^2 - 3

sage: R.<x> = Qp(11,5,print_mode='digits')[]
sage: f = x^2 - 3
sage: f.root_field('x', print_mode='bars')
Traceback (most recent call last):
  ... ValueError: polynomial must be irreducible
```

### 2.1.15 p-adic Capped Relative Dense Polynomials

class sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense

Bases: sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_cdv, sage.rings.polynomial.padics.polynomial_padic.Polynomial_padic

**degree**(secure=False)

Return the degree of self.

**INPUT:**

- secure – a boolean (default: False)

If secure is True and the degree of this polynomial is not determined (because the leading coefficient is indistinguishable from 0), an error is raised.

If secure is False, the returned value is the largest n so that the coefficient of \(x^n\) does not compare equal to 0.

**EXAMPLES:**

```python
sage: K = Qp(3,10)
sage: R.<T> = K[]
sage: f = T + 2; f
(1 + O(3^10))*T + 2 + O(3^10)
sage: f.degree()
1
```

(continues on next page)
sage: (f-T).degree()
0
sage: (f-T).degree(secure=True)
Traceback (most recent call last):
  ...  PrecisionError: the leading coefficient is indistinguishable from 0

sage: x = O(3^5)
sage: li = [3^i * x for i in range(0,5)]; li
[O(3^5), O(3^6), O(3^7), O(3^8), O(3^9)]
sage: f = R(li); f
O(3^9)*T^4 + O(3^8)*T^3 + O(3^7)*T^2 + O(3^6)*T + O(3^5)
sage: f.degree()
-1
sage: f.degree(secure=True)
Traceback (most recent call last):
  ...  PrecisionError: the leading coefficient is indistinguishable from 0

disc()

factor_mod()
Return the factorization of self modulo \( p \).

is_eisenstein(secure=False)
Return True if this polynomial is an Eisenstein polynomial.

EXAMPLES:

sage: K = Qp(5)
sage: R.<t> = K[
]
sage: f = 5 + 5*t + t^4
sage: f.is_eisenstein()
True

AUTHOR:
• Xavier Caruso (2013-03)

lift()
Return an integer polynomial congruent to this one modulo the precision of each coefficient.

Note: The lift that is returned will not necessarily be the same for polynomials with the same coefficients (i.e. same values and precisions): it will depend on how the polynomials are created.

EXAMPLES:

sage: K = Qp(13,7)
sage: R.<t> = K[
]
sage: a = 13^7*t^3 + K(169,4)*t - 13^4
sage: a.lift()
62748517*t^3 + 169*t - 28561

list(copy=True)
Return a list of coefficients of self.
Note: The length of the list returned may be greater than expected since it includes any leading zeros that have finite absolute precision.

EXAMPLES:

```
sage: K = Qp(13, 7)
sage: R.<t> = K[]
sage: a = 2*t^3 + 169*t - 1
sage: a
(2 + O(13^7))*t^3 + (13^2 + O(13^9))*t + 12 + 12*13 + 12*13^2 + 12*13^3 + 12*13^4 + 12*13^5 + 12*13^6 + O(13^7)
sage: a.list()
[12 + 12*13 + 12*13^2 + 12*13^3 + 12*13^4 + 12*13^5 + 12*13^6 + O(13^7),
  13^2 + 0(13^9),
  2 + O(13^7)]
```

\texttt{lshift_coeffs}(\textit{shift}, \textit{no_list}=False)

Return a new polynomials whose coefficients are multiplied by \( p^\text{shift} \).

EXAMPLES:

```
sage: K = Qp(13, 4)
sage: R.<t> = K[]
sage: a = t + 52
sage: a.lshift_coeffs(3)
(13^3 + O(13^7))*t + 4*13^4 + O(13^8)
```

\texttt{newton_polygon}()

Return the Newton polygon of this polynomial.

\textbf{Note:} If some coefficients have not enough precision an error is raised.

\textbf{OUTPUT:}

- a Newton polygon

EXAMPLES:

```
sage: K = Qp(2, prec=5)
sage: P.<x> = K[]
sage: f = x^4 + 2^3*x^3 + 2^13*x^2 + 2^21*x + 2^37
sage: f.newton_polygon()
Finite Newton polygon with 4 vertices: (0, 37), (1, 21), (3, 3), (4, 0)

sage: K = Qp(5)
sage: R.<t> = K[]
sage: f = 5 + 3*t + t^4 + 25*t^10
sage: f.newton_polygon()
Finite Newton polygon with 4 vertices: (0, 1), (1, 0), (4, 0), (10, 2)
```

Here is an example where the computation fails because precision is not sufficient:

```python
sage: g = f + K(0,0)*t^4; g
(5^2 + O(5^22))*t^10 + O(5^0)*t^4 + (3 + O(5^20))*t + 5 + O(5^21)
sage: g.newton_polygon()
Traceback (most recent call last):
... PrecisionError: The coefficient of t^4 has not enough precision
```

**AUTHOR:**
- Xavier Caruso (2013-03-20)

`newton_slopes(repetition=True)`

Return a list of the Newton slopes of this polynomial.

These are the valuations of the roots of this polynomial.

If `repetition` is True, each slope is repeated a number of times equal to its multiplicity. Otherwise it appears only one time.

**INPUT:**
- `repetition` – boolean (default True)

**OUTPUT:**
- a list of rationals

**EXAMPLES:**

```python
sage: K = Qp(5)
sage: R.<t> = K[]
sage: f = 5 + 3*t + t^4 + 25*t^10
sage: f.newton_polygon()
Finite Newton polygon with 4 vertices: (0, 1), (1, 0), (4, 0), (10, 2)
sage: f.newton_slopes()
[1, 0, 0, 0, -1/3, -1/3, -1/3, -1/3, -1/3, -1/3]
sage: f.newton_slopes(repetition=False)
[1, 0, -1/3]
```

**AUTHOR:**
- Xavier Caruso (2013-03-20)

`prec_degree()`

Return the largest $n$ so that precision information is stored about the coefficient of $x^n$.

Always greater than or equal to degree.

**EXAMPLES:**

```python
sage: K = Qp(3,10)
sage: R.<T> = K[]
sage: f = T + 2; f
(1 + O(3^10))*T + 2 + O(3^10)
sage: f.prec_degree()
1
```
**precision_absolute**(n=None)
Return absolute precision information about self.

**INPUT:**
- **self** – a p-adic polynomial
- **n** – None or an integer (default None).

**OUTPUT:**
- If n == None, returns a list of absolute precisions of coefficients. Otherwise, returns the absolute precision of the coefficient of x^n.

**EXAMPLES:**
```
sage: K = Qp(3,10)
sage: R.<T> = K[]
sage: f = T + 2; f
(1 + O(3^10))*T + 2 + O(3^10)
sage: f.precision_absolute()
[10, 10]
```

**precision_relative**(n=None)
Return relative precision information about self.

**INPUT:**
- **self** – a p-adic polynomial
- **n** – None or an integer (default None).

**OUTPUT:**
- If n == None, returns a list of relative precisions of coefficients. Otherwise, returns the relative precision of the coefficient of x^n.

**EXAMPLES:**
```
sage: K = Qp(3,10)
sage: R.<T> = K[]
sage: f = T + 2; f
(1 + O(3^10))*T + 2 + O(3^10)
sage: f.precision_relative()
[10, 10]
```

**quo_rem**(right, secure=False)
Return the quotient and remainder in division of self by right.

**EXAMPLES:**
```
sage: K = Qp(3,10)
sage: R.<T> = K[]
sage: f = T + 2
sage: g = T^4 + 3*T+22
sage: g.quo_rem(f)
((1 + O(3^10))*T^3 + (1 + 2*3 + 2*3^2 + 2*3^3 + 2*3^4 + 2*3^5 + 2*3^6 + 2*3^7 +
  ... + 2*3^8 + 2*3^9 + O(3^10))*T^2 + (1 + 3 + O(3^10))*T + 1 + 3 + 2*3^2 + 2*3^3 +
  ... + 2*3^4 + 2*3^5 + 2*3^6 + 2*3^7 + 2*3^8 + 2*3^9 + O(3^10),
  2 + 3 + 3*3 + 0(3^10))
```
rescale\((a)\)
Return \(f(a \times X)\)

Todo: Need to write this function for integer polynomials before this works.

EXAMPLES:

```sage
sage: K = Zp(13, 5)
sage: R.<t> = K[]
sage: f = t^3 + K(13, 3) * t
sage: f.rescale(2)  # not implemented
```

reverse\((\text{degree} = \text{None})\)
Return the reverse of the input polynomial, thought as a polynomial of degree \(\text{degree}\).

If \(f\) is a degree-\(d\) polynomial, its reverse is \(x^d f(1/x)\).

INPUT:

- \(\text{degree} \) (\text{None} or an integer) - if specified, truncate or zero pad the list of coefficients to this degree before reversing it.

EXAMPLES:

```sage
sage: K = Qp(13,7)
sage: R.<t> = K[]
sage: f = t^3 + 4*t; f
(1 + O(13^7))*t^3 + (4 + O(13^7))*t
sage: f.reverse()
0*t^3 + (4 + O(13^7))*t^2 + 1 + O(13^7)
sage: f.reverse(3)
0*t^3 + (4 + O(13^7))*t^2 + 1 + O(13^7)
sage: f.reverse(2)
0*t^2 + (4 + O(13^7))*t
sage: f.reverse(4)
0*t^4 + (4 + O(13^7))*t^3 + (1 + O(13^7))*t
sage: f.reverse(6)
0*t^6 + (4 + O(13^7))*t^5 + (1 + O(13^7))*t^3
```

rshift_coeffs\((\text{shift}, \text{no_list}=\text{False})\)
Return a new polynomial whose coefficients are \(p\)-adically shifted to the right by \(\text{shift}\).

Note: Type \(\text{Qp}(5)(0).\_\_\_rshift\_\_?\) for more information.

EXAMPLES:

```sage
sage: K = Zp(13, 4)
sage: R.<t> = K[]
sage: a = t^2 + K(13,3)*t + 169; a
(1 + 0(13^4))*t^2 + (13 + 0(13^3))*t + 13^2 + O(13^6)
sage: b = a.rshift_coeffs(1); b
0(13^3)*t^2 + (1 + 0(13^2))*t + 13 + 0(13^5)
sage: b.list()
[13 + 0(13^5), 1 + 0(13^2), 0(13^3)]
```
sage: b = a.rshift_coeffs(2); b
O(13^2)*t^2 + O(13)*t + 1 + O(13^4)
sage: b.list()
[1 + O(13^4), 0(13), 0(13^2)]

valuation(val_of_var=None)

Return the valuation of self.

INPUT:

self – a p-adic polynomial
val_of_var – None or a rational (default None).

OUTPUT:

If val_of_var == None, returns the largest power of the variable dividing self. Otherwise, returns the valuation of self where the variable is assigned valuation val_of_var.

EXAMPLES:

sage: K = Qp(3,10)
sage: R.<T> = K[]
sage: f = T + 2; f
(1 + O(3^10))*T + 2 + O(3^10)
sage: f.valuation()
0

valuation_of_coefficient(n=None)

Return valuation information about self’s coefficients.

INPUT:

self – a p-adic polynomial
n – None or an integer (default None).

OUTPUT:

If n == None, returns a list of valuations of coefficients. Otherwise, returns the valuation of the coefficient of x^n.

EXAMPLES:

sage: K = Qp(3,10)
sage: R.<T> = K[]
sage: f = T + 2; f
(1 + O(3^10))*T + 2 + O(3^10)
sage: f.valuation_of_coefficient(1)
0

sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.make_padic_poly(parent, x, version)
2.1.16 p-adic Flat Polynomials

class sage.rings.polynomial.padics.polynomial_padic_flat.Polynomial_padic_flat(parent, x=None, check=True, is_gen=False, construct=False, absprec=None)

Bases: sage.rings.polynomial.polynomial_element.Polynomial_generic_dense, sage.rings.polynomial.padics.polynomial_padic.Polynomial_padic

2.1.17 Univariate Polynomials over GF(p^e) via NTL's ZZ_pEX

AUTHOR:
• Yann Laigle-Chapuy (2010-01) initial implementation

class sage.rings.polynomial.polynomial_zz_pex.Polynomial_ZZ_pEX

Bases: sage.rings.polynomial.polynomial_zz_pex.Polynomial_template

Univariate Polynomials over GF(p^n) via NTL's ZZ_pEX.

EXAMPLES:

```
sage: K.<a>=GF(next_prime(2**60)**3)
sage: R.<x> = PolynomialRing(K,implementation='NTL')
sage: (x^3 + a*x^2 + 1) * (x + a)
x^4 + 2*a*x^3 + a^2*x^2 + x + a
```

```
is_irreducible(algorithm='fast_when_false', iter=1)
```

Returns True precisely when self is irreducible over its base ring.

INPUT:

Parameters

• algorithm – a string (default “fast_when_false”), there are 3 available algorithms: “fast_when_true”, “fast_when_false” and “probabilistic”.

• iter – (default: 1) if the algorithm is “probabilistic” defines the number of iterations. The error probability is bounded by \(q^{-iter}\) for polynomials in \(GF(q)[x]\).

EXAMPLES:

```
sage: K.<a>=GF(next_prime(2**60)**3)
sage: R.<x> = PolynomialRing(K,implementation='NTL')
sage: P = x^3+(2-a)*x+1
sage: P.is_irreducible(algorithm="fast_when_false")
True
sage: P.is_irreducible(algorithm="fast_when_true")
True
sage: P.is_irreducible(algorithm="probabilistic")
True
sage: Q = (x^2+a)*(x+a^3)
sage: Q.is_irreducible(algorithm="fast_when_false")
```

(continues on next page)
False
 sage: Q.is_irreducible(algorithm="fast_when_true")
 False
 sage: Q.is_irreducible(algorithm="probabilistic")
 False

list(copy=True)
Return the list of coefficients.

EXAMPLES:

    sage: K.<a> = GF(next_prime(2**60)**3)
sage: P = PolynomialRing(K, 'x')
sage: f = P.random_element(100)
sage: f.list() == [f[i] for i in range(f.degree()+1)]
    True
 sage: P.0.list()
 [0, 1]

resultant(other)
Returns the resultant of self and other, which must lie in the same polynomial ring.

INPUT:

    Parameters other -- a polynomial

OUTPUT: an element of the base ring of the polynomial ring

EXAMPLES:

    sage: K.<a>=GF(next_prime(2**60)**3)
sage: R.<x> = PolynomialRing(K,implementation='NTL')
sage: f=(x-a)*(x-a**2)*(x+1)
sage: g=(x-a**3)*(x-a**4)*(x+a)
sage: r = f.resultant(g)
sage: r == prod(u-v for (u,eu) in f.roots() for (v,ev) in g.roots())
    True

shift(n)
EXAMPLES:

    sage: K.<a> = GF(next_prime(2**60)**3)
sage: R.<x> = PolynomialRing(K,implementation='NTL')
sage: f = x^3 + x^2 + 1
 sage: f.shift(1)
 x^4 + x^3 + x
 sage: f.shift(-1)
 x^2 + x

class sage.rings.polynomial.polynomial_zz_pex.Polynomial_ZZ_pX
     Bases: sage.rings.polynomial.polynomial_zz_pex.Polynomial_template

class sage.rings.polynomial.polynomial_zz_pex.Polynomial_template
     Bases: sage.rings.polynomial.polynomial_element.Polynomial

Template for interfacing to external C / C++ libraries for implementations of polynomials.

2.1. Univariate Polynomials and Polynomial Rings 187
AUTHORS:

- Robert Bradshaw (2008-10): original idea for templating
- Martin Albrecht (2008-10): initial implementation

This file implements a simple templating engine for linking univariate polynomials to their C/C++ library implementations. It requires a ‘linkage’ file which implements the `elem_` functions (see `sage.libsntl.ntl_GF2X_linkage` for an example). Both parts are then plugged together by inclusion of the linkage file when inheriting from this class. See `sage.rings.polynomial.polynomial_gf2x` for an example.

We illustrate the generic glueing using univariate polynomials over $\mathbb{GF}(2)$.

**Note:** Implementations using this template MUST implement coercion from base ring elements and `get_unsafe()`. See `Polynomial_GF2X` for an example.

### degree()

**EXAMPLES:**

```
 sage: P.<x> = GF(2)[]
 sage: x.degree()
 1
 sage: P(1).degree()
 0
 sage: P(0).degree()
 -1
```

### gcd(other)

Return the greatest common divisor of self and other.

**EXAMPLES:**

```
 sage: P.<x> = GF(2)[]
 sage: f = x*(x+1)
 sage: f.gcd(x+1)
 x + 1
 sage: f.gcd(x^2)
 x
```

### get_cparent()

### is_gen()

**EXAMPLES:**

```
 sage: P.<x> = GF(2)[]
 sage: x.is_gen()
 True
 sage: (x+1).is_gen()
 False
```

### is_one()

**EXAMPLES:**

```
 sage: P.<x> = GF(2)[]
 sage: P(1).is_one()
 True
```
is_zero()
EXAMPLES:

```python
sage: P.<x> = GF(2)[]
sage: x.is_zero()
False
```

list(copy=True)
EXAMPLES:

```python
sage: P.<x> = GF(2)[]
sage: x.list()
[0, 1]
sage: list(x)
[0, 1]
```

quo_rem(right)
EXAMPLES:

```python
sage: P.<x> = GF(2)[]
sage: f = x^2 + x + 1
sage: f.quo_rem(x + 1)
(x, 1)
```

shift(n)
EXAMPLES:

```python
sage: P.<x> = GF(2)[]
sage: f = x^3 + x^2 + 1
sage: f.shift(1)
x^4 + x^3 + x
sage: f.shift(-1)
x^2 + x
```

truncate(n)
Returns this polynomial mod \(x^n\).
EXAMPLES:

```python
sage: R.<x> =GF(2)[]
sage: f = sum(x^n for n in range(10)); f
x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1
sage: f.truncate(6)
x^5 + x^4 + x^3 + x^2 + x + 1
```

If the precision is higher than the degree of the polynomial then the polynomial itself is returned:

```python
sage: f.truncate(10) is f
True
```

If the precision is negative, the zero polynomial is returned:

```python
sage: f.truncate(-1)
0
```
\texttt{xgcd}(\texttt{other})

Computes extended gcd of self and other.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: P.<x> = GF(7)[]
sage: f = x*(x+1)
sage: f.xgcd(x+1)
(x + 1, 0, 1)
sage: f.xgcd(x^2)
(x, 1, 6)
\end{verbatim}

\subsection{2.1.18 Isolate Real Roots of Real Polynomials}

\textbf{AUTHOR:}

\begin{itemize}
  \item Carl Witty (2007-09-19): initial version
\end{itemize}

This is an implementation of real root isolation. That is, given a polynomial with exact real coefficients, we compute isolating intervals for the real roots of the polynomial. (Polynomials with integer, rational, or algebraic real coefficients are supported.)

We convert the polynomials into the Bernstein basis, and then use de Casteljau's algorithm and Descartes' rule of signs on the Bernstein basis polynomial (using interval arithmetic) to locate the roots. The algorithm is similar to that in "A Descartes Algorithm for Polynomials with Bit-Stream Coefficients", by Eigenwillig, Kettner, Krandick, Mehlhorn, Schmitt, and Wolpert, but has three crucial optimizations over the algorithm in that paper:

\begin{itemize}
  \item Precision reduction: at certain points in the computation, we discard the low-order bits of the coefficients, widening the intervals.
  \item Degree reduction: at certain points in the computation, we find lower-degree polynomials that are approximately equal to our high-degree polynomial over the region of interest.
  \item When the intervals are too wide to continue (either because of a too-low initial precision, or because of precision or degree reduction), and we need to restart with higher precision, we recall which regions have already been proven not to have any roots and do not examine them again.
\end{itemize}

The best description of the algorithms used (other than this source code itself) is in the slides for my Sage Days 4 talk, currently available from \url{https://wiki.sagemath.org/days4schedule}.

\begin{verbatim}
exception sage.rings.polynomial.real_roots.PrecisionError
Bases: ValueError

sage.rings.polynomial.real_roots.bernstein_down(d1, d2, s)
Given polynomial degrees d1 and d2 (where d1 < d2), and a number of samples s, computes a matrix bd.

If you have a Bernstein polynomial of formal degree d2, and select s of its coefficients (according to subsample_vec), and multiply the resulting vector by bd, then you get the coefficients of a Bernstein polynomial of formal degree d1, where this second polynomial is a good approximation to the first polynomial over the region of the Bernstein basis.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.rings.polynomial.real_roots import *
sage: bernstein_down(3, 8, 5)
[ 612/245 -348/245 -37/49 338/245 -172/245
[-724/441 132/49 395/441 -290/147 452/441]
\end{verbatim}
\end{verbatim}
sage.rings.polynomial.real_roots.bernstein_expand\((c, d2)\)

Given an integer vector representing a Bernstein polynomial \(p\), and a degree \(d2\), compute the representation of \(p\) as a Bernstein polynomial of formal degree \(d2\).

This is similar to multiplying by the result of bernstein_up, but should be faster for large \(d2\) (this has about the same number of multiplies, but in this version all the multiplies are by single machine words).

Returns a pair consisting of the expanded polynomial, and the maximum error \(E\). (So if an element of the returned polynomial is \(a\), and the true value of that coefficient is \(b\), then \(a <= b < a + E\).)

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: c = vector(ZZ, [1000, 2000, -3000])
sage: bernstein_expand(c, 3)
((1000, 1666, 333, -3000), 1)
sage: bernstein_expand(c, 4)
((1000, 1500, 1000, -500, -3000), 1)
sage: bernstein_expand(c, 20)
((1000, 1100, 1168, 1205, 1210, 1184, 1126, 1036, 915, 763, 578, 363, 115, -474, -816, -1190, -1595, -2032, -2500, -3000), 1)
```

class sage.rings.polynomial.real_roots.bernstein_polynomial_factory

An abstract base class for bernstein_polynomial factories. That is, elements of subclasses represent Bernstein polynomials (exactly), and are responsible for creating interval_bernstein_polynomial_integer approximations at arbitrary precision.

Supports four methods, coeffs_bitsize(), bernstein_polynomial(), lsign(), and usign(). The coeffs_bitsize() method gives an integer approximation to the log2 of the max of the absolute values of the Bernstein coefficients. The bernstein_polynomial(scale_log2) method gives an approximation where the maximum coefficient has approximately coeffs_bitsize() - scale_log2 bits. The lsign() and usign() methods give the (exact) sign of the first and last coefficient, respectively.

```python
lsign()
Returns the sign of the first coefficient of this Bernstein polynomial.

usign()
Returns the sign of the last coefficient of this Bernstein polynomial.
```

class sage.rings.polynomial.real_roots.bernstein_polynomial_factory_ar\((poly, neg)\)

This class holds an exact Bernstein polynomial (represented as a list of algebraic real coefficients), and returns arbitrarily-precise interval approximations of this polynomial on demand.

bernstein_polynomial\((scale_log2)\)

Compute an interval_bernstein_polynomial_integer that approximates this polynomial, using the given scale_log2. (Smaller scale_log2 values give more accurate approximations.)

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: x = polygen(AA)
```

(continues on next page)
(continued from previous page)

```python
sage: p = (x - 1) * (x - sqrt(AA(2))) * (x - 2)
sage: bpf = bernstein_polynomial_factory_ar(p, False)
sage: print(bpf.bernstein_polynomial(0))
degree 3 IBP with 2-bit coefficients
sage: bpf.bernstein_polynomial(-20)
<IBP: ((-2965821, 2181961, -1542880, 1048576) + [0 .. 1)) * 2^-20>
sage: bpf = bernstein_polynomial_factory_ar(p, True)
sage: bpf.bernstein_polynomial(-20)
<IBP: ((-2965821, -2181962, -1542880, -1048576) + [0 .. 1)) * 2^-20>
sage: p = x^2 - 1
sage: bpf = bernstein_polynomial_factory_ar(p, False)
sage: bpf.bernstein_polynomial(-10)
<IBP: ((-1024, 0, 1024) + [0 .. 1)) * 2^-10>
```

coeffs_bitsize()

Computes the approximate log2 of the maximum of the absolute values of the coefficients.

EXAMPLES:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: x = polygen(AA)
sage: p = (x - 1) * (x - sqrt(AA(2))) * (x - 2)
sage: bernstein_polynomial_factory_ar(p, False).coeffs_bitsize()
1
```

class `sage.rings.polynomial.real_roots.bernstein_polynomial_factory_intlist(coefs)`

Bases: `sage.rings.polynomial.real_roots.bernstein_polynomial_factory`

This class holds an exact Bernstein polynomial (represented as a list of integer coefficients), and returns arbitrarily-precise interval approximations of this polynomial on demand.

bernstein_polynomial(scale_log2)

Compute an interval_bernstein_polynomial_integer that approximates this polynomial, using the given scale_log2. (Smaller scale_log2 values give more accurate approximations.)

EXAMPLES:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: bpf = bernstein_polynomial_factory_intlist([10, -20, 30, -40])
sage: print(bpf.bernstein_polynomial(0))
degree 3 IBP with 6-bit coefficients
sage: bpf.bernstein_polynomial(20)
<IBP: ((0, -1, 0, -1) + [0 .. 1)) * 2^20; lsign 1>
sage: bpf.bernstein_polynomial(0)
<IBP: (10, -20, 30, -40) + [0 .. 1)>
sage: bpf.bernstein_polynomial(-20)
<IBP: ((10485760, -20971520, 31457280, -41943040) + [0 .. 1)) * 2^-20>
```

coeffs_bitsize()

Computes the approximate log2 of the maximum of the absolute values of the coefficients.

EXAMPLES:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: bernstein_polynomial_factory_intlist([1, 2, 3, -60000]).coeffs_bitsize()
```
class sage.rings.polynomial.real_roots.bernstein_polynomial_factory_ratlist(coeffs)
Bases: sage.rings.polynomial.real_roots.bernstein_polynomial_factory

This class holds an exact Bernstein polynomial (represented as a list of rational coefficients), and returns arbitrarily-precise interval approximations of this polynomial on demand.

bernstein_polynomial(scale_log2)
Compute an interval_bernstein_polynomial_integer that approximates this polynomial, using the given scale_log2. (Smaller scale_log2 values give more accurate approximations.)

EXAMPLES:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: bpf = bernstein_polynomial_factory_ratlist([1/3, -22/7, 193/71, -140/99])
sage: print(bpf.bernstein_polynomial(0))
degree 3 IBP with 3-bit coefficients
sage: bpf.bernstein_polynomial(20)
<IBP: ((0, -1, 0, -1) + [0 .. 1)) * 2^20; lsign 1>
sage: bpf.bernstein_polynomial(0)
<IBP: (0, -4, 2, -2) + [0 .. 1); lsign 1>
sage: bpf.bernstein_polynomial(-20)
<IBP: ((349525, -3295525, 2850354, -1482835) + [0 .. 1)) * 2^-20>
```

coeffs_bitsize()
Computes the approximate log2 of the maximum of the absolute values of the coefficients.

EXAMPLES:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: bernstein_polynomial_factory_ratlist([1, 2, 3, -60000]).coeffs_bitsize()
15
sage: bernstein_polynomial_factory_ratlist([65535/65536]).coeffs_bitsize()
-1
sage: bernstein_polynomial_factory_ratlist([65536/65535]).coeffs_bitsize()
1
```

sage.rings.polynomial.real_roots.bernstein_up(dl, d2, s=None)
Given polynomial degrees dl and d2, where dl < d2, compute a matrix bu.

If you have a Bernstein polynomial of formal degree dl, and multiply its coefficient vector by bu, then the result is the coefficient vector of the same polynomial represented as a Bernstein polynomial of formal degree d2.

If s is not None, then it represents a number of samples; then the product only gives s of the coefficients of the new Bernstein polynomial, selected according to subsample_vec.

EXAMPLES:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: bernstein_down(3, 7, 4)
[ 12/5 -4 3 -2/5]
[-13/15 16/3 -4 8/15]
[ 8/15 -4 16/3 -13/15]
[ -2/5 3 -4 12/5]
```

sage.rings.polynomial.real_roots.bitsize_doctest(n)
sage.rings.polynomial.real_roots.cl_maximum_root(cl)
Given a polynomial represented by a list of its coefficients (as RealIntervalFieldElements), compute an upper bound on its largest real root.

Uses two algorithms of Akritas, Strzeboński, and Vigklas, and picks the better result.

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: cl_maximum_root([RIF(-1), RIF(0), RIF(1)])
1.00000000000000
```

sage.rings.polynomial.real_roots.cl_maximum_root_first_lambda(cl)
Given a polynomial represented by a list of its coefficients (as RealIntervalFieldElements), compute an upper bound on its largest real root.


EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: cl_maximum_root_first_lambda([RIF(-1), RIF(0), RIF(1)])
1.00000000000000
```

sage.rings.polynomial.real_roots.cl_maximum_root_local_max(cl)
Given a polynomial represented by a list of its coefficients (as RealIntervalFieldElements), compute an upper bound on its largest real root.


EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: cl_maximum_root_local_max([RIF(-1), RIF(0), RIF(1)])
1.41421356237310
```

class sage.rings.polynomial.real_roots.context
Bases: object

A simple context class, which is passed through parts of the real root isolation algorithm to avoid global variables.

Holds logging information, a random number generator, and the target machine wordsize.

```
get_be_log()
```

```
get_dc_log()
```

sage.rings.polynomial.real_roots.de_casteljau_doublevec(c, x)
Given a polynomial in Bernstein form with floating-point coefficients over the region [0 .. 1], and a split point x, use de Casteljau’s algorithm to give polynomials in Bernstein form over [0 .. x] and [x .. 1].

This function will work for an arbitrary rational split point x, as long as 0 < x < 1; but it has a specialized code path for x==1/2.

```
INPUT:
• c – vector of coefficients of polynomial in Bernstein form
• x – rational splitting point; 0 < x < 1
```

```
OUTPUT:
```
• **c1** – coefficients of polynomial over range [0 .. x]

• **c2** – coefficients of polynomial over range [x .. 1]

• **err_inc** – number of half-ulps by which error intervals widened

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: c = vector(RDF, [0.7, 0, 0, 0, 0, 0])
sage: de_casteljau_doublevec(c, 1/2)
((0.7, 0.35, 0.175, 0.0875, 0.04375, 0.021875), (0.021875, 0.0, 0.0, 0.0, 0.0, 0.0), ← 5)
sage: de_casteljau_doublevec(c, 1/3)  # rel tol
((0.7, 0.4666666666666667, 0.31111111111111117, 0.20740740740740746, 0.
→ 13827160493827165, 0.09218106995884777), (0.09218106995884777, 0.0, 0.0, 0.0, 0.0,
→ 0.0), 15)
sage: de_casteljau_doublevec(c, 7/22)  # rel tol
((0.7, 0.4772727272727273, 0.3254132231404959, 0.22187265214124724, 0.
→ 15127680827812312, 0.10314327837144759), (0.10314327837144759, 0.0, 0.0, 0.0, 0.0,
→ 0.0), 15)
```

```python
sage.rings.polynomial.real_roots.de_casteljau_intvec(c, c_bitsize, x, use_ints)
```

Given a polynomial in Bernstein form with integer coefficients over the region [0 .. 1], and a split point x, use de Casteljau’s algorithm to give polynomials in Bernstein form over [0 .. x] and [x .. 1].

This function will work for an arbitrary rational split point x, as long as 0 < x < 1; but it has specialized code paths that make some values of x faster than others. If \( x = \frac{a}{a + b} \), there are special efficient cases for \( a=1, b=1 \), \( a+b \) fits in a machine word, \( a+b \) is a power of 2, \( a \) fits in a machine word, \( b \) fits in a machine word. The most efficient case is \( x=1/2 \).

Given split points \( x = \frac{a}{a + b} \) and \( y = \frac{c}{c + d} \), where \( \min(a, b) \) and \( \min(c, d) \) fit in the same number of machine words and \( a+b \) and \( c+d \) are both powers of two, then \( x \) and \( y \) should be equally fast split points.

If use_ints is nonzero, then instead of checking whether numerators and denominators fit in machine words, we check whether they fit in ints (32 bits, even on 64-bit machines). This slows things down, but allows for identical results across machines.

**INPUT:**

• **c** – vector of coefficients of polynomial in Bernstein form

• **c_bitsize** – approximate size of coefficients in c (in bits)

• **x** – rational splitting point; 0 < x < 1

**OUTPUT:**

• **c1** – coefficients of polynomial over range [0 .. x]

• **c2** – coefficients of polynomial over range [x .. 1]

• **err_inc** – amount by which error intervals widened

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: c = vector(ZZ, [1048576, 0, 0, 0, 0, 0])
sage: de_casteljau_intvec(c, 20, 1/2, 1)
((1048576, 524288, 262144, 131072, 65536, 32768), (32768, 0, 0, 0, 0, 0), 1)
sage: de_casteljau_intvec(c, 20, 1/3, 1)
```

(continues on next page)
((1048576, 699050, 466033, 310689, 207126, 138084), (138084, 0, 0, 0, 0, 0), 1)

sage: de_casteljau_intvec(c, 20, 7/22, 1)
((1048576, 714938, 487457, 332357, 226607, 154505), (154505, 0, 0, 0, 0, 0), 1)

sage.rings.polynomial.real_roots.degree_reduction_next_size(n)
Given n (a polynomial degree), returns either a smaller integer or None. This defines the sequence of degrees followed by our degree reduction implementation.

EXAMPLES:

sage: from sage.rings.polynomial.real_roots import *

sage: degree_reduction_next_size(1000)
30
sage: degree_reduction_next_size(20)
15
sage: degree_reduction_next_size(3)
2
sage: degree_reduction_next_size(2) is None
True

sage.rings.polynomial.real_roots.dprod_imatrow_vec(m, v, k)
Computes the dot product of row k of the matrix m with the vector v (that is, compute one element of the product m*v).
If v has more elements than m has columns, then elements of v are selected using subsample_vec.

EXAMPLES:

sage: from sage.rings.polynomial.real_roots import *

sage: m = matrix(3, range(9))
sage: dprod_imatrow_vec(m, vector(ZZ, [1, 0, 0, 0]), 1)
0
sage: dprod_imatrow_vec(m, vector(ZZ, [0, 1, 0, 0]), 1)
3
sage: dprod_imatrow_vec(m, vector(ZZ, [0, 0, 1, 0]), 1)
4
sage: dprod_imatrow_vec(m, vector(ZZ, [0, 0, 0, 1]), 1)
5
sage: dprod_imatrow_vec(m, vector(ZZ, [1, 0, 0]), 1)
3
sage: dprod_imatrow_vec(m, vector(ZZ, [0, 1, 0]), 1)
4
sage: dprod_imatrow_vec(m, vector(ZZ, [0, 0, 1]), 1)
5
sage: dprod_imatrow_vec(m, vector(ZZ, [1, 2, 3]), 1)
26

sage.rings.polynomial.real_roots.get_realfield_rndu(n)
A simple cache for RealField fields (with rounding set to round-to-positive-infinity).

EXAMPLES:

sage: from sage.rings.polynomial.real_roots import *

sage: get_realfield_rndu(20)
Real Field with 20 bits of precision and rounding RNDU
class sage.rings.polynomial.real_roots.interval_bernstein_polynomial

Bases: object

An interval_bernstein_polynomial is an approximation to an exact polynomial. This approximation is in the form of a Bernstein polynomial (a polynomial given as coefficients over a Bernstein basis) with interval coefficients.

The Bernstein basis of degree n over the region \([a .. b]\) is the set of polynomials

\[
\binom{n}{k} (x - a)^k (b - x)^{n-k} / (b - a)^n
\]

for \(0 \leq k \leq n\).

A degree-n interval Bernstein polynomial \(P\) with its region \([a .. b]\) can represent an exact polynomial \(p\) in two different ways: it can “contain” the polynomial or it can “bound” the polynomial.

We say that \(P\) contains \(p\) if, when \(p\) is represented as a degree-n Bernstein polynomial over \([a .. b]\), its coefficients are contained in the corresponding interval coefficients of \(P\). For instance, \([0.9 .. 1.1]*x^2\) (which is a degree-2 interval Bernstein polynomial over \([0 .. 1]\)) contains \(x^2\).

We say that \(P\) bounds \(p\) if, for all \(a \leq x \leq b\), there exists a polynomial \(p'\) contained in \(P\) such that \(p(x) = p'(x)\). For instance, \([0 .. 1]*x\) is a degree-1 interval Bernstein polynomial which bounds \(x^2\) over \([0 .. 1]\).

If \(P\) contains \(p\), then \(P\) bounds \(p\); but the converse is not necessarily true. In particular, if \(n < m\), it is possible for a degree-n interval Bernstein polynomial to bound a degree-m polynomial; but it cannot contain the polynomial.

In the case where \(P\) bounds \(p\), we maintain extra information, the “slope error”. We say that \(P\) (over \([a .. b]\)) bounds \(p\) with a slope error of \(E\) (where \(E\) is an interval) if there is a polynomial \(p'\) contained in \(P\) such that the derivative of \((p - p')\) is bounded by \(E\) in the range \([a .. b]\). If \(P\) bounds \(p\) with a slope error of 0 then \(P\) contains \(p\).

(Note that “contains” and “bounds” are not standard terminology; I just made them up.)

Interval Bernstein polynomials are useful in finding real roots because of the following properties:

- Given an exact real polynomial \(p\), we can compute an interval Bernstein polynomial over an arbitrary region containing \(p\).
- Given an interval Bernstein polynomial \(P\) over \([a .. c]\), where \(a < b < c\), we can compute interval Bernstein polynomials \(P_1\) over \([a .. b]\) and \(P_2\) over \([b .. c]\), where \(P_1\) and \(P_2\) contain (or bound) all polynomials that \(P\) contains (or bounds).
- Given a degree-n interval Bernstein polynomial \(P\) over \([a .. b]\), and \(m < n\), we can compute a degree-m interval Bernstein polynomial \(P'\) over \([a .. b]\) that bounds all polynomials that \(P\) bounds.
- It is sometimes possible to prove that no polynomial bounded by \(P\) over \([a .. b]\) has any roots in \([a .. b]\). (Roughly, this is possible when no polynomial contained by \(P\) has any complex roots near the line segment \([a .. b]\), where “near” is defined relative to the length \(b-a\).)
- It is sometimes possible to prove that every polynomial bounded by \(P\) over \([a .. b]\) with slope error \(E\) has exactly one root in \([a .. b]\). (Roughly, this is possible when every polynomial contained by \(P\) over \([a .. b]\) has exactly one root in \([a .. b]\), and every other complex roots near the line segment \([a .. b]\), and every polynomial contained in \(P\) has a derivative which is bounded away from zero over \([a .. b]\) by an amount which is large relative to \(E\).)
- Starting from a sufficiently precise interval Bernstein polynomial, it is always possible to split it into polynomials which provably have 0 or 1 roots (as long as your original polynomial has no multiple real roots).
So a rough outline of a family of algorithms would be:

• Given a polynomial $p$, compute a region $[a .. b]$ in which any real roots must lie.
• Compute an interval Bernstein polynomial $P$ containing $p$ over $[a .. b]$.
• Keep splitting $P$ until you have isolated all the roots. Optionally, reduce the degree or the precision of the interval Bernstein polynomials at intermediate stages (to reduce computation time). If this seems not to be working, go back and try again with higher precision.

Obviously, there are many details to be worked out to turn this into a full algorithm, like:

• What initial precision is selected for computing $P$?
• How do you decide when to reduce the degree of intermediate polynomials?
• How do you decide when to reduce the precision of intermediate polynomials?
• How do you decide where to split the interval Bernstein polynomial regions?
• How do you decide when to give up and start over with higher precision?

Each set of answers to these questions gives a different algorithm (potentially with very different performance characteristics), but all of them can use this `interval_bernstein_polynomial` class as their basic building block.

To save computation time, all coefficients in an `interval_bernstein_polynomial` share the same interval width. (There is one exception: when creating an `interval_bernstein_polynomial`, the first and last coefficients can be marked as “known positive” or “known negative”. This has some of the same effect as having a (potentially) smaller interval width for these two coefficients, although it does not affect de Casteljau splitting.) To allow for widely varying coefficient magnitudes, all coefficients in an `interval_bernstein_polynomial` are scaled by $2^n$ (where $n$ may be positive, negative, or zero).

There are two representations for `interval_bernstein_polynomials`, integer and floating-point. These are the two subclasses of this class; `interval_bernstein_polynomial` itself is an abstract class.

`interval_bernstein_polynomial` and its subclasses are not expected to be used outside this file.

```python
region()
region_width()
try_rand_split(ctx, logging_note)
```

## EXAMPLES:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpi([50, 20, -90, -70, 200], error=5)
sage: bp1, bp2, _ = bp.try_rand_split(mk_context(), None)
sage: bp1
<IBP: (50, 29, -27, -56, -11) + [0 .. 6) over [0 .. 43/64]>
sage: bp2
<IBP: (-11, 10, 49, 111, 200) + [0 .. 6) over [43/64 .. 1]>
sage: bp1, bp2, _ = bp.try_rand_split(mk_context(seed=42), None)
sage: bp1
<IBP: (50, 32, -11, -41, -29) + [0 .. 6) over [0 .. 583/1024]>
sage: bp2
```

(continues on next page)
try_split(ctx, logging_note)

Try doing a de Casteljau split of this polynomial at 1/2, resulting in polynomials p1 and p2. If we see that the sign of this polynomial is determined at 1/2, then return (p1, p2, 1/2); otherwise, return None.

EXAMPLES:

sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpi([50, 20, -90, -70, 200], error=5)
sage: bp1, bp2, _ = bp.try_split(mk_context(), None)
sage: bp1
<IBP: (50, 35, 0, -29, -31) + [0 .. 6) over [0 .. 1/2]>
sage: bp2
<IBP: (-31, -33, -8, 65, 200) + [0 .. 6) over [1/2 .. 1]>

variations()

Consider a polynomial (written in either the normal power basis or the Bernstein basis). Take its list of coefficients, omitting zeroes. Count the number of positions in the list where the sign of one coefficient is opposite the sign of the next coefficient.

This count is the number of sign variations of the polynomial. According to Descartes’ rule of signs, the number of real roots of the polynomial (counted with multiplicity) in a certain interval is always less than or equal to the number of sign variations, and the difference is always even. (If the polynomial is written in the power basis, the region is the positive reals; if the polynomial is written in the Bernstein basis over a particular region, then we count roots in that region.)

In particular, a polynomial with no sign variations has no real roots in the region, and a polynomial with one sign variation has one real root in the region.

In an interval Bernstein polynomial, we do not necessarily know the signs of the coefficients (if some of the coefficient intervals contain zero), so the polynomials contained by this interval polynomial may not all have the same number of sign variations. However, we can compute a range of possible numbers of sign variations.

This function returns the range, as a 2-tuple of integers.

class sage.rings.polynomial.real_roots.interval_bernstein_polynomial_float
Bases: sage.rings.polynomial.real_roots.interval_bernstein_polynomial
This is the subclass of interval_bernstein_polynomial where polynomial coefficients are represented using floating-point numbers.

In the floating-point representation, each coefficient is represented as an IEEE double-precision float $A$, and the (shared) lower and upper interval widths $E1$ and $E2$. These represent the coefficients $(A+E1)*2^n <= c <= (A+E2)*2^n$.

Note that we always have $E1 <= 0 <= E2$. Also, each floating-point coefficient has absolute value less than one.

(Note that mk_ibpf is a simple helper function for creating elements of interval_bernstein_polynomial_float in doctests.)

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpf([0.1, 0.2, 0.3], pos_err=0.5); print(bp)
degree 2 IBP with floating-point coefficients
sage: bp
<IBP: (0.1, 0.2, 0.3) + [0.0 .. 0.5]>
sage: bp.variations()
(0, 0)
sage: bp = mk_ibpf([-0.3, -0.1, 0.1, -0.1, -0.3, -0.1], lower=1, upper=5/4, usign=1,
              pos_err=0.2, scale_log2=-3, level=2, slope_err=RIF(pi)); print(bp)
degree 5 IBP with floating-point coefficients
sage: bp
<IBP: ((-0.3, -0.1, 0.1, -0.1, -0.3, -0.1) + [0.0 .. 0.2]) * 2^-3 over [1 .. 5/4];
              usign 1; level 2; slope_err 3.141592653589794?>
sage: bp.variations()
(3, 3)
```

**as_float()**

**de_casteljau(ctx, mid, msign=0)**

Uses de Casteljau’s algorithm to compute the representation of this polynomial in a Bernstein basis over new regions.

**INPUT:**

- `mid` – where to split the Bernstein basis region; $0 < mid < 1$
- `msign` – default 0 (unknown); the sign of this polynomial at `mid`

**OUTPUT:**

- `bp1`, `bp2` – the new interval Bernstein polynomials
- `ok` – a boolean; True if the sign of the original polynomial at `mid` is known

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: ctx = mk_context()
sage: bp = mk_ibpf([0.5, 0.2, -0.9, -0.7, 0.99], neg_err=-0.1, pos_err=0.01)
sage: bp1, bp2, ok = bp.de_casteljau(ctx, 1/2)
sage: bp1
<IBP: (0.5, 0.35, 0.0, -0.2875, -0.369375) + [-0.10000000000000023 .. 0.
              010000000000000226] over [0 .. 1/2]>
sage: bp2
<IBP: (-0.369375, -0.45125, -0.3275, 0.14500000000000002, 0.99) + [-0.
              10000000000000023 .. 0.010000000000000226] over [1/2 .. 1]>
```
sage: bp1, bp2, ok = bp.de_casteljau(ctx, 2/3)
sage: bp1
# rel tol 2e-16
<IBP: (0.5, 0.30000000000000004, -0.2555555555555555, -0.5444444444444444, -0.
   →32172839506172846) + [-0.10000000000000006 .. 0.0100000000000000677] over [0 ..
   →2/3]>
sage: bp2
# rel tol 3e-15
<IBP: (-0.32172839506172846, -0.21037037037037046, 0.028888888888888797, 0.
   →42666666666666666, 0.99) + [-0.10000000000000006 .. 0.0100000000000000677] over␣
   →[2/3 .. 1]>
sage: bp1, bp2, ok = bp.de_casteljau(ctx, 7/39)
sage: bp1
# rel tol
<IBP: (0.5, 0.4461538461538461, 0.3665317422748183, 0.27328680523946786, 0.
   →176592706232836) + [-0.10000000000000006 .. 0.0100000000000000677] over [
   →0 .. 7/39]>
sage: bp2
# rel tol
<IBP: (0.176592706232836, -0.26556803047927313, -0.7802038132807364, -0.
   →3966666666666666, 0.99) + [-0.10000000000000006 .. 0.0100000000000000677] over␣
   →[7/39 .. 1]>

get_msb_bit()
Returns an approximation of the log2 of the maximum of the absolute values of the coefficients, as an
integer.
slope_range()
Compute a bound on the derivative of this polynomial, over its region.

EXAMPLES:

sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpf([0.5, 0.2, -0.9, -0.7, 0.99], neg_err=-0.1, pos_err=0.01)
sage: bp.slope_range().str(style='brackets')
'[-4.8400000000000017 .. 7.2000000000000011]'

class sage.rings.polynomial.real_roots.interval_bernstein_polynomial_integer
Bases: sage.rings.polynomial.real_roots.interval_bernstein_polynomial

This is the subclass of interval_bernstein_polynomial where polynomial coefficients are represented using inte-
gers.

In this integer representation, each coefficient is represented by a GMP arbitrary-precision integer A, and a
(shared) interval width E (which is a machine integer). These represent the coefficients A*2^n <= c < (A+E)*2^n.
(Note that mk_ibpi is a simple helper function for creating elements of interval_bernstein_polynomial_integer
in doctests.)

EXAMPLES:

sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpi([1, 2, 3], error=5); print(bp)
degree 2 IBP with 2-bit coefficients
sage: bp
<IBP: (1, 2, 3) + [0 .. 5]>
sage: bp.variations()
(0, 0)
sage: bp = mk_ibpi([-3, -1, 1, -1, -3, -1], lower=1, upper=5/4, usign=1, error=2,␣
   →scale_log2=-3, level=2, slope_err=RIF(pi)); print(bp)
degree 5 IBP with 2-bit coefficients

```
sage: bp
<IBP: ((-3, -1, 1, -1, -3, -1) + [0 .. 2)) * 2^-3 over [1 .. 5/4]; usign 1; level 2;
  → slope_err 3.141592653589794?>
sage: bp.variations()
(3, 3)
```

```
as_float()

Compute an interval_bernstein_polynomial_float which contains (or bounds) all the polynomials this in-
terval polynomial contains (or bounds).

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpi([50, 20, -90, -70, 200], error=5)
sage: print(bp.as_float())
degree 4 IBP with floating-point coefficients
<sage:bp.as_float()>
<IBP: ((0.1953125, 0.078125, -0.3515625, -0.2734375, 0.78125) + [-1.
  → 1275702593849246e-16 .. 0.01953125000000017]) * 2^8>
```

```
de_casteljau(ctx, mid, msign=0)

Uses de Casteljau's algorithm to compute the representation of this polynomial in a Bernstein basis over
new regions.

INPUT:
- mid – where to split the Bernstein basis region; 0 < mid < 1
- msign – default 0 (unknown); the sign of this polynomial at mid

OUTPUT:
- bp1, bp2 – the new interval Bernstein polynomials
- ok – a boolean; True if the sign of the original polynomial at mid is known

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpi([50, 20, -90, -70, 200], error=5)
sage: ctx = mk_context()
sage: bp1, bp2, ok = bp.de_casteljau(ctx, 1/2)
sage: bp1
<IBP: (50, 35, 0, -29, -31) + [0 .. 6) over [0 .. 1/2]>
sage: bp2
<IBP: (-31, -33, -8, 65, 200) + [0 .. 6) over [1/2 .. 1]>
sage: bp1, bp2, ok = bp.de_casteljau(ctx, 2/3)
sage: bp1
<IBP: (50, 30, -26, -55, -13) + [0 .. 6) over [0 .. 2/3]>
sage: bp2
<IBP: (-13, 8, 47, 110, 200) + [0 .. 6) over [2/3 .. 1]>
sage: bp1, bp2, ok = bp.de_casteljau(ctx, 7/39)
sage: bp1
<IBP: (50, 44, 36, 27, 17) + [0 .. 6) over [0 .. 7/39]>
sage: bp2
<IBP: (17, -26, -75, -22, 200) + [0 .. 6) over [7/39 .. 1]>
```
**down_degree** *(ctx, max_err, exp_err_shift)*

Compute an interval_bernstein_polynomial_integer which bounds all the polynomials this interval polynomial bounds, but is of lesser degree.

During the computation, we find an “expected error” expected_err, which is the error inherent in our approach (this depends on the degrees involved, and is proportional to the error of the current polynomial).

We require that the error of the new interval polynomial be bounded both by max_err, and by expected_err << exp_err_shift. If we find such a polynomial p, then we return a pair of p and some debugging/logging information. Otherwise, we return the pair (None, None).

If the resulting polynomial would have error more than 2^17, then it is downscaled before returning.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpi([0, 100, 400, 903], error=2)
sage: ctx = mk_context()
sage: bp
<IBP: (0, 100, 400, 903) + [0 .. 2)>
sage: dbp, _ = bp.down_degree(ctx, 10, 32)
sage: dbp
<IBP: (-1, 148, 901) + [0 .. 4); level 1; slope_err 0.?e2>
```

**down_degree_iter** *(ctx, max_scale)*

Compute a degree-reduced version of this interval polynomial, by iterating down_degree.

We stop when degree reduction would give a polynomial which is too inaccurate, meaning that either we think the current polynomial may have more roots in its region than the degree of the reduced polynomial, or that the least significant accurate bit in the result (on the absolute scale) would be larger than 1 << max_scale.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpi([0, 100, 400, 903, 1600, 2500], error=2)
sage: ctx = mk_context()
sage: bp
<IBP: (0, 100, 400, 903, 1600, 2500) + [0 .. 2)>
sage: rbp = bp.down_degree_iter(ctx, 6)
sage: rbp
<IBP: (-4, 249, 2497) + [0 .. 9); level 2; slope_err 0.?e3>
```

**downscale** *(bits)*

Compute an interval_bernstein_polynomial_integer which contains (or bounds) all the polynomials this interval polynomial contains (or bounds), but uses “bits” fewer bits.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpi([0, 100, 400, 903], error=2)
sage: bp.downscale(5)
<IBP: ((0, 3, 12, 28) + [0 .. 1)) * 2^5>
```

**get_msb_bit()**

Returns an approximation of the log2 of the maximum of the absolute values of the coefficients, as an integer.
slope_range()
Compute a bound on the derivative of this polynomial, over its region.

EXAMPLES:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpi([0, 100, 400, 903], error=2)
sage: bp.slope_range().str(style='brackets')
'[294.0000000000000 .. 1515.0000000000000]'
```

sage.rings.polynomial.real_roots.intvec_to_doublevec(b, err)
Given a vector of integers \( A = [a_1, \ldots, a_n] \), and an integer error bound \( E \), returns a vector of floating-point numbers \( B = [b_1, \ldots, b_n] \), lower and upper error bounds \( F_1 \) and \( F_2 \), and a scaling factor \( d \), such that

\[
(b_k + F_1) * 2^d \leq a_k
\]

and

\[
a_k + E \leq (b_k + F_2) * 2^d
\]

If \( b_j \) is the element of \( B \) with largest absolute value, then \( 0.5 \leq \text{abs}(b_j) < 1.0 \).

EXAMPLES:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: intvec_to_doublevec(vector(ZZ, [1, 2, 3, 4, 5]), 3)
((0.125, 0.25, 0.375, 0.5, 0.625), -1.1275702593849246e-16, 0.37500000000000017, 3)
```

class sage.rings.polynomial.real_roots.island

Bases: object

This implements the island portion of my ocean-island root isolation algorithm. See the documentation for class ocean, for more information on the overall algorithm.

Island root refinement starts with a Bernstein polynomial whose region is the whole island (or perhaps slightly more than the island in certain cases). There are two subalgorithms; one when looking at a Bernstein polynomial covering a whole island (so we know that there are gaps on the left and right), and one when looking at a Bernstein polynomial covering the left segment of an island (so we know that there is a gap on the left, but the right is in the middle of an island). An important invariant of the left-segment subalgorithm over the region \([l .. r]\) is that it always finds a gap \([r_0 .. r]\) ending at its right endpoint.

Ignoring degree reduction, downscaling (precision reduction), and failures to split, the algorithm is roughly:

Whole island:
1. If the island definitely has exactly one root, then return.
2. Split the island in (approximately) half.
3. If both halves definitely have no roots, then remove this island from its doubly-linked list (merging its left and right gaps) and return.
4. If either half definitely has no roots, then discard that half and call the whole-island algorithm with the other half, then return.
5. If both halves may have roots, then call the left-segment algorithm on the left half.
6. We now know that there is a gap immediately to the left of the right half, so call the whole-island algorithm on the right half, then return.

Left segment:
1. Split the left segment in (approximately) half.

2. If both halves definitely have no roots, then extend the left gap over the segment and return.

3. If the left half definitely has no roots, then extend the left gap over this half and call the left-segment algorithm on the right half, then return.

4. If the right half definitely has no roots, then split the island in two, creating a new gap. Call the whole-island algorithm on the left half, then return.

5. Both halves may have roots. Call the left-segment algorithm on the left half.

6. We now know that there is a gap immediately to the left of the right half, so call the left-segment algorithm on the right half, then return.

Degree reduction complicates this picture only slightly. Basically, we use heuristics to decide when degree reduction might be likely to succeed and be helpful; whenever this is the case, we attempt degree reduction.

Precision reduction and split failure add more complications. The algorithm maintains a stack of different-precision representations of the interval Bernstein polynomial. The base of the stack is at the highest (currently known) precision; each stack entry has approximately half the precision of the entry below it. When we do a split, we pop off the top of the stack, split it, then push whichever half we’re interested in back on the stack (so the different Bernstein polynomials may be over different regions). When we push a polynomial onto the stack, we may heuristically decide to push further lower-precision versions of the same polynomial onto the stack.

In the algorithm above, whenever we say “split in (approximately) half”, we attempt to split the top-of-stack polynomial using try_split() and try_rand_split(). However, these will fail if the sign of the polynomial at the chosen split point is unknown (if the polynomial is not known to high enough precision, or if the chosen split point actually happens to be a root of the polynomial). If this fails, then we discard the top-of-stack polynomial, and try again with the next polynomial down (which has approximately twice the precision). This next polynomial may not be over the same region; if not, we split it using de Casteljau’s algorithm to get a polynomial over (approximately) the same region first.

If we run out of higher-precision polynomials (if we empty out the entire stack), then we give up on root refinement for this island. The ocean class will notice this, provide the island with a higher-precision polynomial, and restart root refinement. Basically the only information kept in that case is the lower and upper bounds on the island. Since these are updated whenever we discover a “half” (of an island or a segment) that definitely contains no roots, we never need to re-examine these gaps. (We could keep more information. For example, we could keep a record of split points that succeeded and failed. However, a split point that failed at lower precision is likely to succeed at higher precision, so it’s not worth avoiding. It could be useful to select split points that are known to succeed, but starting from a new Bernstein polynomial over a slightly different region, hitting such split points would require de Casteljau splits with non-power-of-two denominators, which are much much slower.)

**bp_done**(bp)

Examine the given Bernstein polynomial to see if it is known to have exactly one root in its region. (In addition, we require that the polynomial region not include 0 or 1. This makes things work if the user gives explicit bounds to real_roots(), where the lower or upper bound is a root of the polynomial. real_roots() deals with this by explicitly detecting it, dividing out the appropriate linear polynomial, and adding the root to the returned list of roots; but then if the island considers itself “done” with a region including 0 or 1, the returned root regions can overlap with each other.)

**done**(ctx)

Check to see if the island is known to contain zero roots or is known to contain one root.

**has_root**()

Assuming that the island is done (has either 0 or 1 roots), reports whether the island has a root.

**less_bits**(ancestors, bp)

Heuristically pushes lower-precision polynomials on the polynomial stack. See the class documentation for class island for more information.
more_bits$(ctx, ancestors, bp, rightmost)$

Find a Bernstein polynomial on the “ancestors” stack with more precision than $bp$; if it is over a different region, then shrink its region to (approximately) match that of $bp$. (If this is rightmost – if $bp$ covers the whole island – then we only require that the new region cover the whole island fairly tightly; if this is not rightmost, then the new region will have exactly the same right boundary as $bp$, although the left boundary may vary slightly.)

refine$(ctx)$

Attempts to shrink and/or split this island into sub-island that each definitely contain exactly one root.

refine_recurse$(ctx, bp, ancestors, history, rightmost)$

This implements the root isolation algorithm described in the class documentation for class island. This is the implementation of both the whole-island and the left-segment algorithms; if the flag rightmost is True, then it is the whole-island algorithm, otherwise the left-segment algorithm.

The precision-reduction stack is (ancestors + $[bp]$); that is, the top-of-stack is maintained separately.

reset_root_width$(target_width)$

Modify the criteria for this island to require that it is not “done” until its width is less than or equal to $target_width$.

shrink_bp$(ctx)$

If the island’s Bernstein polynomial covers a region much larger than the island itself (in particular, if either the island’s left gap or right gap are totally contained in the polynomial’s region) then shrink the polynomial down to cover the island more tightly.

class sage.rings.polynomial.real_roots.linear_map$(lower, upper)$

Bases: object

A simple class to map linearly between original coordinates (ranging from $[lower .. upper]$) and ocean coordinates (ranging from $[0 .. 1]$).

from_ocean$(region)$

to_ocean$(region)$

sage.rings.polynomial.real_roots.max_abs_doublevec$(c)$

Given a floating-point vector, return the maximum of the absolute values of its elements.

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: max_abs_doublevec(vector(RDF, [0.1, -0.767, 0.3, 0.693]))
0.767
```

sage.rings.polynomial.real_roots.max_bitsize_intvec_doctest$(b)$

sage.rings.polynomial.real_roots.maximum_root_first_lambda$(p)$

Given a polynomial with real coefficients, computes an upper bound on its largest real root, using the first-lambda algorithm from “Implementations of a New Theorem for Computing Bounds for Positive Roots of Polynomials”, by Akritas, Strzeboński, and Vigklas.

EXAMPLES:

```
sage: from sage.rings.polynomial.real_roots import *
sage: x = polygen(ZZ)
sage: maximum_root_first_lambda((x-1)*(x-2)*(x-3))
6.00000000000001
sage: maximum_root_first_lambda((x+1)*(x+2)*(x+3))
0.000000000000000
```

(continues on next page)
sage rings polynomial real roots maximum root local max \( p \)

Given a polynomial with real coefficients, computes an upper bound on its largest real root, using the local-max algorithm from “Implementations of a New Theorem for Computing Bounds for Positive Roots of Polynomials”, by Akritas, Strzeboński, and Vigklas.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: x = polygen(ZZ)
sage: maximum_root_local_max((x-1)*(x-2)*(x-3))
12
sage: maximum_root_local_max((x+1)*(x+2)*(x+3))
0
sage: maximum_root_local_max(x^2 - 1)
1.41421356237310
```

sage rings polynomial real roots min max delta intvec \( a, b \)

Given two integer vectors \( a \) and \( b \) (of equal, nonzero length), return a pair of the minimum and maximum values taken on by \( a[i] - b[i] \).

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: a = vector(ZZ, [10, -30])
sage: b = vector(ZZ, [15, -60])
sage: min_max_delta_intvec(a, b)
(30, -5)
```

sage rings polynomial real roots min max diff doublevec \( c \)

Given a floating-point vector \( b = (b_0, ..., b_n) \), compute the minimum and maximum values of \( b_{j+1} - b_j \).

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: min_max_diff_doublevec(vector(RDF, [1, 7, -2]))
(-9.0, 6.0)
```

sage rings polynomial real roots min max diff intvec \( b \)

Given an integer vector \( b = (b_0, ..., b_n) \), compute the minimum and maximum values of \( b_{j+1} - b_j \).

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: min_max_diff_intvec(vector(ZZ, [1, 7, -2]))
(-9, 6)
```

sage rings polynomial real roots mk context \( \text{do\_logging=False, seed=0, wordsize=32} \)

A simple wrapper for creating context objects with coercions, defaults, etc.

For use in doctests.

**EXAMPLES:**
sage: from sage.rings.polynomial.real_roots import *
sage: mk_context(do_logging=True, seed=3, wordsize=64)
root isolation context: seed=3; do_logging=True; wordsize=64

sage.rings.polynomial.real_roots.mk_ibpf(coeffs, lower=0, upper=1, lsign=0, usign=0, neg_err=0, pos_err=0, scale_log2=0, level=0, slope_err=None)

A simple wrapper for creating interval_bernstein_polynomial_float objects with coercions, defaults, etc.
For use in doctests.
EXAMPLES:

sage: from sage.rings.polynomial.real_roots import *

sage: print(mk_ibpf([0.5, 0.2, -0.9, -0.7, 0.99], pos_err=0.1, neg_err=-0.01))
degree 4 IBP with floating-point coefficients

sage.rings.polynomial.real_roots.mk_ibpi(coeffs, lower=0, upper=1, lsign=0, usign=0, error=1, scale_log2=0, level=0, slope_err=None)

A simple wrapper for creating interval_bernstein_polynomial_integer objects with coercions, defaults, etc.
For use in doctests.
EXAMPLES:

class sage.rings.polynomial.real_roots.ocean
Bases: object

Given the tools we've defined so far, there are many possible root isolation algorithms that differ on where to select split points, what precision to work at when, and when to attempt degree reduction.

Here we implement one particular algorithm, which I call the ocean-island algorithm. We start with an interval Bernstein polynomial defined over the region \([0..1]\). This region is the “ocean”. Using de Casteljau’s algorithm and Descartes’ rule of signs, we divide this region into subregions which may contain roots, and subregions which are guaranteed not to contain roots. Subregions which may contain roots are “islands”; subregions known not to contain roots are “gaps”.

All the real root isolation work happens in class island. See the documentation of that class for more information.

An island can be told to refine itself until it contains only a single root. This may not succeed, if the island’s interval Bernstein polynomial does not have enough precision. The ocean basically loops, refining each of its islands, then increasing the precision of islands which did not succeed in isolating a single root; until all islands are done.

Increasing the precision of unsuccessful islands is done in a single pass using split_for_target(); this means it is possible to share work among multiple islands.

all_done()
Returns true iff all islands are known to contain exactly one root.

EXAMPLES:

sage: from sage.rings.polynomial.real_roots import *
sage: oc = ocean(mk_context(), bernstein_polynomial_factory_ratlist([1/3, -22/7, \rightarrow 193/71, -140/99]), lmap)
sage: oc.all_done()
(continues on next page)
approx_bp(scale_log2)

Returns an approximation to our Bernstein polynomial with the given scale_log2.

EXAMPLES:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: oc = ocean(mk_context(), bernstein_polynomial_factory_ratlist([1/3, -22/7, 193/71, -140/99]), lmap)
sage: oc.approx_bp(0)
<IBP: (0, -4, 2, -2) + [0 .. 1); lsign 1>
sage: oc.approx_bp(-20)
<IBP: ((349525, -3295525, 2850354, -1482835) + [0 .. 1)) * 2^-20>
```

find_roots()

Isolate all roots in this ocean.

EXAMPLES:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: oc = ocean(mk_context(), bernstein_polynomial_factory_ratlist([1/3, -22/7, 193/71, -140/99]), lmap)
sage: oc
ocean with precision 120 and 1 island(s)
sage: oc.find_roots()
sage: oc
ocean with precision 120 and 3 island(s)
```

increase_precision()

Increase the precision of the interval Bernstein polynomial held by any islands which are not done. (In normal use, calls to this function are separated by calls to self.refine_all().)

EXAMPLES:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: oc = ocean(mk_context(), bernstein_polynomial_factory_ratlist([1/3, -22/7, 193/71, -140/99]), lmap)
sage: oc
ocean with precision 120 and 1 island(s)
sage: oc.increase_precision()
sage: oc
ocean with precision 960 and 1 island(s)
```
**refine_all()**
Refine all islands which are not done (which are not known to contain exactly one root).

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: oc = ocean(mk_context(), bernstein_polynomial_factory_ratlist([1/3, -22/7, 193/71, -140/99]), lmap)
sage: oc
ocean with precision 120 and 1 island(s)
sage: oc.refine_all()
sage: oc
ocean with precision 120 and 3 island(s)
```

**reset_root_width(isle_num, target_width)**
Require that the isle_num island have a width at most target_width. If this is followed by a call to find_roots(), then the corresponding root will be refined to the specified width.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: oc = ocean(mk_context(), bernstein_polynomial_factory_ratlist([-1, -1, 1]), lmap)
sage: oc.find_roots()
sage: oc.roots()
[(1/2, 3/4)]
sage: oc.reset_root_width(0, 1/2^200)
sage: oc.find_roots()
sage: oc
ocean with precision 240 and 1 island(s)
sage: RR(RealIntervalField(300)(oc.roots()[0]).absolute_diameter()).log2()
-232.668979560890
```

**roots()**
Return the locations of all islands in this ocean. (If run after find_roots(), this is the location of all roots in the ocean.)

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: oc = ocean(mk_context(), bernstein_polynomial_factory_ratlist([1/3, -22/7, 193/71, -140/99]), lmap)
sage: oc.find_roots()
sage: oc.roots()
[(1/32, 1/16), (1/2, 5/8), (3/4, 7/8)]
sage: oc = ocean(mk_context(), bernstein_polynomial_factory_ratlist([1, 0, 1111/2, 0, 0, 0, 0, -1]), lmap)
sage: oc.find_roots()
sage: oc.roots()
[(95761241267509487747625/9671406556917033397649408, 191522482605387719863145/
  19342813113834066795298816), (1496269395904347376805/151115727451828646838272,
  374067366568272936175/37778931862957161709568), (31/32, 63/64)]
```

**sage.rings.polynomial.real_roots.precompute_degree_reduction_cache(n)**
Compute and cache the matrices used for degree reduction, starting from degree n.
EXAMPLES:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: precompute_degree_reduction_cache(5)
sage: dr_cache[5]
([121/126 8/63 -1/9 -2/63 11/126 -2/63],
[ -3/7 37/42 16/21 1/21 -3/7 1/6],
[ 1/6 -3/7 1/21 16/21 37/42 -3/7],
3, [-2/63 11/126 -2/63 -1/9 8/63 121/126], 2,
(121 16 -14 -4 11 -4],
[-54 111 96 6 -54 21],
[21 -54 6 96 111 -54],
[-4 11 -4 -14 16 121], 126)
```

sage.rings.polynomial.real_roots.pseudoinverse(m)

sage.rings.polynomial.real_roots.rational_root_bounds(p)

Given a polynomial p with real coefficients, computes rationals a and b, such that for every real root r of p, a < r < b. We try to find rationals which bound the roots somewhat tightly, yet are simple (have small numerators and denominators).

EXAMPLES:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: x = polygen(ZZ)
sage: rational_root_bounds((x-1)*(x-2)*(x-3))
(0, 7)
sage: rational_root_bounds(x^2)
(-1/2, 1/2)
sage: rational_root_bounds(x^(x+1))
(-3/2, 1/2)
sage: rational_root_bounds((x+2)*(x-3))
(-3, 6)
sage: rational_root_bounds(x^995 * (x^2 - 9999) - 1)
(-100, 1000/7)
sage: rational_root_bounds(x^995 * (x^2 - 9999) + 1)
(-142, 213/2)
```

If we can see that the polynomial has no real roots, return None.

sage: rational_root_bounds(x^2 + 7) is None
True

sage.rings.polynomial.real_roots.real_roots(p, bounds=None, seed=None, skip_squarefree=False, do_logging=False, wordsize=32, retval='rational', strategy=None, max_diameter=None)

Compute the real roots of a given polynomial with exact coefficients (integer, rational, and algebraic real coefficients are supported). Returns a list of pairs of a root and its multiplicity.

The root itself can be returned in one of three different ways. If retval=='rational', then it is returned as a pair of rationals that define a region that includes exactly one root. If retval=='interval', then it is returned as a RealIntervalFieldElement that includes exactly one root. If retval=='algebraic_real', then it is returned as an AlgebraicReal. In the former two cases, all the intervals are disjoint.

An alternate high-level algorithm can be used by selecting strategy='warp'. This affects the conversion into
Bernstein polynomial form, but still uses the same ocean-island algorithm as the default algorithm. The ‘warp’ algorithm performs the conversion into Bernstein polynomial form much more quickly, but performs the rest of the computation slightly slower in some benchmarks. The ‘warp’ algorithm is particularly likely to be helpful for low-degree polynomials.

Part of the algorithm is randomized; the seed parameter gives a seed for the random number generator. (By default, the same seed is used for every call, so that results are repeatable.) The random seed may affect the running time, or the exact intervals returned, but the results are correct regardless of the seed used.

The bounds parameter lets you find roots in some proper subinterval of the reals; it takes a pair of a rational lower and upper bound and only roots within this bound will be found. Currently, specifying bounds does not work if you select strategy=’warp’, or if you use a polynomial with algebraic real coefficients.

By default, the algorithm will do a squarefree decomposition to get squarefree polynomials. The skip_squarefree parameter lets you skip this step. (If this step is skipped, and the polynomial has a repeated real root, then the algorithm will loop forever! However, repeated non-real roots are not a problem.)

For integer and rational coefficients, the squarefree decomposition is very fast, but it may be slow for algebraic reals. (It may trigger exact computation, so it might be arbitrarily slow. The only other way that this algorithm might trigger exact computation on algebraic real coefficients is that it checks the constant term of the input polynomial for equality with zero.)

Part of the algorithm works (approximately) by splitting numbers into word-size pieces (that is, pieces that fit into a machine word). For portability, this defaults to always selecting pieces suitable for a 32-bit machine; the wordsize parameter lets you make choices suitable for a 64-bit machine instead. (This affects the running time, and the exact intervals returned, but the results are correct on both 32- and 64-bit machines even if the wordsize is chosen “wrong”.)

The precision of the results can be improved (at the expense of time, of course) by specifying the max_diameter parameter. If specified, this sets the maximum diameter() of the intervals returned. (Sage defines diameter() to be the relative diameter for intervals that do not contain 0, and the absolute diameter for intervals containing 0.) This directly affects the results in rational or interval return mode; in algebraic_real mode, it increases the precision of the intervals passed to the algebraic number package, which may speed up some operations on that algebraic real.

Some logging can be enabled with do_logging=True. If logging is enabled, then the normal values are not returned; instead, a pair of the internal context object and a list of all the roots in their internal form is returned.

ALGORITHM: We convert the polynomial into the Bernstein basis, and then use de Casteljau’s algorithm and Descartes’ rule of signs (using interval arithmetic) to locate the roots.

EXAMPLES:

```sage
glob from from sage.rings.polynomial.real_roots import *
glob sage: x = polygen(ZZ)
glob sage: real_roots(x^3 - x^2 - x - 1)
((7/4, 19/8), 1)
```

```sage
glob sage: real_roots((x-1)*(x-2)*(x-3)*(x-5)*(x-8)*(x-13)*(x-21)*(x-34))
((11/16, 33/32), 1), ((11/8, 33/16), 1), ((11/4, 55/16), 1), ((77/16, 165/32), 1),
→((11/2, 33/4), 1), ((11/5, 55/4), 1), ((165/8, 341/16), 1), ((22, 44), 1)
```

```sage
glob sage: real_roots(x^5 * (x^2 - 9999)^2 - 1)
((-29274496381311/9007199254740992, 419601125186091/2251799813685248), 1),
→((2126658450145849495395106165441513249597/21267647932558653996460912964485513216,
→425331690727133001853696359533061621799/2453529586511730793292128529871026432),
→1), ((10633292262787402824531752558954186101/
→1063382396627932698323045682242756608, 531664614358685696701445201630854654353/
→531691198319663491615228241121378304), 1)
```

```sage
glob sage: real_roots(x^5 * (x^2 - 9999)^2 - 1, seed=42)
```

(continues on next page)
[((-123196838408289/180143985094819049184, 29396474345789/9007199254740992), 1), (...)

sage: real_roots(x^5 * (x^2 - 9999)^2 - 1, wordsize=64)
[((-62865630832021500000003/1934281311383406679528816, 901086456237337315214990987922889050101157/544424563237337315214990987922889050101157), 1), (...)

sage: v = 2^40
sage: real_roots((x^2-1)^2 * (x^2 - (v+1)/v))
[((-128555043540777682108850902171420745040205819129106632833/128555043540777682108850902171420745040205819129106632833, -64227551727783884104529501585875936959586605840418686045385497/64227551727783884104529501585875936959586605840418686045385497), 1), (...)

sage: real_roots(x^2 - 2)
[((-3/2, -1), 1), ((1, 3/2), 1)]

If the polynomial has no real roots, we get an empty list.

```
sage: (x^2 + 1).real_root_intervals()
[]
```

We can compute Conway’s constant (see http://mathworld.wolfram.com/ConwaysConstant.html) to arbitrary precision.

```
sage: p = x^71 - x^69 - 2*x^68 - x^67 + 2*x^66 + 2*x^65 + x^64 - x^63 - x^62 - x^61 - x^60 - x^59 + 2*x^58 + 5*x^57 + 3*x^56 - 2*x^55 - 10*x^54 - 3*x^53 - 2*x^52 - 6*x^51 + 6*x^50 + x^49 + 9*x^48 - 3*x^47 - 7*x^46 - 8*x^45 - 8*x^44 + 10*x^43 + 6*x^42 + 8*x^41 - 5*x^40 - 12*x^39 + 7*x^38 - 7*x^37 + 7*x^36 + x^35 - 3*x^34 + 10*x^33 + x^32 - 6*x^31 + 2*x^30 - 10*x^29 - 3*x^28 + 2*x^27 + 9*x^26 - 3*x^25 + 14*x^24 - 8*x^23 - 7*x^21 + 9*x^20 + 3*x^19 - 4*x^18 - 10*x^17 - 7*x^16 + 12*x^15 + 7*x^14 + 2*x^13 - 12*x^12 - 4*x^11 + 2*x^10 + 5*x^9 + 9*x^7 - 7*x^6 + 7*x^5 - 4*x^4 - 4 + 12*x^3 - 6*x^2 + 3*x - 6
sage: cc = real_roots(p, retval='algebraic_real')[2][0] # long time
sage: RealField(180)(cc) # long time
1.3035772690342963912570991121525518907307025046594049
```

Now we play with algebraic real coefficients.

```
sage: x = polygen(AA)
sage: p = (x - 1) * (x - sqrt(AA(2))) * (x - 2)
```

(continues on next page)
sage: real_roots(p)
[[(499/525, 2171/1925), 1], [(1173/875, 2521/1575), 1], [(337/175, 849/175), 1]]
sage: ar_rts = real_roots(p, retval='algebraic_real'); ar_rts
[(1.000000000000000?, 1), (1.414213562373095?, 1), (2.000000000000000?, 1)]
sage: ar_rts[1][0]^2 == 2
True
sage: ar_rts = real_roots(x*(x-1), retval='algebraic_real')
sage: ar_rts[0][0] == 0
True
sage: p2 = p * (p - 1/100); p2
x^6 - 8.82842712474619?*x^5 + 31.97056274847714?*x^4 - 60.77955262170047?*x^3 + 63. → 98526763257801?*x^2 - 35.37613490585595?*x + 8.028284271247462?
sage: real_roots(p2, retval='interval')
[(1.00?, 1), (1.1?, 1), (1.38?, 1), (1.5?, 1), (2.00?, 1), (2.1?, 1)]
sage: p = (x - 1) * (x - sqrt(AA(2)))^2 * (x - 2)^3 * sqrt(AA(3))
sage: real_roots(p, retval='interval')
[(1.000000000000000?, 1), (1.414213562373095?, 2), (2.000000000000000?, 3)]
sage: from sage.rings.polynomial.real_roots import *
sage: x = polygen(ZZ)
sage: root_bounds((x-1)*(x-2)*(x-3))
(0.545454545454545, 6.00000000000001)
sage: root_bounds(x^2)
(0.000000000000000, 0.000000000000000)
sage: root_bounds(x*(x+1))
(-1.00000000000000, 0.000000000000000)
sage: root_bounds((x+2)*(x-3))
(-2.44948974278317, 3.46410161513776)

If we can see that the polynomial has no real roots, return None.

sage: root_bounds(x^2 + 1)
is None
True

class sage.rings.polynomial.real_roots.rr_gap

A simple class representing the gaps between islands, in my ocean-island root isolation algorithm. Named "rr_gap" for "real roots gap", because "gap" seemed too short and generic.

region()

c.sage.rings.polynomial.real_roots.scale_intvec_var(c, k)

Given a vector of integers c of length n+1, and a rational k == kn / kd, multiplies each element c[i] by (kd^i)*(kn^(n-i)).

Modifies the input vector; has no return value.

EXAMPLES:

sage: from sage.rings.polynomial.real_roots import *
sage: v = vector(ZZ, [1, 1, 1, 1])
sage: scale_intvec_var(v, 3/4)
sage: v
(64, 48, 36, 27)

sage.rings.polynomial.real_roots.split_for_targets(ctx, bp, target_list, precise=False)

Given an interval Bernstein polynomial over a particular region (assumed to be a (not necessarily proper) sub-region of [0 .. 1]), and a list of targets, uses de Casteljau's method to compute representations of the Bernstein polynomial over each target. Uses degree reduction as often as possible while maintaining the requested precision.

Each target is of the form (lgap, ugap, b). Suppose lgap.region() is (l1, l2), and ugap.region() is (u1, u2). Then we will compute an interval Bernstein polynomial over a region [l .. u], where l1 <= l <= l2 and u1 <= u <= u2. (split_for_targets()) is free to select arbitrary region endpoints within these bounds; it picks endpoints which make the computation easier.) The third component of the target, b, is the maximum allowed scale_log2 of the result; this is used to decide when degree reduction is allowed.

The pair (l1, l2) can be replaced by None, meaning [-infinity .. 0]; or, (u1, u2) can be replaced by None, meaning [1 .. infinity].

216 Chapter 2. Univariate Polynomials
There is another constraint on the region endpoints selected by split_for_targets() for a target ((l1, l2), (u1, u2), b). We set a size goal g, such that (u - l) <= g * (u1 - l2). Normally g is 256/255, but if precise is True, then g is 65536/65535.

EXAMPLES:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpi([1000000, -2000000, 3000000, -4000000, -5000000, -6000000])
sage: ctx = mk_context()
sage: bps = split_for_targets(ctx, bp, [(rr_gap(1/1234567893, 1/1234567892, 1), rr_˓→gap(1/1234567891, 1/1234567890, 1), 12), (rr_gap(1/3, 1/2, -1), rr_gap(2/3, 3/4, -˓→1), 6)])
sage: bps[0]
<IBP: (999992, 999992, 999992) + [0 .. 15) over 8613397477114467984778830327/˓→10633823966279326983230456482242756608 .. 59190816802593439481383652749538294787/˓→730750818666545145910184216358141509827966271488]; level 2; slope_err 0.?e12>
sage: bps[1]
<IBP: (-1562500, -1875001, -2222223, -2592593, -2969137, -3337450) + [0 .. 4) over ˓→[1/2 .. 2863311531/4294967296]>
```

sage.rings.polynomial.real_roots.subsample_vec_doctest(a, slen, llen)

sage.rings.polynomial.real_roots.taylor_shift1_intvec(c)

Given a vector of integers c of length d+1, representing the coefficients of a degree-d polynomial p, modify the vector to perform a Taylor shift by 1 (that is, p becomes p(x+1)).

This is the straightforward algorithm, which is not asymptotically optimal.

Modifies the input vector; has no return value.

EXAMPLES:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: x = polygen(ZZ)
sage: p = (x-1)*(x-2)*(x-3)
sage: v = vector(ZZ, p.list())
sage: p, v
(x^3 - 6*x^2 + 11*x - 6, (-6, 11, -6, 1))
sage: taylor_shift1_intvec(v)
sage: p(x+1), v
(x^3 - 3*x^2 + 2*x, (0, 2, -3, 1))
```

sage.rings.polynomial.real_roots.to_bernstein(p, low=0, high=1, degree=None)

Given a polynomial p with integer coefficients, and rational bounds low and high, compute the exact rational Bernstein coefficients of p over the region [low .. high]. The optional parameter degree can be used to give a formal degree higher than the actual degree.

The return value is a pair (c, scale); c represents the same polynomial as p*scale. (If you only care about the roots of the polynomial, then of course scale can be ignored.)

EXAMPLES:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: x = polygen(ZZ)
sage: to_bernstein(x)
([0, 1], 1)
sage: to_bernstein(x, degree=5)
```

(continues on next page)
Given a polynomial \( p \) with rational coefficients, compute the exact rational Bernstein coefficients of \( p(x/(x+1)) \).

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: x = polygen(ZZ)
sage: to_bernstein_warp(1 + x + x^2 + x^3 + x^4 + x^5)
[1, 1/5, 1/10, 1/10, 1/5, 1]
```

### class `sage.rings.polynomial.real_roots.warp_map`(`neg``)

A class to map between original coordinates and ocean coordinates. If \( \text{neg} \) is False, then the original->ocean transform is \( x \rightarrow x/(x+1) \), and the ocean->original transform is \( x/(1-x) \); this maps between \([0 .. infinity]\) and \([0 .. 1]\). If \( \text{neg} \) is True, then the original->ocean transform is \( x \rightarrow -x/(1-x) \), and the ocean->original transform is the same thing: \(-x/(1-x)\). This maps between \([0 .. -infinity]\) and \([0 .. 1]\).

**from_ocean**( `region` )
**to_ocean**( `region` )

### sage.rings.polynomial.real_roots.wordsize_rational`(`a`, `b`, `wordsize``)`

Given rationals \( a \) and \( b \), selects a de Casteljau split point \( r \) between \( a \) and \( b \). An attempt is made to select an efficient split point (according to the criteria mentioned in the documentation for `de_casteljau_intvec`), with a bias towards split points near \( a \).

In full detail:

Takes as input two rationals, \( a \) and \( b \), such that \( 0 \leq a \leq 1 \), \( 0 \leq b \leq 1 \), and \( a \neq b \). Returns rational \( r \), such that \( a \leq r \leq b \) or \( b \leq r \leq a \). The denominator of \( r \) is a power of 2. Let \( m \) be \( \min(r, 1-r) \), \( nm \) be \( \text{numerator}(m) \), and \( dml \) be \( \log_2(\text{denominator}(m)) \). The return value \( r \) is taken from the first of the following classes to have any members between \( a \) and \( b \) (except that if \( a \leq 1/8 \), or \( 7/8 \leq a \), then class 2 is preferred to class 1).

1. \( dml < \text{wordsize} \)
2. \( \text{bitsize}(nm) \leq \text{wordsize} \)
3. \( \text{bitsize}(nm) \leq 2*\text{wordsize} \)
4. \( \text{bitsize}(nm) \leq 3*\text{wordsize} \)

...  
k. \( \text{bitsize}(nm) \leq (k-1)*\text{wordsize} \)

From the first class to have members between \( a \) and \( b \), \( r \) is chosen as the element of the class which is closest to \( a \).

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: wordsize_rational(1/5, 1/7, 32)
429496729/2147483648
```
2.1.19 Isolate Complex Roots of Polynomials

AUTHOR:

- Carl Witty (2007-11-18): initial version

This is an implementation of complex root isolation. That is, given a polynomial with exact complex coefficients, we compute isolating intervals for the complex roots of the polynomial. (Polynomials with integer, rational, Gaussian rational, or algebraic coefficients are supported.)

We use a simple algorithm. First, we compute a squarefree decomposition of the input polynomial; the resulting polynomials have no multiple roots. Then, we find the roots numerically, using NumPy (at low precision) or Pari (at high precision). Then, we verify the roots using interval arithmetic.

EXAMPLES:

```python
sage: x = polygen(ZZ)
```

```python
sage: (x^5 - x - 1).roots(ring=CIF)
```

```
[(1.167303978261419?, 1), (-0.764884433600585? - 0.352471546031727?*I, 1), (-0.764884433600585? + 0.352471546031727?*I, 1), (0.181232444469876? - 1.0839541013177117*I, 1), (0.181232444469876? + 1.0839541013177117*I, 1)]
```

```python
sage.rings.polynomial.complex_roots.complex_roots(p, skip_squarefree=False, retval='interval', min_prec=0)
```

Compute the complex roots of a given polynomial with exact coefficients (integer, rational, Gaussian rational, and algebraic coefficients are supported). Returns a list of pairs of a root and its multiplicity.

Roots are returned as a ComplexIntervalFieldElement; each interval includes exactly one root, and the intervals are disjoint.

By default, the algorithm will do a squarefree decomposition to get squarefree polynomials. The skip_squarefree parameter lets you skip this step. (If this step is skipped, and the polynomial has a repeated root, then the algorithm will loop forever!)

You can specify retval='interval' (the default) to get roots as complex intervals. The other options are retval='algebraic' to get elements of QQbar, or retval='algebraic_real' to get only the real roots, and to get them as elements of AA.

EXAMPLES:
```python
sage: from sage.rings.polynomial.complex_roots import complex_roots
sage: x = polygen(ZZ)
sage: complex_roots(x^5 - x - 1)
[(1.167303978261419?, 1), (-0.764884433600585? - 0.352471546031727?*I, 1), (-0.764884433600585? + 0.352471546031727?*I, 1), (0.18123244469876? - 1.089354101317711?*I, 1), (0.18123244469876? + 1.089354101317711?*I, 1)]
sage: v=complex_roots(x^2 + 27*x + 181)

Unfortunately due to numerical noise there can be a small imaginary part to each root depending on CPU, compiler, etc, and that affects the printing order. So we verify the real part of each root and check that the imaginary part is small in both cases:
```
```python
sage: v # random
[(-14.61803398874990?..., 1), (-12.3819660112501...? + 0.?e-27*I, 1)]
sage: sorted((v[0][0].real(),v[1][0].real()))
[-14.61803398874989?, -12.3819660112501...?]
sage: v[0][0].imag().upper() < 1e25
True
sage: v[1][0].imag().upper() < 1e25
True
```
```python
sage: K.<im> = QuadraticField(-1)
sage: eps = 1/2^100
sage: x = polygen(K)
sage: p = (x-1)*(x-1-eps)*(x-1+eps)*(x-1-eps*im)*(x-1+eps*im)
This polynomial actually has all-real coefficients, and is very, very close to (x-1)^5:
```
```python
sage: [RR(QQ(a)) for a in list(p - (x-1)^5)]
[3.87259191484932e-121, -3.87259191484932e-121]
sage: rts = complex_roots(p)
sage: [ComplexIntervalField(10)(rt[0] - 1) for rt in rts]
```
We can get roots either as intervals, or as elements of QQbar or AA.
```
```python
sage: p = (x^2 + x - 1)
sage: p = p * p(x*im)
sage: p
-x^4 + (im - 1)*x^3 + im*x^2 + (-im - 1)*x + 1
```
Two of the roots have a zero real component; two have a zero imaginary component. These zero components will be found slightly inaccurately, and the exact values returned are very sensitive to the (non-portable) results of NumPy. So we post-process the roots for printing, to get predictable doctest results.
```
```python
sage: def tiny(x):
....:     return x.contains_zero() and x.absolute_diameter() < 1e-14
sage: def smash(x):
....:     x = CIF(x[0]) # discard multiplicity
....:     if tiny(x.imag()): return x.real()
....:     if tiny(x.real()): return CIF(0, x.imag())
sage: rts = complex_roots(p)
sage: sorted(map(smash, rts))
(<type 'sage.rings.complex_interval.ComplexIntervalFieldElement'>, [-1.6180339887498957, -0.6180339887498957*I, 1.6180339887498957*I, 0.6180339887498957*I])
```
We are given a squarefree polynomial $p$, a list of estimated roots, and a precision. We attempt to verify that the estimated roots are in fact distinct roots of the polynomial, using interval arithmetic of precision $prec$. If we succeed, we return a list of intervals bounding the roots; if we fail, we return None.

**EXAMPLES:**

```python
sage: x = polygen(ZZ)
sage: p = x^3 - 1
sage: rts = [CC.zeta(3)^i for i in range(0, 3)]
sage: from sage.rings.polynomial.complex_roots import interval_roots
sage: interval_roots(p, rts, 53)
[1, -0.500000000000000? + 0.866025403784439?*I, -0.500000000000000? - 0.866025403784439?*I]
sage: interval_roots(p, rts, 200)
[1, -0.500000000000000000000000000000000000000000000000000000000000? + 0.86602540378443864673723170752936183471402626905190314027904?*I, -0.500000000000000000000000000000000000000000000000000000000000? - 0.86602540378443864673723170752936183471402626905190314027904?*I]
```

**sage.rings.polynomial.complex_roots.intervals_disjoint(intvs)**

Given a list of complex intervals, check whether they are pairwise disjoint.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.complex_roots import intervals_disjoint
sage: a = CIF(RIF(0, 3), 0)
sage: b = CIF(0, RIF(1, 3))
sage: c = CIF(RIF(1, 2), RIF(1, 2))
sage: d = CIF(RIF(2, 3), RIF(2, 3))
sage: intervals_disjoint([a, b, c, d])
False
sage: d2 = CIF(RIF(2, 3), RIF(2.001, 3))
sage: intervals_disjoint([a, b, c, d2])
True
```
2.1.20 Refine polynomial roots using Newton–Raphson

This is an implementation of the Newton–Raphson algorithm to approximate roots of complex polynomials. The implementation is based on interval arithmetic.

AUTHORS:

• Carl Witty (2007-11-18): initial version

```python
sage: from sage.rings.polynomial.refine_root import refine_root
sage: x = polygen(ZZ)
sage: p = x^9 - 1
sage: ip = CIF']['x'](p); ip
x^9 - 1
sage: ipd = CIF']['x'](p.derivative()); ipd
9*x^8
sage: irt = CIF(CC(cos(2*pi/9), sin(2*pi/9))); irt
0.76604444311897802? + 0.64278760968653926?*I
sage: ip(irt)
0.?e-14 + 0.?e-14*I
sage: ipd(irt)
6.89439998807080? - 5.78508848717885?*I
sage: refine_root(ip, ipd, irt, CIF)
0.766044443118978? + 0.642787609686540?*I
```

2.1.21 Ideals in Univariate Polynomial Rings

AUTHORS:

• David Roe (2009-12-14) – initial version.

```python
sage: sage.rings.polynomial.ideal.Ideal_ipoly_field(ring, gen)
Bases: sage.rings.ideal.Ideal_pid

An ideal in a univariate polynomial ring over a field.

```groebner_basis(algorithm=None)```

Return a Gröbner basis for this ideal.

The Gröbner basis has 1 element, namely the generator of the ideal. This trivial method exists for compatibility with multi-variate polynomial rings.

INPUT:

• algorithm – ignored

EXAMPLES:

```python
sage: R.<x> = QQ[]
sage: I = R.ideal([x^2 - 1, x^3 - 1])
sage: G = I.groebner_basis(); G
[x - 1]
sage: type(G)
<class 'sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic'>
sage: list(G)
[x - 1]
```

**residue_class_degree()**

Returns the degree of the generator of this ideal.

This function is included for compatibility with ideals in rings of integers of number fields.

**EXAMPLES:**

```python
sage: R.<t> = GF(5)[]
sage: P = R.ideal(t^4 + t + 1)
sage: P.residue_class_degree()
4
```

**residue_field(names=None, check=True)**

If this ideal is \( P \subset F_p[t] \), returns the quotient \( F_p[t]/P \).

**EXAMPLES:**

```python
sage: R.<t> = GF(17)[]; P = R.ideal(t^3 + 2*t + 9)
sage: k.<a> = P.residue_field(); k
Residue field in a of Principal ideal (t^3 + 2*t + 9) of Univariate Polynomial Ring in t over Finite Field of size 17
```

## 2.1.22 Quotients of Univariate Polynomial Rings

**EXAMPLES:**

```python
sage: R.<x> = QQ[]
sage: S = R.quotient(x**3-3*x+1, 'alpha')
sage: S.gen()**2 in S
True
sage: x in S
True
sage: S.gen() in R
False
sage: 1 in S
True
```

```python
class sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRingFactory
Bases: sage.structure.factory.UniqueFactory

Create a quotient of a polynomial ring.

INPUT:

* ring - a univariate polynomial ring
```

2.1. Univariate Polynomials and Polynomial Rings 223
• polynomial - an element of ring with a unit leading coefficient
• names - (optional) name for the variable

OUTPUT: Creates the quotient ring \( R/I \), where \( R \) is the ring and \( I \) is the principal ideal generated by polynomial.

EXAMPLES:

We create the quotient ring \( \mathbb{Z}[x]/(x^3 + 7) \), and demonstrate many basic functions with it:

```python
sage: Z = IntegerRing()
sage: R = PolynomialRing(Z, 'x'); x = R.gen()
sage: S = R.quotient(x^3 + 7, 'a'); a = S.gen()
sage: S
Univariate Quotient Polynomial Ring in a over Integer Ring with modulus x^3 + 7
sage: a^3
-7
sage: S.is_field()
False
sage: a in S
True
sage: x in S
True
sage: a in R
False
sage: S.polynomial_ring()
Univariate Polynomial Ring in x over Integer Ring
sage: S.modulus()
x^3 + 7
sage: S.degree()
3
```

We create the “iterated” polynomial ring quotient

\[
R = (\mathbb{F}_2[y]/(y^2 + y + 1))[x]/(x^3 - 5).
\]

```python
sage: A.<y> = PolynomialRing(GF(2)); A
Univariate Polynomial Ring in y over Finite Field of size 2 (using GF2X)
sage: B = A.quotient(y^2 + y + 1, 'y2'); B
Univariate Quotient Polynomial Ring in y2 over Finite Field of size 2 with modulus y^2 + y + 1
sage: C = PolynomialRing(B, 'x'); x=C.gen(); C
Univariate Polynomial Ring in x over Univariate Quotient Polynomial Ring in y2 over Finite Field of size 2 with modulus y^2 + y + 1
sage: R = C.quotient(x^3 - 5); R
Univariate Quotient Polynomial Ring in xbar over Univariate Quotient Polynomial Ring in y2 over Finite Field of size 2 with modulus y^2 + y + 1 with modulus x^3 + 1
```

Next we create a number field, but viewed as a quotient of a polynomial ring over \( \mathbb{Q} \):

```python
sage: R = PolynomialRing(RationalField(), 'x'); x = R.gen()
sage: S = R.quotient(x^3 + 2*x - 5, 'a')
sage: S
```

(continues on next page)
Univariate Quotient Polynomial Ring in a over Rational Field with modulus $x^3 + 2x - 5$

```
sage: S.is_field()
True
sage: S.degree()
3
```

There are conversion functions for easily going back and forth between quotients of polynomial rings over $\mathbb{Q}$ and number fields:

```
sage: K = S.number_field(); K
Number Field in a with defining polynomial $x^3 + 2x - 5$
sage: K.polynomial_quotient_ring()
Univariate Quotient Polynomial Ring in a over Rational Field with modulus $x^3 + 2x - 5$
```

The leading coefficient must be a unit (but need not be 1).

```
sage: R = PolynomialRing(Integers(9), 'x'); x = R.gen()
sage: S = R.quotient(2*x^4 + 2*x^3 + x + 2, 'a')
sage: S = R.quotient(3*x^4 + 2*x^3 + x + 2, 'a')
Traceback (most recent call last):
... TypeError: polynomial must have unit leading coefficient
```

Another example:

```
sage: R.<x> = PolynomialRing(IntegerRing())
sage: f = x^2 + 1
sage: R.quotient(f)
Univariate Quotient Polynomial Ring in xbar over Integer Ring with modulus $x^2 + 1$
```

This shows that the issue at trac ticket #5482 is solved:

```
sage: R.<x> = PolynomialRing(QQ)
sage: f = x^2 - 1
sage: R.quotient_by_principal_ideal(f)
Univariate Quotient Polynomial Ring in xbar over Rational Field with modulus $x^2 - 1$
```

create_key(ring, polynomial, names=None)

Return a unique description of the quotient ring specified by the arguments.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: PolynomialQuotientRing.create_key(R, x + 1)
(Univariate Polynomial Ring in x over Rational Field, x + 1, ('xbar',))
```

create_object(version, key)

Return the quotient ring specified by key.

EXAMPLES:
class sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_coercion

Bases: sage.structure.coerce_maps.DefaultConvertMap_unique

A coercion map from a PolynomialQuotientRing to a PolynomialQuotientRing that restricts to the coercion map on the underlying ring of constants.

EXAMPLES:

```python
sage: R.<x> = ZZ[]
sage: S.<x> = QQ[]
sage: f = S.quo(x^2 + 1).coerce_map_from(R.quo(x^2 + 1)); f
Coercion map:
    From: Univariate Quotient Polynomial Ring in xbar over Integer Ring with modulus x^2 + 1
    To:   Univariate Quotient Polynomial Ring in xbar over Rational Field with modulus x^2 + 1

is_injective()

Return whether this coercion is injective.

EXAMPLES:

If the modulus of the domain and the codomain is the same and the leading coefficient is a unit in the domain, then the map is injective if the underlying map on the constants is:

```python
sage: R.<x> = ZZ[]
sage: S.<x> = QQ[]
sage: f = S.quo(x^2 + 1).coerce_map_from(R.quo(x^2 + 1))
sage: f.is_injective()
True
```

is_surjective()

Return whether this coercion is surjective.

EXAMPLES:

If the underlying map on constants is surjective, then this coercion is surjective since the modulus of the codomain divides the modulus of the domain:

```python
sage: R.<x> = ZZ[]
sage: f = R.quo(x).coerce_map_from(R.quo(x^2))
sage: f.is_surjective()
True
```

If the modulus of the domain and the codomain is the same, then the map is surjective iff the underlying map on the constants is:

```python
sage: A.<a> = ZqCA(9)
sage: R.<x> = A[]
sage: S.<x> = A.fraction_field()[]
sage: f = S.quo(x^2 + 2).coerce_map_from(R.quo(x^2 + 2))
```
sage: f.is_surjective()
False

class sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_domain(ring, polynomial, name=None, category=None)

Bases: sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic,
sage.rings.ring.IntegralDomain

EXAMPLES:

sage: R.<x> = PolynomialRing(ZZ)
sage: S.<xbar> = R.quotient(x^2 + 1)
sage: S
Univariate Quotient Polynomial Ring in xbar over Integer Ring with modulus x^2 + 1
sage: loads(S.dumps()) == S
True
sage: loads(xbar.dumps()) == xbar
True

field_extension(names)

Takes a polynomial quotient ring, and returns a tuple with three elements: the NumberField defined by
the same polynomial quotient ring, a homomorphism from its parent to the NumberField sending the generators
to one another, and the inverse isomorphism.

OUTPUT:

• field
• homomorphism from self to field
• homomorphism from field to self

EXAMPLES:

sage: R.<x> = PolynomialRing(Rationals())
sage: S.<alpha> = R.quotient(x^3 - 2)
sage: F.<b>, f, g = S.field_extension()
sage: F
Number Field in b with defining polynomial x^3 - 2
sage: a = F.gen()
sage: f(alpha)
b
sage: g(a)
alpha

Note that the parent ring must be an integral domain:

sage: R.<x> = GF(25,'f25')['x']
sage: S.<a> = R.quo(x^3 - 2)
sage: F, g, h = S.field_extension('b')
Traceback (most recent call last):

AttributeError: 'PolynomialQuotientRing_generic_with_category' object has no attribute 'field_extension'

Over a finite field, the corresponding field extension is not a number field:

```
sage: R.<x> = GF(25, 'a')['x']
sage: S.<a> = R.quo(x^3 + 2*x + 1)
sage: F, g, h = S.field_extension('b')
sage: h(F.0^2 + 3)
a^2 + 3
sage: g(x^2 + 2)
b^2 + 2
```

We do an example involving a relative number field:

```
sage: R.<x> = QQ['x']
sage: K.<a> = NumberField(x^3 - 2)
sage: S.<X> = K['X']
sage: Q.<b> = S.quo(X^3 + 2*X + 1)
sage: Q.field_extension('b')
(Number Field in b with defining polynomial X^3 + 2*X + 1 over its base field, ..
... Defn: b |--> b,
       Relative number field morphism:
       From: Number Field in b with defining polynomial X^3 + 2*X + 1 over its base_field
       To:    Univariate Quotient Polynomial Ring in b over Number Field in a with_defining polynomial x^3 - 2 with modulus X^3 + 2*X + 1
       Defn: b |--> b
       a |--> a)
```

We slightly change the example above so it works.

```
sage: R.<x> = QQ['x']
sage: K.<a> = NumberField(x^3 - 2)
sage: S.<X> = K['X']
sage: f = (X+a)^3 + 2*(X+a) + 1
sage: f
X^3 + 3*a*X^2 + (3*a^2 + 2)*X + 2*a + 3
sage: Q.<z> = S.quo(f)
sage: F.<w>, g, h = Q.field_extension()
sage: c = g(z)
sage: f(c)
0
sage: h(g(z))
z
sage: g(h(w))
```

AUTHORS:
- Craig Citro (2006-08-07)
- William Stein (2006-08-06)
class sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_field(ring, polynomial, name=None, category=None)

Bases: sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_domain, sage.rings.ring.Field

EXAMPLES:

sage: R.<x> = PolynomialRing(QQ)
sage: S.<xbar> = R.quotient(x^2 + 1)
sage: S
Univariate Quotient Polynomial Ring in xbar over Rational Field with modulus x^2 + 1
sage: loads(S.dumps()) == S
True
sage: loads(xbar.dumps()) == xbar
True

base_field()
Alias for base_ring, when we're defined over a field.

complex_embeddings(prec=53)
Return all homomorphisms of this ring into the approximate complex field with precision prec.

EXAMPLES:

sage: R.<x> = QQ[]
sage: f = x^5 + x + 17
sage: k = R.quotient(f)
sage: v = k.complex_embeddings(100)
[2.9757207403766761469671194565, -2.4088994371613850098316292196 + 1.902541853025362812407363802*I, -2.4088994371613850098316292196 - 1.902541853025362812407363802*I, 0.9210390669730469364806949137 - 3.075533118845779473265418086*I, 0.9210390669730469364806949137 + 3.075533118845779473265418086*I]

class sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic(ring, polynomial, name=None, category=None)

Bases: sage.rings.ring.CommutativeRing

Quotient of a univariate polynomial ring by an ideal.

EXAMPLES:

sage: R.<x> = PolynomialRing(Integers(8)); R
Univariate Polynomial Ring in x over Ring of integers modulo 8
sage: S.<xbar> = R.quotient(x^2 + 1); S
Univariate Quotient Polynomial Ring in xbar over Ring of integers modulo 8 with modulus x^2 + 1
We demonstrate object persistence.

```
sage: loads(S.dumps()) == S
True
sage: loads(xbar.dumps()) == xbar
True
```

We create some sample homomorphisms:

```
sage: R.<x> = PolynomialRing(ZZ)
sage: S = R.quo(x^2-4)
sage: f = S.hom([2])
sage: f
Ring morphism:
  From: Univariate Quotient Polynomial Ring in xbar over Integer Ring with modulus x^2 - 4
  To: Integer Ring
  Defn: xbar |--> 2
sage: f(x)
2
sage: f(x^2 - 4)
0
sage: f(x^2)
4
```

**Element**

alias of `sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRingElement`

`S_class_group(S, proof=True)`

If self is an étale algebra $D$ over a number field $K$ (i.e. a quotient of $K[x]$ by a squarefree polynomial) and $S$ is a finite set of places of $K$, return a list of generators of the $S$-class group of $D$.

**NOTE:**

Since the `ideal` function behaves differently over number fields than over polynomial quotient rings (the quotient does not even know its ring of integers), we return a set of pairs `(gen, order)`, where `gen` is a tuple of generators of an ideal $I$ and `order` is the order of $I$ in the $S$-class group.

**INPUT:**

- `S` - a set of primes of the coefficient ring
- `proof` - if False, assume the GRH in computing the class group

**OUTPUT:**

A list of generators of the $S$-class group, in the form `(gen, order)`, where `gen` is a tuple of elements generating a fractional ideal $I$ and `order` is the order of $I$ in the $S$-class group.

**EXAMPLES:**

A trivial algebra over $\mathbb{Q}(\sqrt{-5})$ has the same class group as its base:

```
sage: K.<a> = QuadraticField(-5)
sage: R.<x> = K[]
sage: S.<xbar> = R.quotient(x)
sage: S.S_class_group([])
[((2, -a + 1), 2)]
```
When we include the prime \((2, -a + 1)\), the \(S\)-class group becomes trivial:

```
sage: S.S_class_group([K.ideal(2, -a+1)])
[]
```

Here is an example where the base and the extension both contribute to the class group:

```
sage: K.<a> = QuadraticField(-5)
sage: K.class_group()
Class group of order 2 with structure C2 of Number Field in a with defining polynomial \(x^2 + 5\) with \(a = 2.236067977499790?*I\)
sage: R.<x> = K[]
sage: S.<xbar> = R.quotient(x^2 + 23)
sage: S.S_class_group([])
[((2, -a + 1, 1/2*xbar + 1/2, -1/2*a*xbar + 1/2*a + 1), 6)]
sage: S.S_class_group([K.ideal(3, a-1)])
[]
sage: S.S_class_group([K.ideal(2, a+1)])
[]
sage: S.S_class_group([K.ideal(a)])
[((2, -a + 1, 1/2*xbar + 1/2, -1/2*a*xbar + 1/2*a + 1), 6)]
```

Now we take an example over a nontrivial base with two factors, each contributing to the class group:

```
sage: K.<a> = QuadraticField(-5)
sage: R.<x> = K[]
sage: S.<xbar> = R.quotient((x^2 + 23)*(x^2 + 31))
sage: S.S_class_group([])
# representation varies, not tested
[((1/4*xbar^2 + 31/4, (-1/8*a + 1/8)*xbar^2 - 31/8*a + 31/8, 1/16*xbar^3 + 1/16*xbar^2 + 31/16*xbar + 31/16, 6),
 (-1/4*xbar^2 - 23/4, (1/8*a - 1/8)*xbar^2 + 23/8*a - 23/8, -1/16*xbar^3 - 1/16*xbar^2 - 23/16*xbar - 23/16, 6),
 (-5/4*xbar^2 - 115/4, 1/4*a*xbar^2 + 23/4*a, -1/16*xbar^3 - 7/16*xbar^2 - 23/16*xbar - 161/16, 1/16*a*xbar^3 - 1/16*a*xbar^2 + 23/16*a*xbar - 23/16*a), 2)]
```

By using the ideal \((a)\), we cut the part of the class group coming from \(x^2 + 31\) from 12 to 2, i.e. we lose a generator of order 6 (this was fixed in trac ticket #14489):

```
sage: S.S_class_group([K.ideal(a)])
# representation varies, not tested
[((1/4*xbar^2 + 31/4, (-1/8*a + 1/8)*xbar^2 - 31/8*a + 31/8, 1/16*xbar^3 + 1/16*xbar^2 + 31/16*xbar + 31/16, 6),
 (-1/4*xbar^2 - 23/4, (1/8*a - 1/8)*xbar^2 + 23/8*a - 23/8, -1/16*xbar^3 - 1/16*xbar^2 - 23/16*xbar - 23/16, 6),
 (-5/4*xbar^2 - 115/4, 1/4*a*xbar^2 + 23/4*a, -1/16*xbar^3 - 7/16*xbar^2 - 23/16*xbar - 161/16, 1/16*a*xbar^3 - 1/16*a*xbar^2 + 23/16*a*xbar - 23/16*a), 2)]
```

Note that all the returned values live where we expect them to:
Sage: CG = S.S_class_group([])
Sage: type(CG[0][0][1])
<class 'sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic_with_category.element_class'>
Sage: type(CG[0][1])
<type 'sage.rings.integer.Integer'>

S_units(S, proof=True)
If self is an étale algebra $D$ over a number field $K$ (i.e. a quotient of $K[x]$ by a squarefree polynomial) and $S$ is a finite set of places of $K$, return a list of generators of the group of $S$-units of $D$.

INPUT:

• $S$ - a set of primes of the base field
• $proof$ - if False, assume the GRH in computing the class group

OUTPUT:

A list of generators of the $S$-unit group, in the form (gen, order), where gen is a unit of order order.

EXAMPLES:

sage: K.<a> = QuadraticField(-3)
sage: K.unit_group()
Unit group with structure C6 of Number Field in a with defining polynomial $x^2 + 3$ with a = 1.732050807568878?*I
sage: u,o = K.S_units([])[0]; o
6
sage: 2*u - 1 in {a, -a}
True
sage: u^6
1
sage: u^3
-1
sage: 2*u^2 + 1 in {a, -a}
True

sage: K.<a> = QuadraticField(-3)
sage: y = polygen(K)
sage: L.<b> = K[y].quotient(y^3 + 5); L
Univariate Quotient Polynomial Ring in b over Number Field in a with defining polynomial $x^2 + 3$ with a = 1.732050807568878?*I with modulus $y^3 + 5$
sage: [u for u, o in L.S_units([]) if o is Infinity]
[(-1/3*a - 1)*b^2 - 4/3*a*b - 5/6*a + 7/2, 2/3*a*b^2 + (2/3*a - 2)*b - 5/6*a - 7/2]
sage: [u for u, o in L.S_units([K.ideal(1/2*a - 3/2)]) if o is Infinity]
[(-1/3*a - 1/2)*b^2 + (1/3*a - 1)*b + 4/3*a, (-1/3*a - 1)*b^2 - 4/3*a*b - 5/6*a + 7/2, 2/3*a*b^2 + (2/3*a - 2)*b - 5/6*a - 7/2]
sage: [u for u, o in L.S_units([K.ideal(2)]) if o is Infinity]
[(1/2*a - 1/2)*b^2 + (a + 1)*b + 3, (1/6*a + 1/2)*b^2 + (-1/3*a + 1)*b - 5/6*a + 1/2, (1/6*a + 1/2)*b^2 + (-1/3*a + 1)*b - 5/6*a - 1/2,]
Note that all the returned values live where we expect them to:

```
sage: U = L.S_units([])
sage: type(U[0][0])
<class 'sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_field_with_category.element_class'>

sage: type(U[0][1])
<type 'sage.rings.integer.Integer'>

sage: type(U[1][1])
<class 'sage.rings.infinity.PlusInfinity'>
```

### ambient()

`ambient()`

Return the ambient ring of the polynomial ring, of which this ring is a quotient.

**EXAMPLES:**

The ambient ring of \( \mathbb{Z}[z]/(z^3 + z^2 + z + 1) \) is \( \mathbb{Z} \).

```
sage: R.<z> = PolynomialRing(ZZ)
sage: S.<beta> = R.quo(z^3 + z^2 + z + 1)
sage: S.ambient()
Integer Ring
```

Next we make a polynomial quotient ring over \( S \) and ask for its ambient ring.

```
sage: T.<t> = PolynomialRing(S)
sage: W = T.quotient(t^99 + 99)
sage: W.ambient()
Univariate Quotient Polynomial Ring in beta over Integer Ring with modulus z^3 + z^2 + z + 1
```

### cardinality()

`cardinality()`

Return the number of elements of this quotient ring.

`order` is an alias of `cardinality`.

**EXAMPLES:**

```
sage: R.<x> = ZZ[]
sage: R.quo(1).cardinality()
1
sage: R.quo(x^3-2).cardinality()
+Infinity

sage: R.quo(1).order()
1
sage: R.quo(x^3-2).order()
+Infinity
```
characteristic()

Return the characteristic of this quotient ring.

This is always the same as the characteristic of the base ring.

EXAMPLES:

```python
sage: R.<z> = PolynomialRing(ZZ)
sage: S.<a> = R.quo(z - 19)
sage: S.characteristic()
0
sage: R.<x> = PolynomialRing(GF(9,'a'))
sage: S = R.quotient(x^3 + 1)
sage: S.characteristic()
3
```
The same algebra constructed in a different way:

```sage
K.<a> = QQ['x'].quotient(x^2 + 5)
sage: K.class_group()
[[(2, a + 1), 2]]
```

Here is an example where the base and the extension both contribute to the class group:

```sage
K.<a> = QuadraticField(-5)
sage: K.class_group()
Class group of order 2 with structure C2 of Number Field in a with defining polynomial x^2 + 5 with a = 2.236067977499790?*I
sage: R.<x> = K[]
sage: S.<xbar> = R.quotient(x^2 + 23)
sage: S.class_group()
[[(2, -a + 1, 1/2*xbar + 1/2, -1/2*a*xbar + 1/2*a + 1), 6]]
```

Here is an example of a product of number fields, both of which contribute to the class group:

```sage
R.<x> = QQ[]
sage: S.<xbar> = R.quotient((x^2 + 23)*(x^2 + 47))
sage: S.class_group()
```

Now we take an example over a nontrivial base with two factors, each contributing to the class group:

```sage
K.<a> = QuadraticField(-5)
sage: R.<x> = K[]
sage: S.<xbar> = R.quotient((x^2 + 23)*(x^2 + 31))
sage: S.class_group()  # representation varies, not tested
[[(1/4*a*xbar^2 + 31/4,
  (-1/8*a + 1/8)*xbar^2 - 31/8*a + 31/8,
  1/16*xbar^3 + 1/16*xbar^2 + 31/16*xbar + 31/16,
  -1/16*a*xbar^3 + (1/16*a + 1/8)*xbar^2 - 31/16*a^2*xbar + 31/16*a + 31/8),
  6),
  ((-1/4*xbar^2 - 23/4,
    (1/8*a - 1/8)*xbar^2 + 23/8*a - 23/8,
    -1/16*xbar^3 - 1/16*xbar^2 - 23/16*xbar - 23/16,
    1/16*a*xbar^3 + (-1/16*a - 1/8)*xbar^2 + 23/16*a^2*xbar - 23/16*a - 23/8),
    6),
  ((-5/4*xbar^2 - 115/4,
    1/4*a*xbar^2 + 23/4*a,
    -1/16*xbar^3 - 7/16*xbar^2 - 23/16*xbar - 161/16,
    1/16*a*xbar^3 - 1/16*a^2*xbar^2 + 23/16*a^2*xbar - 23/16*a),
    2)]
```

Note that all the returned values live where we expect them to:

2.1. Univariate Polynomials and Polynomial Rings 235
construction()

Functorial construction of self

EXAMPLES:

```
sage: P.<t>=ZZ[]
sage: Q = P.quo(5+t^2)
sage: F, R = Q.construction()
sage: F(R) == Q
True
sage: P.<t> = GF(3)[]
sage: Q = P.quo([2+t^2])
sage: F, R = Q.construction()
sage: F(R) == Q
True
```

AUTHOR:

– Simon King (2010-05)

cover_ring()

Return the polynomial ring of which this ring is the quotient.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S = R.quotient(x^2-2)
sage: S.polynomial_ring()
Univariate Polynomial Ring in x over Rational Field
```

degree()

Return the degree of this quotient ring. The degree is the degree of the polynomial that we quotiented out by.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(GF(3))
sage: S = R.quotient(x^2005 + 1)
sage: S.degree()
2005
```

discriminant(v=None)

Return the discriminant of this ring over the base ring. This is by definition the discriminant of the polynomial that we quotiented out by.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S = R.quotient(x^3 + x^2 + x + 1)
```
The discriminant of the quotient polynomial ring need not equal the discriminant of the corresponding number field, since the discriminant of a number field is by definition the discriminant of the ring of integers of the number field:

```
sage: S = R.quotient((x + 1) * (x + 1))
sage: S.discriminant()
0
```

**gen($n=0$)**

Return the generator of this quotient ring. This is the equivalence class of the image of the generator of the polynomial ring.

**EXAMPLES:**

```
sage: R.<x> = PolynomialRing(QQ)
sage: S = R.quotient(x^2 - 8, 'gamma')
sage: S.gen()
gamma
```

**is_field($proof=True$)**

Return whether or not this quotient ring is a field.

**EXAMPLES:**

```
sage: R.<z> = PolynomialRing(ZZ)
sage: S = R.quo(z^2-2)
sage: S.is_field()
False
sage: R.<x> = PolynomialRing(QQ)
sage: S = R.quotient(x^2 - 2)
sage: S.is_field()
True
```

If proof is True, requires the is_irreducible method of the modulus to be implemented:

```
sage: R1.<x> = Qp(2)[]
sage: F1 = R1.quotient_ring(x^4+x+1)
sage: R2.<x> = F1[]
sage: F2 = R2.quotient_ring(x^2+x+1)
sage: F2.is_field()
Traceback (most recent call last):
...
NotImplementedError: cannot rewrite Univariate Quotient Polynomial Ring in xbar over 2-adic Field with capped relative precision 20 with modulus (1 + O(2^20))\*x^2 + (1 + O(2^20))\*x + 1 + O(2^20) as an isomorphic ring
sage: F2.is_field(proof = False)
```

---

2.1. Univariate Polynomials and Polynomial Rings 237
**is_finite()**

Return whether or not this quotient ring is finite.

**EXAMPLES:**

```python
sage: R.<x> = ZZ[]
sage: R.quo(1).is_finite()
True
sage: R.quo(x^3-2).is_finite()
False
```

```python
sage: R.<x> = GF(9, 'a')[]
sage: R.quo(2*x^3+x+1).is_finite()
True
sage: R.quo(2).is_finite()
True
```

```python
sage: P.<v> = GF(2)[]
sage: P.quotient(v^2-v).is_finite()
True
```

**krull_dimension()**

**lift(x)**

Return an element of the ambient ring mapping to the given argument.

**EXAMPLES:**

```python
sage: P.<x> = QQ[]
sage: Q = P.quotient(x^2+2)
sage: Q.lift(Q.0^3)
-2*x
sage: Q(-2*x)
-2*xbar
sage: Q.0^3
-2*xbar
```

**modulus()**

Return the polynomial modulus of this quotient ring.

**EXAMPLES:**

```python
sage: R.<x> = PolynomialRing(GF(3))
sage: S = R.quotient(x^2 - 2)
sage: S.modulus()
x^2 + 1
```

**ngens()**

Return the number of generators of this quotient ring over the base ring. This function always returns 1.

**EXAMPLES:**
sage: R.<x> = PolynomialRing(QQ)
sage: S.<y> = PolynomialRing(R)
sage: T.<z> = S.quotient(y + x)
sage: T
Univariate Quotient Polynomial Ring in z over Univariate Polynomial Ring in x
  over Rational Field with modulus y + x
sage: T.ngens()
1

**number_field()**
Return the number field isomorphic to this quotient polynomial ring, if possible.

**EXAMPLES:**

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<alpha> = R.quotient(x^29 - 17*x - 1)
sage: K = S.number_field()
sage: K
Number Field in alpha with defining polynomial x^29 - 17*x - 1
sage: alpha = K.gen()
sage: alpha^29
17*alpha + 1
```

**order()**
Return the number of elements of this quotient ring.
order is an alias of cardinality.

**EXAMPLES:**

```
sage: R.<x> = ZZ[]
sage: R.quo(1).cardinality()
1
sage: R.quo(x^3-2).cardinality()
+Infinity
sage: R.quo(1).order()
1
sage: R.quo(x^3-2).order()
+Infinity
```

```
sage: R.<x> = GF(9,'a')[]
sage: R.quo(2*x^3+x+1).cardinality()
729
sage: GF(9,'a').extension(2*x^3+x+1).cardinality()
729
sage: R.quo(2).cardinality()
1
```

**polynomial_ring()**
Return the polynomial ring of which this ring is the quotient.

**EXAMPLES:**
sage: R.<x> = PolynomialRing(QQ)
sage: S = R.quotient(x^2-2)
sage: S.polynomial_ring()
Univariate Polynomial Ring in x over Rational Field

random_element(*args, **kwds)
Return a random element of this quotient ring.

INPUT:
• *args, **kwds - Arguments for randomization that are passed on to the random_element method of the polynomial ring, and from there to the base ring

OUTPUT:
• Element of this quotient ring

EXAMPLES:
sage: F1.<a> = GF(2^7)
sage: P1.<x> = F1[]
sage: F2 = F1.extension(x^2+x+1, 'u')
sage: F2.random_element()
(a^6 + a^5 + a^2 + a)*u + a^6 + a^4 + a^3 + a^2 + 1

retract(x)
Return the coercion of x into this polynomial quotient ring.

The rings that coerce into the quotient ring canonically are:
• this ring
• any canonically isomorphic ring
• anything that coerces into the ring of which this is the quotient

selmer_generators(S, m, proof=True)
If self is an étale algebra $D$ over a number field $K$ (i.e. a quotient of $K[x]$ by a squarefree polynomial) and $S$ is a finite set of places of $K$, compute the Selmer group $D(S, m)$. This is the subgroup of $D^*/(D^*)^m$ consisting of elements $a$ such that $D(\sqrt[m]{a})/D$ is unramified at all primes of $D$ lying above a place outside of $S$.

INPUT:
• S - A set of primes of the coefficient ring (which is a number field).
• m - a positive integer
• proof - if False, assume the GRH in computing the class group

OUTPUT:
A list of generators of $D(S, m)$.

EXAMPLES:
sage: K.<a> = QuadraticField(-5)
sage: R.<x> = K[]
sage: D.<T> = R.quotient(x)
sage: D.selmer_generators(), 2)
[-1, 2]
sage: D.selmer_generators([K.ideal(2, -a+1)], 2)
[2, -1]
sage: D.selmer_generators([K.ideal(2, -a+1), K.ideal(3, a+1)], 2)
[2, a + 1, -1]
sage: D.selmer_generators((K.ideal(2, -a+1), K.ideal(3, a+1)), 4)
[2, a + 1, -1]
sage: D.selmer_generators([K.ideal(2, -a+1)], 3)
[2]
sage: D.selmer_generators([K.ideal(2, -a+1), K.ideal(3, a+1)], 3)
[2, a + 1]
sage: D.selmer_generators([K.ideal(2, -a+1), K.ideal(3, a+1), K.ideal(a)], 3)
[2, a + 1, a]

**selmer_group**(S, m, proof=True)

If self is an étale algebra D over a number field K (i.e. a quotient of K[x] by a squarefree polynomial) and S is a finite set of places of K, compute the Selmer group D(S, m). This is the subgroup of D*/(D*)^m consisting of elements a such that D((√a)/D is unramified at all primes of D lying above a place outside of S.

**INPUT:**

- S - A set of primes of the coefficient ring (which is a number field).
- m - a positive integer
- proof - if False, assume the GRH in computing the class group

**OUTPUT:**

A list of generators of D(S, m).

**EXAMPLES:**

```python
sage: K.<a> = QuadraticField(-5)
sage: R.<x> = K[]
sage: D.<T> = R.quotient(x)
sage: D.selmer_generators((), 2)
[-1, 2]
sage: D.selmer_generators([K.ideal(2, -a+1)], 2)
[2, -1]
sage: D.selmer_generators([K.ideal(2, -a+1), K.ideal(3, a+1)], 2)
[2, a + 1, -1]
sage: D.selmer_generators((K.ideal(2, -a+1), K.ideal(3, a+1)), 4)
[2, a + 1, -1]
sage: D.selmer_generators([K.ideal(2, -a+1)], 3)
[2]
sage: D.selmer_generators([K.ideal(2, -a+1), K.ideal(3, a+1)], 3)
[2, a + 1]
sage: D.selmer_generators([K.ideal(2, -a+1), K.ideal(3, 1), K.ideal(a)], 3)
[2, a + 1, a]
```

**units**(proof=True)

If this quotient ring is over a number field K, by a polynomial of nonzero discriminant, returns a list of generators of the units.

**INPUT:**
• **proof** - if False, assume the GRH in computing the class group

**OUTPUT:**

A list of generators of the unit group, in the form \((\text{gen, order})\), where \(\text{gen}\) is a unit of order \(\text{order}\).

**EXAMPLES:**

```
sage: K.<a> = QuadraticField(-3)
sage: K.unit_group()
Unit group with structure C6 of Number Field in a with defining polynomial x^2 + 3 with a = 1.732050807568878?*I
sage: K.<a> = QQ['x'].quotient(x^2 + 3)
sage: u = K.units()[0][0]
sage: 2*u - 1 in {a, -a}
True
sage: u^6
1
sage: u^3
-1
sage: 2*u^2 + 1 in {a, -a}
True
sage: K.<a> = QQ['x'].quotient(x^2 + 5)
sage: K.units()()
[(-1, 2)]
```

```
sage: K.<a> = QuadraticField(-3)
sage: y = polygen(K)
sage: L.<b> = K[y].quotient(y^3 + 5); L
Univariate Quotient Polynomial Ring in b over Number Field in a with defining polynomial x^2 + 3 with a = 1.732050807568878?*I with modulus y^3 + 5
sage: [u for u, o in L.units() if o is Infinity]
[(-1/3*a - 1)*b^2 - 4/3*a*b - 5/6*a + 7/2,
  2/3*a*b^2 + (2/3*a - 2)*b - 5/6*a - 7/2]
```

```
sage: K.<a> = QuadraticField(-3)
sage: y = polygen(K)
sage: L.<b> = K[y].quotient(y^3 + 5)
sage: U = L.units()
sage: type(U[0][0])
<class 'sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_field_with_category.element_class'>
sage: type(U[0][1])
<type 'sage.rings.integer.Integer'>
sage: type(U[1][1])
<class 'sage.rings.infinity.PlusInfinity'>
```

Note that all the returned values live where we expect them to:
2.1.23 Elements of Quotients of Univariate Polynomial Rings

EXAMPLES: We create a quotient of a univariate polynomial ring over $\mathbb{Z}$.

```
sage: R.<x> = ZZ[]
sage: S.<a> = R.quotient(x^3 + 3*x - 1)
sage: 2 * a^3
-6*a + 2
```

Next we make a univariate polynomial ring over $\mathbb{Z}[x]/(x^3 + 3x - 1)$.

```
sage: S1.<y> = S[]
```

And, we quotient out that by $y^2 + a$.

```
sage: T.<z> = S1.quotient(y^2+a)
```

In the quotient $z^2$ is $-a$.

```
sage: z^2
-a
```

And since $a^3 = -3x + 1$, we have:

```
sage: z^6
3*a - 1
```

For the purposes of comparison in Sage the quotient element $a^3$ is equal to $x^3$. This is because when the comparison is performed, the right element is coerced into the parent of the left element, and $x^3$ coerces to $a^3$.

```
sage: a == x
True
sage: a^3 == x^3
True
sage: x^3
x^3
sage: S(x^3)
2
```

AUTHORS:
class sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRingElement(parent, polynomial, check=True)


Element of a quotient of a polynomial ring.

EXAMPLES:

sage: P.<x> = QQ[]
sage: Q.<xi> = P.quo([(x^2+1)])
sage: xi^2
-1
sage: singular(xi)
xi
sage: (singular(xi)*singular(xi)).NF('std(0)')
-1

charpoly(var)

The characteristic polynomial of this element, which is by definition the characteristic polynomial of right multiplication by this element.

INPUT:

• var - string - the variable name

EXAMPLES:

sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quo(x^3 -389*x^2 + 2*x - 5)
sage: a.charpoly('X')
X^3 - 389*X^2 + 2*X - 5
sage: S(1).fcp('y')
(y - 1)^3

fcp(var='x')

Return the factorization of the characteristic polynomial of this element.

EXAMPLES:

sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quotient(x^3 -389*x^2 + 2*x - 5)
sage: a.fcp('x')
x^3 - 389*x^2 + 2*x - 5
sage: S(1).fcp('y')
(y - 1)^3

field_extension(names)

Given a polynomial with base ring a quotient ring, return a 3-tuple: a number field defined by the same polynomial, a homomorphism from its parent to the number field sending the generators to one another, and the inverse isomorphism.

INPUT:

• names - name of generator of output field

OUTPUT:
• field
• homomorphism from self to field
• homomorphism from field to self

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<alpha> = R.quotient(x^3-2)
sage: F.<a>, f, g = alpha.field_extension()
sage: F
Number Field in a with defining polynomial x^3 - 2
sage: a = F.gen()
sage: f(alpha)
a
sage: g(a)
alpha
```

Over a finite field, the corresponding field extension is not a number field:

```
sage: R.<x> = GF(25, 'b')['x']
sage: S.<a> = R.quo(x^3 + 2*x + 1)
sage: F.<b>, g, h = a.field_extension()
sage: h(b^2 + 3)
a^2 + 3
sage: g(x^2 + 2)
b^2 + 2
```

We do an example involving a relative number field:

```
sage: R.<x> = QQ['x']
sage: K.<a> = NumberField(x^3-2)
sage: S.<X> = K['X']
sage: Q.<b> = S.quo(X^3 + 2*X + 1)
sage: F, g, h = b.field_extension('c')
```

Another more awkward example:

```
sage: R.<x> = QQ['x']
sage: K.<a> = NumberField(x^3-2)
sage: S.<X> = K['X']
sage: f = (X+a)^3 + 2*(X+a) + 1
sage: f
X^3 + 3*a*X^2 + (3*a^2 + 2)*X + 2*a + 3
sage: Q.<z> = S.quo(f)
sage: F.<w>, g, h = z.field_extension()
sage: c = g(z)
sage: f(c)
0
sage: h(g(z))
z
sage: g(h(w))
w
```

AUTHORS:
is_unit()
Return True if self is invertible.

Warning: Only implemented when the base ring is a field.

EXAMPLES:

```python
sage: R.<x> = QQ[]
sage: S.<y> = R.quotient(x^2 + 2*x + 1)
sage: (2*y).is_unit()
True
sage: (y+1).is_unit()
False
```

lift()
Return lift of this polynomial quotient ring element to the unique equivalent polynomial of degree less than the modulus.

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quotient(x^3 - 2)
sage: b = a^2 - 3
sage: b
a^2 - 3
sage: b.lift()
x^2 - 3
```

list(copy=True)
Return list of the elements of self, of length the same as the degree of the quotient polynomial ring.

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quotient(x^3 + 2*x - 5)
sage: a^10
-134*a^2 - 35*a + 300
sage: (a^10).list()
[300, -35, -134]
```

matrix()
The matrix of right multiplication by this element on the power basis for the quotient ring.

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quotient(x^3 + 2*x - 5)
sage: a.matrix()
[ 0 1 0]
[ 0 0 1]
[ 5 -2 0]
```
minpoly()
The minimal polynomial of this element, which is by definition the minimal polynomial of right multipli-
cation by this element.

norm()
The norm of this element, which is the determinant of the matrix of right multiplication by this element.

EXAMPLES:

```sage
R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quotient(x^3 - 389*x^2 + 2*x - 5)
sage: a.norm()
5
```

trace()
The trace of this element, which is the trace of the matrix of right multiplication by this element.

EXAMPLES:

```sage
R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quotient(x^3 - 389*x^2 + 2*x - 5)
sage: a.trace()
389
```

2.1.24 Polynomial Compilers

AUTHORS:

- Tom Boothby, initial design & implementation
- Robert Bradshaw, bug fixes / suggested & assisted with significant design improvements

class sage.rings.polynomial.polynomial_compiled.CompiledPolynomialFunction
Bases: object

Builds a reasonably optimized directed acyclic graph representation for a given polynomial. A CompiledPoly-
nomialFunction is callable from python, though it is a little faster to call the eval function from pyrex.

This class is not intended to be called by a user, rather, it is intended to improve the performance of immutable
polynomial objects.

Todo:

- Recursive calling
- Faster casting of coefficients / argument
- Multivariate polynomials
- Cython implementation of Pippenger's Algorithm that doesn't depend heavily upon dicts.
- Computation of parameter sequence suggested by Pippenger
- Univariate exponentiation can use Brauer's method to improve extremely sparse polynomials of very high
degree

class sage.rings.polynomial.polynomial_compiled.abc_pd
Bases: sage.rings.polynomial.polynomial_compiled.binary_pd
2.1.25 Polynomial multiplication by Kronecker substitution

2.2 Generic Convolution

Asymptotically fast convolution of lists over any commutative ring in which the multiply-by-two map is injective. (More precisely, if \( x \in R \) and \( x = 2^k \cdot y \) for some \( k \geq 0 \), we require that \( R(x/2^k) \) returns \( y \).)

The main function to be exported is convolution().

EXAMPLES:

```python
sage: convolution([1, 2, 3, 4, 5], [6, 7])
[6, 19, 32, 45, 58, 35]
```

The convolution function is reasonably fast, even though it is written in pure Python. For example, the following takes less than a second:

```python
sage: v = convolution(list(range(1000)), list(range(1000)))
```

ALGORITHM: Converts the problem to multipicication in the ring \( S[x]/(x^M - 1) \), where \( S = R[y]/(y^K + 1) \) (where \( R \) is the original base ring). Performs FFT with respect to the roots of unity \( 1, y, y^2, \ldots, y^{2K-1} \) in \( S \). The FFT/IFFT are accomplished with just additions and subtractions and rotating python lists. (I think this algorithm is essentially due to Schonhage, not completely sure.) The pointwise multiplications are handled recursively, switching to a classical algorithm at some point.

Complexity is \( O(n \log(n) \log(\log(n))) \) additions/subtractions in \( R \) and \( O(n \log(n)) \) multiplications in \( R \).
AUTHORS:

- David Harvey (2007-07): first implementation
- William Stein: editing the docstrings for inclusion in Sage.

sage.rings.polynomial.convolution

\texttt{convolution(L1, L2)}

Returns convolution of non-empty lists \(L1\) and \(L2\). \(L1\) and \(L2\) may have arbitrary lengths.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: convolution([1, 2, 3], [4, 5, 6, 7]) [4, 13, 28, 34, 32, 21]
sage: R = Integers(47)
sage: L1 = [R.random_element() for _ in range(1000)]
sage: L2 = [R.random_element() for _ in range(3756)]
sage: L3 = convolution(L1, L2)
sage: len(L3) == 1000 + 3756 - 1 True
\end{verbatim}

2.3 Fast calculation of cyclotomic polynomials

This module provides a function \texttt{cyclo} \texttt{tomic_coeffs()}, which calculates the coefficients of cyclotomic polynomials. This is not intended to be invoked directly by the user, but it is called by the method \texttt{cyclotomic_polynomial()} method of univariate polynomial ring objects and the top-level \texttt{cyclotomic_polynomial()} function.

sage.rings.polynomial.cyclotomic

\texttt{bateman_bound(nn)}

Reference:

Bateman, P. T.; Pomerance, C.; Vaughan, R. C. \textit{On the size of the coefficients of the cyclotomic polynomial}.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.rings.polynomial.cyclotomic import bateman_bound
sage: bateman_bound(2**8*1234567893377) 66944986927
\end{verbatim}

sage.rings.polynomial.cyclotomic

\texttt{cyclo} \texttt{tomic_coeffs(nn, sparse=None)}

Return the coefficients of the \(n\)-th cyclotomic polynomial by using the formula

\[ \Phi_n(x) = \prod_{d|n} (1 - x^{n/d}) \mu(d) \]

where \(\mu(d)\) is the Möbius function that is 1 if \(d\) has an even number of distinct prime divisors, -1 if it has an odd number of distinct prime divisors, and 0 if \(d\) is not squarefree.

Multiplications and divisions by polynomials of the form \(1 - x^n\) can be done very quickly in a single pass.

If \texttt{sparse} is \texttt{True}, the result is returned as a dictionary of the non-zero entries, otherwise the result is returned as a list of python ints.

\textbf{EXAMPLES:}
```python
sage: from sage.rings.polynomial.cyclotomic import cyclotomic_coeffs
sage: cyclotomic_coeffs(30)
[1, 1, 0, -1, -1, -1, 0, 1, 1]
sage: cyclotomic_coeffs(10^5)
{0: 1, 10000: -1, 20000: 1, 30000: -1, 40000: 1}
sage: R = QQ['x']
sage: R(cyclotomic_coeffs(30))
x^8 + x^7 - x^5 - x^4 - x^3 + x + 1
```

Check that it has the right degree:

```python
sage: euler_phi(30)
8
sage: R(cyclotomic_coeffs(14)).factor()
x^6 - x^5 + x^4 - x^3 + x^2 - x + 1
```

The coefficients are not always +/-1:

```python
sage: cyclotomic_coeffs(105)
[1, 1, 1, 0, 0, -1, -1, -2, -1, -1, 0, 1, 1, 1, 1, 1, 0, 0, -1, 0, -1, 0, -1, ...
˓→0, -1, 0, -1, 0, 0, 1, 1, 1, 1, 1, 0, 0, -1, -1, -2, -1, -1, 0, 0, 1, 1, 1]
```

In fact the height is not bounded by any polynomial in n (Erdos), although takes a while just to exceed linear:

```python
sage: v = cyclotomic_coeffs(1181895)
sage: max(v)
14102773
```

The polynomial is a palindrome for any n:

```python
sage: n = ZZ.random_element(50000)
sage: factor(n)
3 * 10009
sage: v = cyclotomic_coeffs(n, sparse=False)
sage: v == list(reversed(v))
True
```

AUTHORS:

- Robert Bradshaw (2007-10-27): initial version (inspired by work of Andrew Arnold and Michael Monagan)

REFERENCE:

- [http://www.cecm.sfu.ca/~ada26/cyclotomic/](http://www.cecm.sfu.ca/~ada26/cyclotomic/)

sage.rings.polynomial.cyclotomic.

cyclotomic_value(n, x)

Return the value of the \( n \)-th cyclotomic polynomial evaluated at \( x \).

INPUT:

- \( n \) – an Integer, specifying which cyclotomic polynomial is to be evaluated
- \( x \) – an element of a ring

OUTPUT:

- the value of the cyclotomic polynomial \( \Phi_n \) at \( x \)

ALGORITHM:
• Reduce to the case that $n$ is squarefree: use the identity
\[ \Phi_n(x) = \Phi_q(x^{n/q}) \]
where $q$ is the radical of $n$.

• Use the identity
\[ \Phi_n(x) = \prod_{d|n} (x^d - 1)^{\mu(n/d)}, \]
where $\mu$ is the Möbius function.

• Handles the case that $x^d = 1$ for some $d$, but not the case that $x^d - 1$ is non-invertible: in this case polynomial evaluation is used instead.

EXAMPLES:

```
sage: cyclotomic_value(51, 3)
1282860140677441
sage: cyclotomic_polynomial(51)(3)
1282860140677441
```

It works for non-integral values as well:

```
sage: cyclotomic_value(144, 4/3)
7914874543504023621920372161/79766443076872509863361
sage: cyclotomic_polynomial(144)(4/3)
7914874543504023621920372161/79766443076872509863361
```
3.1 Multivariate Polynomials and Polynomial Rings

Sage implements multivariate polynomial rings through several backends. The most generic implementation uses the classes `sage.rings.polynomial.polydict.PolyDict` and `sage.rings.polynomial.polydict.ETuple` to construct a dictionary with exponent tuples as keys and coefficients as values.

Additionally, specialized and optimized implementations over many specific coefficient rings are implemented via a shared library interface to SINGULAR; and polynomials in the boolean polynomial ring

\[ F_2[x_1, ..., x_n]/(x_1^2 + x_1, ..., x_n^2 + x_n). \]

are implemented using the PolyBoRi library (cf. `sage.rings.polynomial.pbori`).

3.1.1 Term orders

Sage supports the following term orders:

**Lexicographic (lex)** \( x^a < x^b \) if and only if there exists \( 1 \leq i \leq n \) such that \( a_1 = b_1, \ldots, a_{i-1} = b_{i-1}, a_i < b_i \). This term order is called ‘lp’ in Singular.

**EXAMPLES:**

```
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order='lex')
sage: x > y
True
sage: x > y^2
True
sage: x > 1
True
sage: x^1*y^2 > y^3*z^4
True
sage: x^3*y^2*z^4 < x^3*y^2*z^1
False
```

**Degree reverse lexicographic (degrevlex)** Let \( \deg(x^a) = a_1 + a_2 + \cdots + a_n \), then \( x^a < x^b \) if and only if \( \deg(x^a) < \deg(x^b) \) or \( \deg(x^a) = \deg(x^b) \) and there exists \( 1 \leq i \leq n \) such that \( a_n = b_n, \ldots, a_{i+1} = b_{i+1}, a_i > b_i \). This term order is called ‘dp’ in Singular.

**EXAMPLES:**
In the SageMath system, polynomials are handled with the `PolynomialRing` class. For instance, creating a polynomial ring in three variables over the rational numbers with the degree reverse lexicographic order can be done as follows:

```sage
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order='degrevlex')
sage: x > y
True
sage: x > y^2*z
False
sage: x > 1
True
sage: x^1*y^5*z^2 > x^4*y^1*z^3
True
sage: x^2*y*z^2 > x*y^3*z
False
```

**Degree lexicographic (deglex)** Let \( \deg(x^a) = a_1 + a_2 + \cdots + a_n \), then \( x^a < x^b \) if and only if \( \deg(x^a) < \deg(x^b) \) or \( \deg(x^a) = \deg(x^b) \) and there exists \( 1 \leq i \leq n \) such that \( a_1 = b_1, \ldots, a_{i-1} = b_{i-1}, a_i < b_i \). This term order is called ‘Dp’ in Singular.

**EXAMPLES:**

```sage
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order='deglex')
sage: x > y
True
sage: x > y^2*z
False
sage: x > 1
True
sage: x^1*y^5*z^2 > x^4*y^1*z^3
True
sage: x^2*y*z^2 > x*y^3*z
False
```

**Inverse lexicographic (invlex)** \( x^a < x^b \) if and only if there exists \( 1 \leq i \leq n \) such that \( a_n = b_n, \ldots, a_{i+1} = b_{i+1}, a_i < b_i \). This order is called ‘rp’ in Singular.

**EXAMPLES:**

```sage
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order='invlex')
sage: x > y
False
sage: y > x^2
True
sage: x > 1
True
sage: x*y > z
False
```

This term order only makes sense in a non-commutative setting because if \( P \) is the ring \( k[x_1, \ldots, x_n] \) and term order ‘invlex’ then it is equivalent to the ring \( k[x_n, \ldots, x_1] \) with term order ‘lex’.

**Negative lexicographic (neglex)** \( x^a < x^b \) if and only if there exists \( 1 \leq i \leq n \) such that \( a_1 = b_1, \ldots, a_{i-1} = b_{i-1}, a_i > b_i \). This term order is called ‘ls’ in Singular.

**EXAMPLES:**

```sage
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order='neglex')
sage: x > y
```

(continues on next page)
**Negative degree reverse lexicographic (negdegrevlex)** Let \( \deg(x^a) = a_1 + a_2 + \cdots + a_n \), then \( x^a < x^b \) if and only if \( \deg(x^a) > \deg(x^b) \) or \( \deg(x^a) = \deg(x^b) \) and there exists \( 1 \leq i \leq n \) such that \( a_n = b_n, \ldots, a_{i+1} = b_{i+1}, a_i > b_i \). This term order is called ‘ds’ in Singular.

**EXAMPLES:**

```python
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order='negdegrevlex')
sage: x > y
True
sage: x > x^2
True
sage: x > 1
False
sage: x^1*y^2 > y^3*z^4
True
sage: x^2*y*z^2 > x*y^3*z
False
```

**Negative degree lexicographic (negdeglex)** Let \( \deg(x^a) = a_1 + a_2 + \cdots + a_n \), then \( x^a < x^b \) if and only if \( \deg(x^a) > \deg(x^b) \) or \( \deg(x^a) = \deg(x^b) \) and there exists \( 1 \leq i \leq n \) such that \( a_1 = b_1, \ldots, a_{i-1} = b_{i-1}, a_i < b_i \). This term order is called ‘Ds’ in Singular.

**EXAMPLES:**

```python
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order='negdeglex')
sage: x > y
True
sage: x > x^2
True
sage: x > 1
False
sage: x^1*y^2 > y^3*z^4
True
sage: x^2*y*z^2 > x*y^3*z
True
```

**Weighted degree reverse lexicographic (wdegrevlex), positive integral weights** Let \( \deg_w(x^a) = a_1 w_1 + a_2 w_2 + \cdots + a_n w_n \) with weights \( w \), then \( x^a < x^b \) if and only if \( \deg_w(x^a) > \deg_w(x^b) \) or \( \deg_w(x^a) = \deg_w(x^b) \) and there exists \( 1 \leq i \leq n \) such that \( a_n = b_n, \ldots, a_{i+1} = b_{i+1}, a_i > b_i \). This term order is called ‘wp’ in Singular.

**EXAMPLES:**

```python
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order=TermOrder('wdegrevlex',(1,2,3)))
sage: x > y
```

(continues on next page)
Weighted degree lexicographic (wdeglex), positive integral weights

\[
\text{Let } \deg_w(x^a) = a_1 w_1 + a_2 w_2 + \cdots + a_n w_n \\
\text{with weights } w, \text{ then } x^a < x^b \text{ if and only if } \deg_w(x^a) < \deg_w(x^b) \text{ or } \deg_w(x^a) = \deg_w(x^b) \text{ and there exists } 1 \leq i \leq n \text{ such that } a_i = b_i, \ldots, a_{i-1} = b_{i-1}, a_i < b_i. \text{ This term order is called ‘Wp’ in Singular.}
\]

**EXAMPLES:**

```
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order=TermOrder('wdeglex',(1,2,3)))
sage: x > y  # False
sage: x > x^2  # False
sage: x > 1  # True
sage: x^1*y^2 > x^2*z  # False
sage: y*z > x^3*y  # False
```

Negative weighted degree reverse lexicographic (negwdegrevlex), positive integral weights

\[
\text{Let } \deg_w(x^a) = a_1 w_1 + a_2 w_2 + \cdots + a_n w_n \text{ with weights } w, \text{ then } x^a < x^b \text{ if and only if } \deg_w(x^a) > \deg_w(x^b) \text{ or } \deg_w(x^a) = \deg_w(x^b) \text{ and there exists } 1 \leq i \leq n \text{ such that } a_n = b_n, \ldots, a_{i+1} = b_{i+1}, a_i > b_i. \text{ This term order is called ‘ws’ in Singular.}
\]

**EXAMPLES:**

```
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order=TermOrder('negwdegrevlex',(1,2,3)))
sage: x > y  # True
sage: x > x^2  # True
sage: x > 1  # False
sage: x^1*y^2 > x^2*z  # True
sage: y*z > x^3*y  # False
```

Degree negative lexicographic (degneglex)

\[
\text{Let } \deg(x^a) = a_1 + a_2 + \cdots + a_n, \text{ then } x^a < x^b \text{ if and only if } \deg(x^a) < \deg(x^b) \text{ or } \deg(x^a) = \deg(x^b) \text{ and there exists } 1 \leq i \leq n \text{ such that } a_1 = b_1, \ldots, a_{i-1} = b_{i-1}, a_i > b_i. \text{ This term order is called ‘dp_asc’ in PolyBoRi. Singular has the extra weight vector ordering (a(1:n),ls) for this purpose.}
\]

**EXAMPLES:**
Negative weighted degree lexicographic (negwdeglex), positive integral weights

Let \( \deg_w(x^a) = a_1 w_1 + a_2 w_2 + \cdots + a_n w_n \) with weights \( w \), then \( x^a < x^b \) if and only if \( \deg_w(x^a) > \deg_w(x^b) \) or \( \deg_w(x^a) = \deg_w(x^b) \) and there exists \( 1 \leq i \leq n \) such that \( a_1 = b_1, \ldots, a_{i-1} = b_{i-1}, a_i < b_i \). This term order is called ‘Ws’ in Singular.

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order=TermOrder('negwdeglex',(1,2,3)))
sage: x > y
True
sage: x > x^2
True
sage: x > 1
False
sage: x^1*y^2 > x^2*z
False
sage: y*z > x^3*y
False
```

Of these, only ‘degrevlex’, ‘deglex’, ‘degneglex’, ‘wdegrevlex’, ‘wdeglex’, ‘invlex’ and ‘lex’ are global orders.

Sage also supports matrix term order. Given a square matrix \( A \),

\[ x^a <_A x^b \] if and only if \( Aa < Ab \)

where \(<\) is the lexicographic term order.

EXAMPLES:

```
sage: m = matrix(2,[2,3,0,1]); m
[2 3]
[0 1]
sage: T = TermOrder(m); T
Matrix term order with matrix
[2 3]
[0 1]
sage: P.<a,b> = PolynomialRing(QQ,2,order=T)
sage: P
Multivariate Polynomial Ring in a, b over Rational Field
sage: a > b
False
sage: a^3 < b^2
True
sage: S = TermOrder('M(2,3,0,1)')
sage: T == S
True
```

Additionally all these monomial orders may be combined to product or block orders, defined as:

Let \( x = (x_1, x_2, \ldots, x_n) \) and \( y = (y_1, y_2, \ldots, y_m) \) be two ordered sets of variables, \(<_1 \) a monomial order on \( k[x] \) and \(<_2 \) a monomial order on \( k[y] \).
The product order (or block order) \( < := (<_1, <_2) \) on \( k[x,y] \) is defined as: \( x^a y^b < x^A y^B \) if and only if \( x^a <_1 x^A \) or \( (x^a = x^A \text{ and } y^b <_2 y^B) \).

These block orders are constructed in Sage by giving a comma separated list of monomial orders with the length of each block attached to them.

**EXAMPLES:**

As an example, consider constructing a block order where the first four variables are compared using the degree reverse lexicographical order while the last two variables in the second block are compared using negative lexicographical order.

```python
sage: P.<a,b,c,d,e,f> = PolynomialRing(QQ, 6, order='degrevlex(4),neglex(2)')
sage: a > c^4
False
sage: a > e^4
True
sage: e > f^2
False
```

The same result can be achieved by:

```python
sage: T1 = TermOrder('degrevlex',4)
sage: T2 = TermOrder('neglex',2)
sage: T = T1 + T2
sage: P.<a,b,c,d,e,f> = PolynomialRing(QQ, 6, order=T)
sage: a > c^4
False
sage: a > e^4
True
```

If any other unsupported term order is given the provided string can be forced to be passed through as is to Singular, Macaulay2, and Magma. This ensures that it is for example possible to calculate a Groebner basis with respect to some term order Singular supports but Sage doesn’t:

```python
sage: T = TermOrder("royalorder")
Traceback (most recent call last):
 ... ValueError: unknown term order 'royalorder'
sage: T = TermOrder("royalorder",force=True)
sage: T
royalorder term order
terms
'singular_str()
'royalorder'
```

**AUTHORS:**

- David Joyner and William Stein: initial version of multi_polynomial_ring
- Kiran S. Kedlaya: added macaulay2 interface
- Martin Albrecht: implemented native term orders, refactoring
- Kwankyu Lee: implemented matrix and weighted degree term orders
- Simon King (2011-06-06): added termorder_from_singular

```python
class sage.rings.polynomial.term_order.TermOrder(name='lex', n=0, force=False)
Bases: sage.structure.sage_object.SageObject
A term order.
```
See `sage.rings.polynomial.term_order` for details on supported term orders.

**blocks()**
Return the term order blocks of self.

**NOTE:**
This method has been added in [trac ticket #11316](https://trac.sagemath.org/ticket/11316). There used to be an *attribute* of the same name and the same content. So, it is a backward incompatible syntax change.

**EXAMPLES:**
```
sage: t=TermOrder('deglex',2)+TermOrder('lex',2)
sage: t.blocks()
(Degree lexicographic term order, Lexicographic term order)
```

**greater_tuple**
The default `greater_tuple` method for this term order.

**EXAMPLES:**
```
sage: O = TermOrder()
sage: O.greater_tuple.__func__ is O.greater_tuple_lex.__func__
True
sage: O = TermOrder('deglex')
sage: O.greater_tuple.__func__ is O.greater_tuple_deglex.__func__
True
```

**greater_tuple_block**(*f*, *g*)
Return the greater exponent tuple with respect to the block order as specified when constructing this element.

This method is called by the `lm/lc/lt` methods of `MPolynomial_polydict`.

**INPUT:**
- *f* - exponent tuple
- *g* - exponent tuple

**EXAMPLES:**
```
sage: P.<a,b,c,d,e,f>=PolynomialRing(QQbar, 6, order='degrevlex(3),degrevlex(3) →')
sage: f = a + c^4; f.lm() # indirect doctest
c^4
sage: g = a + e^4; g.lm()
a
```

**greater_tuple_deglex**(*f*, *g*)
Return the greater exponent tuple with respect to the total degree lexicographical term order.

**INPUT:**
- *f* - exponent tuple
- *g* - exponent tuple

**EXAMPLES:**
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order='deglex')
sage: f = x + y; f.lm() # indirect doctest
x
sage: f = x + y^2*z; f.lm()
y^2*z

This method is called by the lm/lc/lt methods of MPolynomial_polydict.

greater_tuple_degrevlex(f, g)
Return the greater exponent tuple with respect to the total degree reversed lexicographical term order.

INPUT:
- f - exponent tuple
- g - exponent tuple

EXAMPLES:

sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order='degrevlex')
sage: f = x + y; f.lm() # indirect doctest
x
sage: f = x + y^2*z; f.lm()
y^2*z

This method is called by the lm/lc/lt methods of MPolynomial_polydict.

greater_tuple_invlex(f, g)
Return the greater exponent tuple with respect to the inversed lexicographical term order.

INPUT:
- f - exponent tuple
- g - exponent tuple

EXAMPLES:

sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order='invlex')
sage: f = x + y; f.lm() # indirect doctest
y
sage: f = x + y^2*z; f.lm()
y^2*z

(continues on next page)
This method is called by the lm/lc/lc methods of MPolynomial_polydict.

**greater_tuple_lex**\( (f, g) \)

Return the greater exponent tuple with respect to the lexicographical term order.

**INPUT:**
- \( f \) - exponent tuple
- \( g \) - exponent tuple

**EXAMPLES:**

```
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order='lex')
sage: f = x + y^2; f.lm() # indirect doctest
x
```

This method is called by the lm/lc/lc methods of MPolynomial_polydict.

**greater_tuple_matrix**\( (f, g) \)

Return the greater exponent tuple with respect to the matrix term order.

**INPUT:**
- \( f \) - exponent tuple
- \( g \) - exponent tuple

**EXAMPLES:**

```
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='m(1,3,1,0)')
sage: y > x^2 # indirect doctest
True
sage: y > x^3
False
```

**greater_tuple_negdeglex**\( (f, g) \)

Return the greater exponent tuple with respect to the negative degree lexicographical term order.

**INPUT:**
- \( f \) - exponent tuple
- \( g \) - exponent tuple

**EXAMPLES:**

```
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order='negdeglex')
sage: f = x + y; f.lm() # indirect doctest
x
sage: f = x + x^2; f.lm()
x
sage: f = x^2*y*z^2 + x*y^3*z; f.lm()
x^2*y*z^2
```

This method is called by the lm/lc/lc methods of MPolynomial_polydict.
\textbf{greater_tuple_negdegrevlex}\((f, g)\)

Return the greater exponent tuple with respect to the negative degree reverse lexicographical term order.

\textbf{INPUT}:

\begin{itemize}
  \item \textit{f} - exponent tuple
  \item \textit{g} - exponent tuple
\end{itemize}

\textbf{EXAMPLES}:

\begin{verbatim}
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order='negdegrevlex')
sage: f = x + y; f.lm() # indirect doctest
  x
sage: f = x + x^2; f.lm()  
x
sage: f = x^2*y*z^2 + x*y^3*z; f.lm()
x*y^3*z
\end{verbatim}

This method is called by the \texttt{lm/lc/lt} methods of \texttt{MPolynomial_polydict}.

\textbf{greater_tuple_neglex}\((f, g)\)

Return the greater exponent tuple with respect to the negative lexicographical term order.

This method is called by the \texttt{lm/lc/lt} methods of \texttt{MPolynomial_polydict}.

\textbf{INPUT}:

\begin{itemize}
  \item \textit{f} - exponent tuple
  \item \textit{g} - exponent tuple
\end{itemize}

\textbf{EXAMPLES}:

\begin{verbatim}
sage: P.<a,b,c,d,e,f> = PolynomialRing(QQbar, 6, order='degrevlex(3),degrevlex(3) \rightarrow')
sage: f = a + c^4; f.lm() # indirect doctest
c^4
sage: g = a + e^4; g.lm()  
a
\end{verbatim}

\textbf{greater_tuple_negwdeglex}\((f, g)\)

Return the greater exponent tuple with respect to the negative weighted degree lexicographical term order.

\textbf{INPUT}:

\begin{itemize}
  \item \textit{f} - exponent tuple
  \item \textit{g} - exponent tuple
\end{itemize}

\textbf{EXAMPLES}:

\begin{verbatim}
sage: t = TermOrder('negwdeglex',(1,2,3))
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order=t)
sage: f = x + y; f.lm() # indirect doctest
  x
sage: f = x + x^2; f.lm()  
x
sage: f = x^3 + z; f.lm()
x^3
\end{verbatim}

This method is called by the \texttt{lm/lc/lt} methods of \texttt{MPolynomial_polydict}.  

262 Chapter 3. Multivariate Polynomials
greater_tuple_negwdegrevlex\((f, g)\)
Return the greater exponent tuple with respect to the negative weighted degree reverse lexicographical term order.

INPUT:
• \(f\) - exponent tuple
• \(g\) - exponent tuple

EXAMPLES:
```
sage: t = TermOrder('negwdegrevlex',(1,2,3))
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order=t)
sage: f = x + y; f.lm() \# indirect doctest
x
sage: f = x + x^2; f.lm()
x
sage: f = x^3 + z; f.lm()
x^3
```

This method is called by the \texttt{lm/lc/lt} methods of \texttt{MPolynomial_polydict}.

greater_tuple_wdeglex\((f, g)\)
Return the greater exponent tuple with respect to the weighted degree lexicographical term order.

INPUT:
• \(f\) - exponent tuple
• \(g\) - exponent tuple

EXAMPLES:
```
sage: t = TermOrder('wdeglex',(1,2,3))
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order=t)
sage: f = x + y; f.lm() \# indirect doctest
y
sage: f = x*y + z; f.lm()
x^y
```

This method is called by the \texttt{lm/lc/lt} methods of \texttt{MPolynomial_polydict}.

greater_tuple_wdegrevlex\((f, g)\)
Return the greater exponent tuple with respect to the weighted degree reverse lexicographical term order.

INPUT:
• \(f\) - exponent tuple
• \(g\) - exponent tuple

EXAMPLES:
```
sage: t = TermOrder('wdegrevlex',(1,2,3))
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order=t)
sage: f = x + y; f.lm() \# indirect doctest
y
sage: f = x + y^2*z; f.lm()
y^2*z
```

This method is called by the \texttt{lm/lc/lt} methods of \texttt{MPolynomial_polydict}.
is_block_order()
Return true if self is a block term order.

EXAMPLES:

```
sage: t=TermOrder('deglex',2)+TermOrder('lex',2)
sage: t.is_block_order()
True
```

is_global()
Return true if this term order is definitely global. Return false otherwise, which includes unknown term orders.

EXAMPLES:

```
sage: T = TermOrder('lex')
sage: T.is_global()
True
sage: T = TermOrder('degrevlex', 3) + TermOrder('degrevlex', 3)
sage: T.is_global()
True
sage: T = TermOrder('degrevlex', 3) + TermOrder('negdegrevlex', 3)
sage: T.is_global()
False
sage: T = TermOrder('degneglex', 3)
sage: T.is_global()
True
```

is_local()
Return true if this term order is definitely local. Return false otherwise, which includes unknown term orders.

EXAMPLES:

```
sage: T = TermOrder('lex')
sage: T.is_local()
False
sage: T = TermOrder('negdeglex', 3) + TermOrder('negdegrevlex', 3)
sage: T.is_local()
True
sage: T = TermOrder('degrevlex', 3) + TermOrder('negdegrevlex', 3)
sage: T.is_local()
False
```

is_weighted_degree_order()
Return true if self is a weighted degree term order.

EXAMPLES:

```
sage: t=TermOrder('wdeglex',(2,3))
sage: t.is_weighted_degree_order()
True
```

macaulay2_str()
Return a Macaulay2 representation of self.
Used to convert polynomial rings to their Macaulay2 representation.
EXAMPLES:

```python
sage: P = PolynomialRing(GF(127), 8, names='x', order='degrevlex(3), lex(5)')
sage: T = P.term_order()
sage: T.macaulay2_str()
'{GRevLex => 3, Lex => 5}'
sage: P._macaulay2_().options()['MonomialOrder']  # optional - macaulay2
{MonomialSize => 16}
{GRevLex => {1, 1, 1}}
{Lex => 5}
{Position => Up}
```

`magma_str()`

Return a MAGMA representation of self.

Used to convert polynomial rings to their MAGMA representation.

EXAMPLES:

```python
sage: P = PolynomialRing(GF(127), 10, names='x', order='degrevlex')
sage: magma(P)  # optional - magma
Polynomial ring of rank 10 over GF(127)
Order: Graded Reverse Lexicographical
Variables: x0, x1, x2, x3, x4, x5, x6, x7, x8, x9
```

`matrix()`

Return the matrix defining matrix term order.

EXAMPLES:

```python
sage: t = TermOrder('M(1,2,0,1)')
sage: t.matrix()
[1 2]
[0 1]
```

`name()`

EXAMPLES:

```python
sage: TermOrder('lex').name()
'lex'
```

`singular_moreblocks()`

Return a the number of additional blocks SINGULAR needs to allocate for handling non-native orderings like degneglex.

EXAMPLES:

```python
sage: P = PolynomialRing(GF(127), 10, names='x', order='lex(3), deglex(5), lex(2)')
sage: T = P.term_order()
sage: T.singular_moreblocks()
0
```
sage: P = PolynomialRing(GF(127),10,names='x',order='lex(3),degneglex(5),lex(2)')
sage: T = P.term_order()
sage: T.singular_moreblocks()
1
sage: P = PolynomialRing(GF(127),10,names='x',order='degneglex(5),degneglex(5)')
sage: T = P.term_order()
sage: T.singular_moreblocks()
2

`singular_str()`

Return a SINGULAR representation of self.

Used to convert polynomial rings to their SINGULAR representation.

**EXAMPLES:**

```
sage: P = PolynomialRing(GF(127),10,names='x',order='lex(3),deglex(5),lex(2)')
sage: T = P.term_order()
sage: T.singular_str()
'(lp(3),Dp(5),lp(2))'
sage: P._singular_()
polynomial ring, over a field, global ordering
// coefficients: ZZ/127
// number of vars : 10
// block 1 : ordering lp
// : names x0 x1 x2
// block 2 : ordering Dp
// : names x3 x4 x5 x6 x7
// block 3 : ordering lp
// : names x8 x9
// block 4 : ordering C
```

The `degneglex` ordering is somehow special, it looks like a block ordering in SINGULAR:

```
sage: T = TermOrder("degneglex", 2)
sage: P = PolynomialRing(QQ,2, names='x', order=T)
sage: T = P.term_order()
sage: T.singular_str()
'(a(1:2),ls(2))'
```

```
sage: T = TermOrder("degneglex", 2) + TermOrder("degneglex", 2)
sage: P = PolynomialRing(QQ,4, names='x', order=T)
sage: T = P.term_order()
sage: T.singular_str()
'(a(1:2),ls(2),a(1:2),ls(2))'
sage: P._singular_()
polynomial ring, over a field, global ordering
// coefficients: QQ
// number of vars : 4
// block 1 : ordering a
// : names x0 x1
// : weights 1 1
```
The position of the ordering \( C \) block can be controlled by setting \_singular\_ringorder\_column attribute to an integer:

```python
sage: T = TermOrder("degneglex", 2) + TermOrder("degneglex", 2)
sage: T._singular_ringorder_column = 0
sage: P = PolynomialRing(QQ, 4, names='x', order=T)
sage: P._singular_()
polynomial ring, over a field, global ordering
// coefficients: QQ
// number of vars : 4
// block 1 : ordering C
// block 2 : ordering a
// : names x0 x1
// : weights 1 1
// block 3 : ordering ls
// : names x0 x1
// block 4 : ordering a
// : names x2 x3
// : weights 1 1
// block 5 : ordering ls
// : names x2 x3
```

```python
sage: T._singular_ringorder_column = 1
sage: P = PolynomialRing(QQ, 4, names='y', order=T)
sage: P._singular_()
polynomial ring, over a field, global ordering
// coefficients: QQ
// number of vars : 4
// block 1 : ordering c
// block 2 : ordering a
// : names y0 y1
// : weights 1 1
// block 3 : ordering ls
// : names y0 y1
// block 4 : ordering a
// : names y2 y3
// : weights 1 1
// block 5 : ordering ls
// : names y2 y3
```

```python
sage: T._singular_ringorder_column = 2
sage: P = PolynomialRing(QQ, 4, names='z', order=T)
sage: P._singular_()
```

(continues on next page)
polynomial ring, over a field, global ordering
// coefficients: QQ
// number of vars : 4
// block 1 : ordering a
// : names z0 z1
// : weights 1 1
// block 2 : ordering C
// block 3 : ordering ls
// : names z0 z1
// block 4 : ordering a
// : names z2 z3
// : weights 1 1
// block 5 : ordering ls
// : names z2 z3

sortkey
The default sortkey method for this term order.

EXAMPLES:

```
sage: O = TermOrder()
sage: O.sortkey.__func__ is O.sortkey_lex.__func__
True
sage: O = TermOrder('deglex')
sage: O.sortkey.__func__ is O.sortkey_deglex.__func__
True
```

sortkey_block(f)
Return the sortkey of an exponent tuple with respect to the block order as specified when constructing this element.

INPUT:

• f – exponent tuple

EXAMPLES:

```
sage: P.<a,b,c,d,e,f>=PolynomialRing(QQbar, 6, order='degrevlex(3),degrevlex(3)
˓→')
sage: a > c^4 # indirect doctest
False
sage: a > e^4
True
```

sortkey_deglex(f)
Return the sortkey of an exponent tuple with respect to the degree lexicographical term order.

INPUT:

• f – exponent tuple

EXAMPLES:

```
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='deglex')
sage: x > y^2 # indirect doctest
False
```

(continues on next page)
sage: x > 1
True

sortkey_degneglex(f)
Return the sortkey of an exponent tuple with respect to the degree negative lexicographical term order.

INPUT:
• f – exponent tuple

EXAMPLES:

sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order='degneglex')
sage: x^y > y^z # indirect doctest
False
sage: x^y > x
True

sortkey_degrevlex(f)
Return the sortkey of an exponent tuple with respect to the degree reversed lexicographical term order.

INPUT:
• f – exponent tuple

EXAMPLES:

sage: P.<x,y> = PolynomialRing(QQbar, 2, order='degrevlex')
sage: x > y^2 # indirect doctest
False
sage: x > 1
True

sortkey_invlex(f)
Return the sortkey of an exponent tuple with respect to the inversed lexicographical term order.

INPUT:
• f – exponent tuple

EXAMPLES:

sage: P.<x,y> = PolynomialRing(QQbar, 2, order='invlex')
sage: x > y^2 # indirect doctest
False
sage: x > 1
True

sortkey_lex(f)
Return the sortkey of an exponent tuple with respect to the lexicographical term order.

INPUT:
• f – exponent tuple

EXAMPLES:
sortkey_matrix(f)
Return the sortkey of an exponent tuple with respect to the matrix term order.

INPUT:
• f - exponent tuple

EXAMPLES:

```sage
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='m(1,3,1,0)')
sage: y > x^2 # indirect doctest
True
sage: y > x^3
False
```

sortkey_negdeglex(f)
Return the sortkey of an exponent tuple with respect to the negative degree lexicographical term order.

INPUT:
• f – exponent tuple

EXAMPLES:

```sage
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='negdeglex')
sage: x > y^2 # indirect doctest
True
sage: x > 1
False
```

sortkey_negdegrevlex(f)
Return the sortkey of an exponent tuple with respect to the negative degree reverse lexicographical term order.

INPUT:
• f – exponent tuple

EXAMPLES:

```sage
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='negdegrevlex')
sage: x > y^2 # indirect doctest
True
sage: x > 1
False
```

sortkey_neglex(f)
Return the sortkey of an exponent tuple with respect to the negative lexicographical term order.

INPUT:
• f – exponent tuple

EXAMPLES:
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='neglex')
sage: x > y^2 # indirect doctest
False
sage: x > 1
False

sortkey_negwdeglex(f)
Return the sortkey of an exponent tuple with respect to the negative weighted degree lexicographical term order.

INPUT:
• f – exponent tuple

EXAMPLES:

sage: t = TermOrder('negwdeglex',(3,2))
sage: P.<x,y> = PolynomialRing(QQbar, 2, order=t)
sage: x > y^2 # indirect doctest
True
sage: x^2 > y^3
True

sortkey_negwdegrevlex(f)
Return the sortkey of an exponent tuple with respect to the negative weighted degree reverse lexicographical term order.

INPUT:
• f – exponent tuple

EXAMPLES:

sage: t = TermOrder('negwdegrevlex',(3,2))
sage: P.<x,y> = PolynomialRing(QQbar, 2, order=t)
sage: x > y^2 # indirect doctest
True
sage: x^2 > y^3
True

sortkey_wdeglex(f)
Return the sortkey of an exponent tuple with respect to the weighted degree lexicographical term order.

INPUT:
• f – exponent tuple

EXAMPLES:

sage: t = TermOrder('wdeglex',(3,2))
sage: P.<x,y> = PolynomialRing(QQbar, 2, order=t)
sage: x > y^2 # indirect doctest
False
sage: x > y
True

sortkey_wdegrevlex(f)
Return the sortkey of an exponent tuple with respect to the weighted degree reverse lexicographical term order.
INPUT:

• \( f \) – exponent tuple

EXAMPLES:

```
sage: t = TermOrder('wdegrevlex',(3,2))
sage: P.<x,y> = PolynomialRing(QQbar, 2, order=t)
sage: x > y^2 # indirect doctest
False
sage: x^2 > y^3
True
```

\textit{tuplet\_weight}(f)

Return the weight of tuple \( f \).

INPUT:

• \( f \) - exponent tuple

EXAMPLES:

```
sage: t=TermOrder('wdeglex',(1,2,3))
sage: P.<a,b,c>=PolynomialRing(QQbar, order=t)
sage: P.term_order().tuplet\_weight([3,2,1])
10
```

\textit{weights}()

Return the weights for weighted term orders.

EXAMPLES:

```
sage: t=TermOrder('wdeglex',(2,3))
sage: t.weights()
(2, 3)
```

\texttt{sage.rings.polynomial.term\_order.termoder\_from\_singular}(S)

Return the Sage term order of the basering in the given Singular interface

INPUT:

An instance of the Singular interface.

EXAMPLES:

```
sage: from sage.rings.polynomial.term\_order import termoder\_from\_singular
sage: singular.eval('ring r1 = (9,x),(a,b,c,d,e,f),(M((1,2,3,0)),wp(2,3),lp)')
''
sage: termoder\_from\_singular(singular)
Block term order with blocks:
(Matrix term order with matrix
\[[1 2] [3 0],
Weighted degree reverse lexicographic term order with weights (2, 3),
Lexicographic term order of length 2)
```

A term order in Singular also involves information on orders for modules. This information is reflected in
\texttt{\_singular\_ringorder\_column} attribute of the term order.
sage: singular.ring(0, '(x,y,z,w)', '(C,dp(2),lp(2))')
polynomial ring, over a field, global ordering
// coefficients: QQ
// number of vars : 4
//   block 1 : ordering C
//   block 2 : ordering dp
//     : names x y
//   block 3 : ordering lp
//     : names z w
sage: T = termorder_from_singular(singular)
sage: T
Block term order with blocks:
(Degree reverse lexicographic term order of length 2,
 Lexicographic term order of length 2)
sage: T._singular_ringorder_column
0

sage: singular.ring(0, '(x,y,z,w)', '(c,dp(2),lp(2))')
polynomial ring, over a field, global ordering
// coefficients: QQ
// number of vars : 4
//   block 1 : ordering c
//   block 2 : ordering dp
//     : names x y
//   block 3 : ordering lp
//     : names z w
sage: T = termorder_from_singular(singular)
sage: T
Block term order with blocks:
(Degree reverse lexicographic term order of length 2,
 Lexicographic term order of length 2)
sage: T._singular_ringorder_column
1

3.1.2 Base class for multivariate polynomial rings

class sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base
Bases: sage.rings.ring.CommutativeRing

Create a polynomial ring in several variables over a commutative ring.

EXAMPLES:

sage: R.<x,y> = ZZ['x,y']; R
Multivariate Polynomial Ring in x, y over Integer Ring
sage: class CR(CommutativeRing):
....:     def __init__(self):
....:         CommutativeRing.__init__(self)
....:     def __call__(self,x):
....:         return None
sage: cr = CR()
sage: cr.is_commutative()

(continues on next page)
```
True
sage: cr['x,y']
Multivariate Polynomial Ring in x, y over
<__main__.CR_with_category object at ...>
```

**change_ring**(base_ring=None, names=None, order=None)

Return a new multivariate polynomial ring which isomorphic to self, but has a different ordering given by the parameter ‘order’ or names given by the parameter ‘names’.

**INPUT:**

- base_ring – a base ring
- names – variable names
- order – a term order

**EXAMPLES:**

```
sage: P.<x,y,z> = PolynomialRing(GF(127),3,order='lex')
sage: x > y^2
True
sage: Q.<x,y,z> = P.change_ring(order='degrevlex')
sage: x > y^2
False
```

```
characteristic()

Return the characteristic of this polynomial ring.

**EXAMPLES:**

```
sage: R = PolynomialRing(QQ, 'x', 3)
sage: R.characteristic()
0
sage: R = PolynomialRing(GF(7), 'x', 20)
sage: R.characteristic()
7
```

**completion**(names, prec=20, extras={})

Return the completion of self with respect to the ideal generated by the variable(s) names.

**INPUT:**

- names – variable or list/tuple of variables (given either as elements of the polynomial ring or as strings)
- prec – default precision of resulting power series ring
- extras – passed as keywords to PowerSeriesRing

**EXAMPLES:**

```
sage: P.<x,y,z,w> = PolynomialRing(ZZ)
sage: P.completion('w')
Power Series Ring in w over Multivariate Polynomial Ring in x, y, z over Integer Ring
sage: P.completion((w,x,y))
Multivariate Power Series Ring in w, x, y over Univariate Polynomial Ring in z over Integer Ring
```

(continues on next page)
construction()
Returns a functor F and base ring R such that F(R) == self.

EXAMPLES:

```python
sage: S = ZZ['x,y']
sage: F, R = S.construction(); R
Integer Ring
sage: F
MPoly['x,y']
sage: F(R) == S
True
sage: F(R) == ZZ['x']['y']
False
```

flattening_morphism()
Return the flattening morphism of this polynomial ring.

EXAMPLES:

```python
sage: QQ['a','b']['x','y'].flattening_morphism()
Flattening morphism:
  From: Multivariate Polynomial Ring in x, y over Multivariate Polynomial Ring in a, b over Rational Field
  To:   Multivariate Polynomial Ring in a, b, x, y over Rational Field
sage: QQ['x,y'].flattening_morphism()
Identity endomorphism of Multivariate Polynomial Ring in x, y over Rational Field
```

gen(n=0)

irrelevant_ideal()
Return the irrelevant ideal of this multivariate polynomial ring.

This is the ideal generated by all of the indeterminate generators of this ring.

EXAMPLES:
sage: R.<x,y,z> = QQ[]
sage: R.irrelevant_ideal()
Ideal (x, y, z) of Multivariate Polynomial Ring in x, y, z over Rational Field

**is_exact()**
Test whether this multivariate polynomial ring is defined over an exact base ring.

**EXAMPLES:**

```python
sage: PolynomialRing(QQ, 2, 'x').is_exact()
True
sage: PolynomialRing(RDF, 2, 'x').is_exact()
False
```

**is_field**(proof=True)
Test whether this multivariate polynomial ring is a field.

A polynomial ring is a field when there are no variable and the base ring is a field.

**EXAMPLES:**

```python
sage: PolynomialRing(QQ, 'x', 2).is_field()
False
sage: PolynomialRing(QQ, 'x', 0).is_field()
True
sage: PolynomialRing(ZZ, 'x', 0).is_field()
False
sage: PolynomialRing(Zmod(1), names=['x','y']).is_finite()
True
```

**is_integral_domain**(proof=True)

**EXAMPLES:**

```python
sage: ZZ['x,y'].is_integral_domain()
True
sage: Integers(8)['x,y'].is_integral_domain()
False
```

**is_noetherian()**

**EXAMPLES:**

```python
sage: ZZ['x,y'].is_noetherian()
True
sage: Integers(8)['x,y'].is_noetherian()
True
```

**krull_dimension**

**macaulay_resultant**(*args, **kwds)
This is an implementation of the Macaulay Resultant. It computes the resultant of universal polynomials as well as polynomials with constant coefficients. This is a project done in sage days 55. It’s based on the implementation in Maple by Manfred Minimair, which in turn is based on the references listed below: It calculates the Macaulay resultant for a list of polynomials, up to sign!

**REFERENCES:**
• [CLO2005]
• [Can1990]
• [Mac1916]

AUTHORS:
• Hao Chen, Solomon Vishkautsan (7-2014)

INPUT:
• **args** – a list of \( n \) homogeneous polynomials in \( n \) variables. works when \( \text{args}[0] \) is the list of polynomials, or \( \text{args} \) is itself the list of polynomials

kwds:
• **sparse** – boolean (optional - default: False) if True function creates sparse matrices.

OUTPUT:
• the macaulay resultant, an element of the base ring of self

**Todo:** Working with sparse matrices should usually give faster results, but with the current implementation it actually works slower. There should be a way to improve performance with regards to this.

EXAMPLES:
The number of polynomials has to match the number of variables:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: R.macaulay_resultant([y,x+z])
Traceback (most recent call last):
...TypeError: number of polynomials(= 2) must equal number of variables (= 3)
```

The polynomials need to be all homogeneous:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: R.macaulay_resultant([y, x+z, z+x^3])
Traceback (most recent call last):
...TypeError: resultant for non-homogeneous polynomials is not supported
```

All polynomials must be in the same ring:

```
sage: S.<x,y> = PolynomialRing(QQ, 2)
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: S.macaulay_resultant([y, z+x])
Traceback (most recent call last):
...TypeError: not all inputs are polynomials in the calling ring
```

The following example recreates Proposition 2.10 in Ch.3 in [CLO2005]:

3.1. Multivariate Polynomials and Polynomial Rings 277
```python
 sage: K.<x,y> = PolynomialRing(ZZ, 2)
sage: flist,R = K._macaulay_resultant_universal_polynomials([1,1,2])
sage: R.macaulay_resultant(flist)
u2^2*u4^2*u6 - 2*u1*u2*u4*u5*u6 + u1^2*u5^2*u6 - u2^2*u3*u4*u7 + u1*u2*u3*u5*u7 + u0*u2*u4*u5*u7 - u0*u1*u5^2*u7 + u1*u2*u3*u4*u8 - u0*u2*u4^2*u8 - u1^2*u3*u5*u8 + u0*u1*u4*u5*u8 + u2^2*u3^2*u9 - 2*u0^2*u2*u3^2*u10 + u0^2*u3*u4*u10 + u0*u1*u3*u5*u10 - u0*u2*u4^2*u10 + u1^2*u3^2*u11 - 2*u0*u1*u3*u4*u11 + u0^2*u4^2*u11
```

The following example degenerates into the determinant of a 3 * 3 matrix:

```python
 sage: K.<x,y> = PolynomialRing(ZZ, 2)
sage: flist,R = K._macaulay_resultant_universal_polynomials([1,1,1])
sage: R.macaulay_resultant(flist)
-u2*u4*u6 + u1*u5*u6 + u2*u3*u7 - u0*u5*u7 - u1*u3*u8 + u0*u4*u8
```

The following example is by Patrick Ingram (arXiv 1310.4114):

```python
 sage: U = PolynomialRing(ZZ, 'y',2); y0,y1 = U.gens()
sage: R = PolynomialRing(U,'x',3); x0,x1,x2 = R.gens()
sage: f0 = y0*x2^2 - x0^2 + 2*x1*x2
 sage: f1 = y1*x2^2 - x1^2 + 2*x0*x2
 sage: f2 = x0*x1 - x2^2
 sage: flist = [f0,f1,f2]
sage: R.macaulay_resultant([f0,f1,f2])
y0^2*y1^2 - 4*y0^3 - 4*y1^3 + 18*y0*y1 - 27
```

A simple example with constant rational coefficients:

```python
 sage: R.<x,y,z,w> = PolynomialRing(QQ,4)
sage: R.macaulay_resultant([w,z,y,x])
1
```

An example where the resultant vanishes:

```python
 sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: R.macaulay_resultant([x+y,y^2,x])
0
```

An example of bad reduction at a prime $p = 5$:

```python
 sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: R.macaulay_resultant([y,x^3+25*y^2*x,5*z])
125
```

The input can be given as an unpacked list of polynomials:

```python
 sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: R.macaulay_resultant(y,x^3+25*y^2*x,5*z)
125
```

An example when the coefficients live in a finite field:

```python
```
sage: F = FiniteField(11)
sage: R.<x,y,z,w> = PolynomialRing(F,4)
sage: R.macaulay_resultant([z,x^3,5*y,w])
4

Example when the denominator in the algorithm vanishes (in this case the resultant is the constant term of
the quotient of char polynomials of numerator/denominator):

sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: R.macaulay_resultant([y, x+z, z^2])
-1

When there are only 2 polynomials, macaulay resultant degenerates to the traditional resultant:

sage: R.<x> = PolynomialRing(QQ,1)
sage: f = x^2+1; g = x^5+1
sage: fh = f.homogenize()
sage: gh = g.homogenize()
sage: RH = fh.parent()
sage: f.resultant(g) == RH.macaulay_resultant([fh,gh])
True

\textbf{monomial(*exponents)}

Return the monomial with given exponents.

\textbf{EXAMPLES:}

sage: R.<x,y,z> = PolynomialRing(ZZ, 3)
sage: R.monomial(1,1,1)
x*y*z
sage: e=(1,2,3)
sage: R.monomial(*e)
x*y^2*z^3
sage: m = R.monomial(1,2,3)
sage: R.monomial(*m.degrees()) == m
True

\textbf{ngens()}

\textbf{random_element}(degree=2, terms=None, choose_degree=False, *args, **kwargs)

Return a random polynomial of at most degree $d$ and at most $t$ terms.

First monomials are chosen uniformly random from the set of all possible monomials of degree up to $d$
(inclusive). This means that it is more likely that a monomial of degree $d$ appears than a monomial of
degree $d - 1$ because the former class is bigger.

Exactly $t$ distinct monomials are chosen this way and each one gets a random coefficient (possibly zero)
from the base ring assigned.

The returned polynomial is the sum of this list of terms.

\textbf{INPUT:}

\begin{itemize}
\item \textbf{degree} – maximal degree (likely to be reached) (default: 2)
\item \textbf{terms} – number of terms requested (default: 5). If more terms are requested than exist, then this
parameter is silently reduced to the maximum number of available terms.
\end{itemize}
• `choose_degree` – choose degrees of monomials randomly first rather than monomials uniformly random.

• `**kwargs` – passed to the random element generator of the base ring

EXAMPLES:

```python
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: P.random_element(2, 5)
-6/5*x^2 + 2/3*z^2 - 1
sage: P.random_element(2, 5, choose_degree=True)
-1/4*x*y - x - 1/14*z - 1
```

Stacked rings:

```python
sage: R = QQ['x,y']
sage: S = R['t,u']
sage: S.random_element(degree=2, terms=1)
-1/2*x^2 - 1/4*x*y - 3*y^2 + 4*y
sage: S.random_element(degree=2, terms=1)
(-x^2 - 2*y^2 - 1/3*x + 2*y + 9)*u^2
```

Default values apply if no degree and/or number of terms is provided:

```python
sage: random_matrix(QQ['x,y,z'], 2, 2)
[357*x^2 + 1/4*y^2 + 2*y*z + 2*z^2 + 28*x 2*x^2 + 3/2*y^2 + 2*y*z - 2*z^2 - z]
[ x^2 - y^2 + z + 2*z^2 - 3/4*y]
```

```python
sage: random_matrix(QQ['x,y,z'], 2, 2, terms=1, degree=2)
[1/2*x^2 - 1/4*x]
[1/2 1/3*x]
```

```python
sage: P.random_element(0, 1)
1
sage: P.random_element(2, 0)
0
```

```python
sage: R.<x> = PolynomialRing(Integers(3), 1)
sage: R.random_element()
2*x^2 + x
```

To produce a dense polynomial, pick `terms=Infinity`:

```python
sage: P.<x,y,z> = GF(127)[]
sage: P.random_element(degree=2, terms=Infinity)
-55*x^2 - 51*x*y + 5*y^2 + 55*x^2*z - 59*y*z + 20*z^2 + 19*x - 55*y - 28*z + 17
sage: P.random_element(degree=3, terms=Infinity)
-54*x^3 + 15*x^2*y - x^2*y^2 - 39*x^2*z + 61*x^2 + 12*x*y*z + 20*y^2*z - 61*x*z^2 + 5*y^2*z + 62*z^3 + 15*x^2 - 47*x*y + 31*y^2 - 14*x*z + 29*y*z + 13*z^2 + 61*x - 40*y - 49*z + 30
```

(continues on next page)
 sage: P.random_element(degree=3, terms=Infinity, choose_degree=True)
57*x^3 - 58*x^2*y + 21*x*y^2 + 36*y^3 + 7*x^2*z - 57*x*y*z +
8*y^2*z - 11*x*z^2 + 7*y^2*z + 6*z^3 - 38*x^2 - 18*x*y -
52*y^2 + 27*x*z + 4*y*z^2 - 51*z^2 - 63*x + 7*y + 48*z + 14

The number of terms is silently reduced to the maximum available if more terms are requested:

 sage: P.<x,y,z> = GF(127)[]
 sage: P.random_element(degree=2, terms=1000)
5*x^2 - 10*x*y + 10*y^2 - 44*x*z + 31*y*z + 19*z^2 - 42*x
- 50*y - 49*z - 60

 remove_var(order=None, *var)
Remove a variable or sequence of variables from self.

If order is not specified, then the subring inherits the term order of the original ring, if possible.

EXAMPLES:

 sage: P.<x,y,z,w> = PolynomialRing(ZZ)
 sage: P.remove_var(z)
Multivariate Polynomial Ring in x, y, w over Integer Ring
 sage: P.remove_var(z,x)
Multivariate Polynomial Ring in y, w over Integer Ring
 sage: P.remove_var(y,z,x)
Univariate Polynomial Ring in w over Integer Ring

Removing all variables results in the base ring:

 sage: P.remove_var(y,z,x,w)
Integer Ring

If possible, the term order is kept:

 sage: R.<x,y,z,w> = PolynomialRing(ZZ, order='deglex')
 sage: R.remove_var(y).term_order()
Degree lexicographic term order

 sage: R.<x,y,z,w> = PolynomialRing(ZZ, order='lex')
 sage: R.remove_var(y).term_order()
Lexicographic term order

Be careful with block orders when removing variables:

 sage: R.<x,y,z,u,v> = PolynomialRing(ZZ, order='deglex(2),lex(3)')
 sage: R.remove_var(x,y,z)
Traceback (most recent call last):
...
ValueError: impossible to use the original term order (most
likely because it was a block order). Please specify the term
order for the subring
 sage: R.remove_var(x,y,z, order='degrevlex')
Multivariate Polynomial Ring in u, v over Integer Ring
**repr_long()**

Return structured string representation of self.

**EXAMPLES:**

```
sage: P.<x,y,z> = PolynomialRing(QQ,order=TermOrder('degrevlex',1)+TermOrder('lex',2))
sage: print(P.repr_long())
Polynomial Ring
  Base Ring : Rational Field
  Size : 3 Variables
    Block 0 : Ordering : degrevlex
      Names : x
    Block 1 : Ordering : lex
      Names : y, z
```

**term_order()**

**univariate_ring(x)**

Return a univariate polynomial ring whose base ring comprises all but one variables of self.

**INPUT:**

- **x** – a variable of self.

**EXAMPLES:**

```
sage: P.<x,y,z> = QQ[]
sage: P.univariate_ring(y)
Univariate Polynomial Ring in y over Multivariate Polynomial Ring in x, z over Rational Field
```

**variable_names_recursive(depth=None)**

Returns the list of variable names of this and its base rings, as if it were a single multi-variate polynomial.

**EXAMPLES:**

```
sage: R = QQ['x,y,z']['z,w']
sage: R.variable_names_recursive()
('x', 'y', 'z', 'w')
sage: R.variable_names_recursive(3)
('y', 'z', 'w')
```

**weyl_algebra()**

Return the Weyl algebra generated from self.

**EXAMPLES:**

```
sage: R = QQ['x,y,z']
sage: W = R.weyl_algebra(); W
Differential Weyl algebra of polynomials in x, y, z over Rational Field
sage: W.polynomial_ring() == R
True
```

`sage.rings.polynomial.multi_polynomial_ring_base.is_MPolynomialRing(x)`

`sage.rings.polynomial.multi_polynomial_ring_base.unpickle_MPolynomialRing_generic(base_ring, n, names, order)`
3.1.3 Base class for elements of multivariate polynomial rings

```python
class sage.rings.polynomial.multi_polynomial.MPolynomial
    Bases: sage.structure.element.CommutativeRingElement

    args()
    Returns the named of the arguments of self, in the order they are accepted from call.

    EXAMPLES:
    sage: R.<x,y> = ZZ[]
    sage: x.args()
    (x, y)

    change_ring(R)
    Return a copy of this polynomial but with coefficients in R, if at all possible.

    INPUT:
    * R -- a ring or morphism.

    EXAMPLES:
    sage: R.<x,y> = QQ[]
    sage: f = x^3 + 3/5*y + 1
    sage: f.change_ring(GF(7))
    x^3 + 2*y + 1
    sage: R.<x,y> = GF(9,'a')[]
    sage: (x+2*y).change_ring(GF(3))
    x - y
    sage: K.<z> = CyclotomicField(3)
    sage: R.<x,y> = K[]
    sage: f = x^2 + z*y
    sage: f.change_ring(K.embeddings(CC)[1])
    x^2 + (-0.500000000000000 - 0.866025403784439*I)*y
```

coefficients()
Return the nonzero coefficients of this polynomial in a list. The returned list is decreasingly ordered by the term ordering of self.parent(), i.e. the list of coefficients matches the list of monomials returned by sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular.monomials().

EXAMPLES:
```
sage: R.<x,y,z> = PolynomialRing(QQ,3,order='degrevlex')
sage: f=23*x^6*y^7 + x^3*y+6*x^7*z
sage: f.coefficients()
(continues on next page)
```
sage: R.<x,y,z> = PolynomialRing(QQ,3,order='lex')
sage: f=23*x^6*y^7 + x^3*y+6*x^7*z
sage: f.coefficients()
[6, 23, 1]

Test the same stuff with base ring \( \mathbb{Z} \) – different implementation:

sage: R.<x,y,z> = PolynomialRing(ZZ,3,order='degrevlex')
sage: f=23*x^6*y^7 + x^3*y+6*x^7*z
sage: f.coefficients()
[23, 6, 1]
sage: R.<x,y,z> = PolynomialRing(ZZ,3,order='lex')
sage: f=23*x^6*y^7 + x^3*y+6*x^7*z
sage: f.coefficients()
[6, 23, 1]

AUTHOR:

• Didier Deshommes

content()  
Returns the content of this polynomial. Here, we define content as the gcd of the coefficients in the base ring.

See also:

content_ideal()

EXAMPLES:

sage: R.<x,y> = ZZ[]
sage: f = 4*x+6*y
sage: f.content()
2
sage: f.content().parent()
Integer Ring

content_ideal()  
Return the content ideal of this polynomial, defined as the ideal generated by its coefficients.

See also:

content()

EXAMPLES:

sage: R.<x,y> = ZZ[]
sage: f = 2*x*y + 6*x - 4*y + 2
sage: f.content_ideal()
Principal ideal (2) of Integer Ring
sage: S.<z,t> = R[]
sage: g = x*z + y*t
sage: g.content_ideal()
Ideal (x, y) of Multivariate Polynomial Ring in x, y over Integer Ring
denominator()

Return a denominator of self.

First, the lcm of the denominators of the entries of self is computed and returned. If this computation fails, the unit of the parent of self is returned.

Note that some subclasses may implement its own denominator function.

**Warning:** This is not the denominator of the rational function defined by self, which would always be 1 since self is a polynomial.

**EXAMPLES:**

First we compute the denominator of a polynomial with integer coefficients, which is of course 1.

```
sage: R.<x,y> = ZZ[

sage: f = x^3 + 17*y + x + y

sage: f.denominator()

1
```

Next we compute the denominator of a polynomial over a number field.

```
sage: R.<x,y> = NumberField(symbolic_expression(x^2+3) , 'a')['x,y']

sage: f = (1/17)*x^19 + (1/6)*y - (2/3)*x + 1/3; f
1/17*x^19 - 2/3*x + 1/6*y + 1/3

sage: f.denominator()
102
```

Finally, we try to compute the denominator of a polynomial with coefficients in the real numbers, which is a ring whose elements do not have a denominator method.

```
sage: R.<a,b,c> = RR[

sage: f = a + b + RR('0.3'); f
a + b + 0.300000000000000

sage: f.denominator()
1.00000000000000
```

Check that the denominator is an element over the base whenever the base has no denominator function. This closes trac ticket #9063:

```
sage: from sage.rings.polynomial.multi_polynomial_element import MPolynomial

sage: isinstance(a / b, MPolynomial)
False

sage: isinstance(a.numerator() / a.denominator(), MPolynomial)
True
```
derivative(*args)
The formal derivative of this polynomial, with respect to variables supplied in args.

Multiple variables and iteration counts may be supplied; see documentation for the global derivative() function for more details.

See also:
_.derivative()

EXAMPLES:

Polynomials implemented via Singular:

```
sage: R.<x, y> = PolynomialRing(FiniteField(5))
sage: f = x^3*y^5 + x^7*y
sage: type(f)
<type 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular'>
sage: f.derivative(x)
2*x^6*y - 2*x^2*y^5
sage: f.derivative(y)
x^7
```

Generic multivariate polynomials:

```
sage: R.<t> = PowerSeriesRing(QQ)
sage: S.<x, y> = PolynomialRing(R)
sage: f = (t^2 + O(t^3))*x^2*y^3 + (37*t^4 + O(t^5))*x^3
sage: type(f)
<class 'sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict'>
sage: f.derivative(x)  # with respect to x
(2*t^2 + O(t^3))*x*y^3 + (111*t^4 + O(t^5))*x^2
sage: f.derivative(y)  # with respect to y
(3*t^2 + O(t^3))*x^2*y^2
sage: f.derivative(t)  # with respect to t (recurses into base ring)
(2*t + O(t^2))*x^2*y^3 + (148*t^3 + O(t^4))*x^3
sage: f.derivative(x, y)  # with respect to x and then y
(6*t^2 + O(t^3))*x*y^2
sage: f.derivative(y, 3)  # with respect to y three times
(6*t^2 + O(t^3))*x^2
sage: f.derivative()  # can't figure out the variable
Traceback (most recent call last):
...    ValueError: must specify which variable to differentiate with respect to
```

Polynomials over the symbolic ring (just for fun...):

```
sage: x = var("x")
sage: S.<u, v> = PolynomialRing(SR)
sage: f = u*v*x
sage: f.derivative(x) == u*v
True
sage: f.derivative(u) == v*x
True
```
**discriminant** *(variable)*

Returns the discriminant of self with respect to the given variable.

**INPUT:**

- **variable** - The variable with respect to which we compute the discriminant

**OUTPUT:**

- An element of the base ring of the polynomial ring.

**EXAMPLES:**

```sage
discriminant(eq, variable)
```

```sage
R.<x,y,z>=QQ[]  
sage: f=4*x*y^2 + 1/4*x*y*z + 3/2*x*z^2 - 1/2*z^2  
sage: f.discriminant(x)  
1  
sage: f.discriminant(y)  
-383/16*x^2*z^2 + 8*x*z^2  
sage: f.discriminant(z)  
-383/16*x^2*y^2 + 8*x*y^2
```

Note that, unlike the univariate case, the result lives in the same ring as the polynomial:

```sage
R.<x,y>=QQ[]  
sage: f=x^5*y+3*x^2*y^2-2*x+y-1  
sage: f.discriminant(y)  
x^10 + 2*x^5 + 24*x^3 + 12*x^2 + 1  
sage: f.polynomial(y).discriminant()  
x^10 + 2*x^5 + 24*x^3 + 12*x^2 + 1  
sage: f.discriminant(y).parent()==f.polynomial(y).discriminant().parent()  
False
```

**AUTHOR:** Miguel Marco

**gcd**(other)

Return a greatest common divisor of this polynomial and other.

**INPUT:**

- **other** – a polynomial with the same parent as this polynomial

**EXAMPLES:**

```sage
gcd(eq1, eq2)
```

```sage
Q.<z> = Frac(QQ['z'])  
R.<x,y> = Q[]  
r = x*y - (2*z-1)/(z^2+z+1) * x + y/z  
p = r^* (x + z^*y - 1/z^2)  
q = r^* (x*y^*z + 1)  
gcd(p,q)  
(z^3 + z^2 + z)*x*y + (-2*z^2 + z)*x + (z^2 + z + 1)*y
```

Polynomials over polynomial rings are converted to a simpler polynomial ring with all variables to compute the gcd:

```sage
A.<z,t> = ZZ[]  
B.<x,y> = A[]
```

(continues on next page)
sage: r = x*y*z*t+1
sage: p = r * (x - y + z - t + 1)
sage: q = r * (x*z - y*t)
sage: gcd(p,q)
z*t*x*y + 1
sage: _.parent()
Multivariate Polynomial Ring in x, y over Multivariate Polynomial Ring in z, t over Integer Ring

Some multivariate polynomial rings have no gcd implementation:

sage: R.<x,y> = GaussianIntegers()[]
sage: x.gcd(x)
Traceback (most recent call last):
... NotImplementedError: GCD is not implemented for multivariate polynomials over
Gaussian Integers in Number Field in I with defining polynomial x^2 + 1 with
I = 1*I

\textbf{gradient()}

Return a list of partial derivatives of this polynomial, ordered by the variables of \texttt{self.parent()}.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: P.<x,y,z> = PolynomialRing(ZZ,3)
sage: f = x*y + 1
sage: f.gradient()
[y, x, 0]
\end{verbatim}

\textbf{homogeneous_components()}

Return the homogeneous components of this polynomial.

\textbf{OUTPUT:}

A dictionary mapping degrees to homogeneous polynomials.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: R.<x,y> = QQ[]
sage: (x^3 + 2*x*y^3 + 4*y^3 + y).homogeneous_components()
{1: y, 3: x^3 + 4*y^3, 4: 2*x*y^3}
sage: R.zero().homogeneous_components()
{}
\end{verbatim}

In case of weighted term orders, the polynomials are homogeneous with respect to the weights:

\begin{verbatim}
sage: S.<a,b,c> = PolynomialRing(ZZ, order=TermOrder('wdegrevlex', (1,2,3)))
sage: (a^6 + b^3 + b*c + a^2*c + c + a + 1).homogeneous_components()
{0: 1, 1: a, 3: c, 5: a^2*c + b*c, 6: a^6 + b^3}
\end{verbatim}

\textbf{homogenize}(\texttt{var='h'})

Return the homogenization of this polynomial.

The polynomial itself is returned if it is homogeneous already. Otherwise, the monomials are multiplied with the smallest powers of \texttt{var} such that they all have the same total degree.

\textbf{INPUT:}
• **var** – a variable in the polynomial ring (as a string, an element of the ring, or a zero-based index in the list of variables) or a name for a new variable (default: ‘h’)

**OUTPUT:**

If **var** specifies a variable in the polynomial ring, then a homogeneous element in that ring is returned. Otherwise, a homogeneous element is returned in a polynomial ring with an extra last variable **var**.

**EXAMPLES:**

```python
sage: R.<x,y> = QQ[]
sage: f = x^2 + y + 1 + 5*x*y^10
sage: f.homogenize()
5*x*y^10 + x^2*h^9 + y*h^10 + h^11
```

The parameter **var** can be used to specify the name of the variable:

```python
sage: g = f.homogenize('z'); g
5*x*y^10 + x^2*z^9 + y*z^10 + z^11
sage: g.parent()
Multivariate Polynomial Ring in x, y, z over Rational Field
```

However, if the polynomial is homogeneous already, then that parameter is ignored and no extra variable is added to the polynomial ring:

```python
sage: f = x^2 + y^2
sage: g = f.homogenize('z'); g
x^2 + y^2
sage: g.parent()
Multivariate Polynomial Ring in x, y over Rational Field
```

If you want the ring of the result to be independent of whether the polynomial is homogenized, you can use **var** to use an existing variable to homogenize:

```python
sage: R.<x,y,z> = QQ[]
sage: f = x^2 - y
sage: g = f.homogenize(z); g
x^2 - y*z
sage: g.parent()
Multivariate Polynomial Ring in x, y, z over Rational Field
```

The parameter **var** can also be given as a zero-based index in the list of variables:

```python
sage: g = f.homogenize(2); g
x^2 - y*z
```

If the variable specified by **var** is not present in the polynomial, then setting it to 1 yields the original polynomial:

```python
sage: g(x,y,1)
x^2 - y
```
If it is present already, this might not be the case:

```python
sage: g = f.homogenize(x); g
x^2 - x*y
sage: g(1,y,z)
-y + 1
```

In particular, this can be surprising in positive characteristic:

```python
sage: R.<x,y> = GF(2)[]
sage: f = x + 1
sage: f.homogenize(x)
0
```

**inverse_mod(I)**

Returns an inverse of self modulo the polynomial ideal $I$, namely a multivariate polynomial $f$ such that

$$self \times f - 1 \in I.$$

**INPUT:**

- $I$ – an ideal of the polynomial ring in which self lives

**OUTPUT:**

- a multivariate polynomial representing the inverse of $f$ modulo $I$

**EXAMPLES:**

```python
sage: R.<x1,x2> = QQ[]
sage: I = R.ideal(x2**2 + x1 - 2, x1**2 - 1)
sage: f = x1 + 3*x2^2; g = f.inverse_mod(I); g
1/16*x1 + 3/16
sage: (f*g).reduce(I)
1
```

Test a non-invertible element:

```python
sage: R.<x1,x2> = QQ[]
sage: I = R.ideal(x2**2 + x1 - 2, x1**2 - 1)
sage: f = x1 + x2
sage: f.inverse_mod(I)
Traceback (most recent call last):
  ...
ArithmeticError: element is non-invertible
```

**is_generator()**

Returns True if this polynomial is a generator of its parent.

**EXAMPLES:**

```python
sage: R.<x,y>=ZZ[]
sage: x.is_generator()
True
sage: (x+y-y).is_generator()
True
sage: (x*y).is_generator()
False
```

(continues on next page)
is_homogeneous()
Return True if self is a homogeneous polynomial.

Note: This is a generic implementation which is likely overridden by subclasses.

is_nilpotent()
Return True if self is nilpotent, i.e., some power of self is 0.

EXAMPLES:

```
sage: R.<x,y> = QQ[]  
sage: x.is_generator()  
True  
sage: (x+y-y).is_generator()  
True  
sage: (x*y).is_generator()  
False
```

is_square(root=False)
Test whether this polynomial is a square root.

INPUT:

* root - if set to True return a pair (True, root) where root is a square root or (False, None) if it is not a square.

EXAMPLES:

```
sage: R.<a,b> = QQ[]  
sage: a.is_square()  
False  
sage: ((1+a*b^2)^2).is_square()  
True  
sage: ((1+a*b^2)^2).is_square(root=True)  
(True, a*b^2 + 1)
```

is_symmetric(group=None)
Return whether this polynomial is symmetric.

3.1. Multivariate Polynomials and Polynomial Rings 291
INPUT:

• group (default: symmetric group) – if set, test whether the polynomial is invariant with respect to the
given permutation group

EXAMPLES:

```python
sage: R.<x,y,z> = QQ[]
sage: p = (x+y+z)**2 - 3 * (x+y)*(x+z)*(y+z)
sage: p.is_symmetric()
True
sage: (x + y - z).is_symmetric()
False
sage: R.one().is_symmetric()
True
sage: p = (x-y)*(y-z)*(z-x)
sage: p.is_symmetric()
False
sage: p.is_symmetric(AlternatingGroup(3))
True
sage: R.<x,y> = QQ[]
sage: ((x + y)**2).is_symmetric()
True
sage: R.one().is_symmetric()
True
sage: (x + 2*y).is_symmetric()
False
```

An example with a GAP permutation group (here the quaternions):

```python
sage: R = PolynomialRing(QQ, 'x', 8)
sage: x = R.gens()
sage: p = sum(prod(x[i] for i in e) for e in [(0,1,2), (0,1,7), (0,2,7), (1,2,7), (3,4,5), (3,4,6), (3,5,6), (4,5,6)])
sage: p.is_symmetric(libgap.TransitiveGroup(8, 5))
True
sage: p = sum(prod(x[i] for i in e) for e in [(0,1,2), (0,1,7), (0,2,7), (1,2,7), (3,4,5), (3,4,6), (3,5,6)])
sage: p.is_symmetric(libgap.TransitiveGroup(8, 5))
False
```

`is_unit()`

Return True if self is a unit, that is, has a multiplicative inverse.

EXAMPLES:

```python
sage: R.<x,y> = QQbar[]
sage: (x+y).is_unit()
False
sage: R(0).is_unit()
False
sage: R(-1).is_unit()
True
```

(continues on next page)
Check that trac ticket #22454 is fixed:

```
sage: _.<x,y> = Zmod(4)[]
sage: (1 + 2*x).is_unit()  
True
sage: (x*y).is_unit()      
False
sage: _.<x,y> = Zmod(36)[]
sage: (7+ 6*x + 12*y - 18*x*y).is_unit() 
True
```

**iterator_exp_coeff** *(as_ETuples=True)*

Iterate over self as pairs of ((E)Tuple, coefficient).

**INPUT:**

- as_ETuples – (default: True) if True iterate over pairs whose first element is an ETuple, otherwise as a tuples

**EXAMPLES:**

```
sage: R.<a,b,c> = QQ[]
sage: f = a*c^3 + a^2*b + 2*b^4
sage: list(f.iterator_exp_coeff())
[((0, 0, 4, 0), 2), ((1, 0, 0, 3), 1), ((2, 1, 0, 0), 1)]
sage: list(f.iterator_exp_coeff(as_ETuples=False))
[((0, 0, 4, 0), 2), ((1, 0, 0, 3), 1), ((2, 1, 0, 0), 1)]
sage: R.<a,b,c> = PolynomialRing(QQ, 3, order='lex')
sage: f = a*c^3 + a^2*b + 2*b^4
sage: list(f.iterator_exp_coeff())
[((2, 1, 0, 0), 1), ((1, 0, 0, 3), 1), ((0, 4, 0, 0), 2)]
```

**jacobian_ideal()**

Return the Jacobian ideal of the polynomial self.

**EXAMPLES:**

```
sage: R.<x,y,z> = QQ[]
sage: f = x^3 + y^3 + z^3
sage: f.jacobian_ideal()  
Ideal (3*x^2, 3*y^2, 3*z^2) of Multivariate Polynomial Ring in x, y, z over Rational Field
```

**lift(I)**

given an ideal $I = (f_1, \ldots , f_r)$ and some $g$ (= self) in $I$, find $s_1, \ldots , s_r$ such that $g = s_1 f_1 + \ldots + s_r f_r$.

**EXAMPLES:**
macaulay_resultant(*args)

This is an implementation of the Macaulay Resultant. It computes the resultant of universal polynomials as well as polynomials with constant coefficients. This is a project done in sage days 55. It’s based on the implementation in Maple by Manfred Minimair, which in turn is based on the references [CLO], [Can], [Mac]. It calculates the Macaulay resultant for a list of Polynomials, up to sign!

AUTHORS:

• Hao Chen, Solomon Vishkautsan (7-2014)

INPUT:

• args – a list of $n - 1$ homogeneous polynomials in $n$ variables. works when args[0] is the list of polynomials, or args is itself the list of polynomials

OUTPUT:

• the macaulay resultant

EXAMPLES:

The number of polynomials has to match the number of variables:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: y.macaulay_resultant(x+z)
Traceback (most recent call last):
  ...  TypeError: number of polynomials(= 2) must equal number of variables (= 3)
```

The polynomials need to be all homogeneous:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: y.macaulay_resultant([x+z, z+x^3])
Traceback (most recent call last):
  ...  TypeError: resultant for non-homogeneous polynomials is not supported
```

All polynomials must be in the same ring:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: S.<x,y> = PolynomialRing(QQ, 2)
sage: y.macaulay_resultant(z-x,z)
Traceback (most recent call last):
  ...  TypeError: not all inputs are polynomials in the calling ring
```

The following example recreates Proposition 2.10 in Ch.3 of Using Algebraic Geometry:
The following example degenerates into the determinant of a $3 \times 3$ matrix:

```python
sage: K.<x,y> = PolynomialRing(ZZ, 2)
sage: flist,R = K._macaulay_resultant_universal_polynomials([1,1,1])
sage: flist[0].macaulay_resultant(flist[1:])
-u2*u4*u6 + u1*u5*u6 + u2*u3*u7 - u0*u5*u7 - u1*u3*u8 + u0*u4*u8
```

The following example is by Patrick Ingram (arXiv 1310.4114):

```python
sage: U = PolynomialRing(ZZ, 'y', 2); y0,y1 = U.gens()
sage: R = PolynomialRing(U, 'x', 3); x0,x1,x2 = R.gens()
sage: f0 = y0*x2^2 - x0^2 + 2*x1*x2
sage: f1 = y1*x2^2 - x1^2 + 2*x0*x2
sage: f2 = x0*x1 - x2^2
sage: f0.macaulay_resultant(f1,f2)
y0^2*y1^2 - 4*y0^3 - 4*y1^3 + 18*y0*y1 - 27
```

a simple example with constant rational coefficients:

```python
sage: R.<x,y,z,w> = PolynomialRing(QQ,4)
sage: w.macaulay_resultant([z,y,x])
1
```

an example where the resultant vanishes:

```python
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: (x+y).macaulay_resultant([y^2,x])
0
```

an example of bad reduction at a prime $p = 5$:

```python
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: y.macaulay_resultant([x^3+25*y^2*x,5*z])
125
```

The input can given as an unpacked list of polynomials:

```python
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: y.macaulay_resultant(x^3+25*y^2*x,5*z)
125
```

an example when the coefficients live in a finite field:

```python
sage: F = FiniteField(11)
sage: R.<x,y,z,w> = PolynomialRing(F,4)
```

(continues on next page)

3.1. Multivariate Polynomials and Polynomial Rings 295
example when the denominator in the algorithm vanishes (in this case the resultant is the constant term of the quotient of char polynomials of numerator/denominator):

```python
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: y.macaulay_resultant([x+z, z^2])
-1
```

when there are only 2 polynomials, macaulay resultant degenerates to the traditional resultant:

```python
sage: R.<x> = PolynomialRing(QQ,1)
sage: f = x^5 + 1; g = x^5 + 1
sage: f.resultant(g) == f.macaulay_resultant(g)
True
```

```python
map_coefficients(f, new_base_ring=None)
```

Returns the polynomial obtained by applying $f$ to the non-zero coefficients of self.

If $f$ is a `sage.categories.map.Map`, then the resulting polynomial will be defined over the codomain of $f$. Otherwise, the resulting polynomial will be over the same ring as self. Set `new_base_ring` to override this behaviour.

**INPUT:**

- $f$ – a callable that will be applied to the coefficients of self.
- `new_base_ring` (optional) – if given, the resulting polynomial will be defined over this ring.

**EXAMPLES:**

```python
sage: k.<a> = GF(9); R.<x,y> = k[]; f = x*a + 2*x^3*y*a + a
sage: f.map_coefficients(lambda a : a + 1)
(-a + 1)*x^3*y + (a + 1)*x + (a + 1)
```

Examples with different base ring:

```python
sage: R.<x> = GF(9); S.<s> = GF(81)
sage: h = Hom(R,S)[0]; h
Ring morphism:
  From: Finite Field in r of size 3^2
  To:  Finite Field in s of size 3^4
  Defn: r |---> 2*s^3 + 2*s^2 + 1
sage: T.<X,Y> = R[]
sage: f = r*X+Y
sage: f = f.map_coefficients(h);
(-a + 1)*x^3*y + (a + 1)*x + (a + 1)
```

(continues on next page)
\texttt{X - Y}
\texttt{sage: g.parent()}
Multivariate Polynomial Ring in X, Y over Finite Field in r of size 3^2
\texttt{sage: g = f.map_coefficients(h, new_base_ring=GF(3)); g}
\texttt{X - Y}
\texttt{sage: g.parent()}
Multivariate Polynomial Ring in X, Y over Finite Field of size 3

\textbf{newton_polytope()}

Return the Newton polytope of this polynomial.

\textbf{EXAMPLES:}

\texttt{sage: R.<x,y> = QQ[]}
\texttt{sage: f = 1 + x*y + x^3 + y^3}
\texttt{sage: P = f.newton_polytope()}
\texttt{sage: P}
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3 vertices
\texttt{sage: P.is_simple()}
True

\textbf{nth_root(n)}

Return a \( n \)-th root of this element.

If there is no such root, a \texttt{ValueError} is raised.

\textbf{EXAMPLES:}

\texttt{sage: R.<x,y,z> = QQ[]}
\texttt{sage: a = 32 * (x*y + 1)^5 * (x+y+z)^5}
\texttt{sage: a.nth_root(5)}
\texttt{2*x^2*y + 2*x*y^2 + 2*x*y*z + 2*x + 2*y + 2*z}
\texttt{sage: b = x + 2*y + 3*z}
\texttt{sage: b.nth_root(42)}
\texttt{Traceback (most recent call last):
... 
ValueError: not a 42nd power}

\texttt{sage: R.<x,y> = QQ[]}
\texttt{sage: S.<z,t> = R[]}
\texttt{sage: T.<u,v> = S[]}
\texttt{sage: p = (1 + x*u + y + v) * (1 + z*t)}
\texttt{sage: (p**3).nth_root(3)}
\texttt{(x*z*t + x)*u + (z*t + 1)*v + (y + 1)*z*t + y + 1}
\texttt{sage: (p**3).nth_root(3).parent().is p.parent()}
True
\texttt{sage: ((1+x+z+t)**2).nth_root(3)}
\texttt{Traceback (most recent call last):
... 
ValueError: not a 3rd power}

\textbf{numerator()}

Return a numerator of self computed as self * self.denominator()

Note that some subclasses may implement its own numerator function.
Warning: This is not the numerator of the rational function defined by self, which would always be self since self is a polynomial.

EXAMPLES:

First we compute the numerator of a polynomial with integer coefficients, which is of course self.

```
sage: R.<x, y> = ZZ[]
sage: f = x^3 + 17*x + y + 1
sage: f.numerator()
x^3 + 17*x + y + 1
sage: f == f.numerator()
True
```

Next we compute the numerator of a polynomial over a number field.

```
sage: R.<x,y> = NumberField(symbolic_expression(x^2+3) ,'a')['x,y']
sage: f = (1/17)*y^19 - (2/3)*x + 1/3; f
1/17*y^19 - 2/3*x + 1/3
sage: f.numerator()
3*y^19 - 34*x + 17
sage: f == f.numerator()
False
```

We try to compute the numerator of a polynomial with coefficients in the finite field of 3 elements.

```
sage: K.<x,y,z> = GF(3)['x, y, z']
sage: f = 2*x*z + 2*z^2 + 2*y + 1; f
-x*z - z^2 - y + 1
sage: f.numerator()
-x*z - z^2 - y + 1
```

We check that the computation the numerator and denominator are valid

```
sage: K=NumberField(symbolic_expression('x^3+2'), 'a')['x']['s,t']
sage: f=K.random_element()
sage: f.numerator() / f.denominator() == f
True
```

```
polynomial(var)
```

Let var be one of the variables of the parent of self. This returns self viewed as a univariate polynomial in var over the polynomial ring generated by all the other variables of the parent.

EXAMPLES:

```
sage: R.<x,w,z> = QQ[]
sage: f = x^3 + 3*w^3*x + w^5 + (17*w^3)*x + z^5
sage: f.polynomial(x)
x^3 + (17*w^3 + 3*w)*x + w^5 + z^5
sage: parent(f.polynomial(x))
```

Univariate Polynomial Ring in x over Multivariate Polynomial Ring in w, z over Rational Field

```
sage: f.polynomial(w)
w^5 + 17*x*w^3 + 3*x^2*w + z^5 + x^3
sage: f.polynomial(z)
z^5 + w^5 + 17*x*w^3 + x^3 + 3*x^2*w
sage: R.<x,w,z,k> = ZZ[]
sage: f = x^3 + 3*w*x + w^5 + (17*w^3)*x + z^5 + x*w*z*k + 5
sage: f.polynomial(x)
x^3 + (17*w^3 + w*z*k + 3*w)*x + w^5 + z^5 + 5
sage: f.polynomial(w)
w^5 + 17*x*w^3 + (x*z*k + 3*x)*w + z^5 + x^3 + 5
sage: f.polynomial(z)
z^5 + x*w*k*z + w^5 + 17*x*w^3 + x^3 + 3*x^2*w + 5
sage: R.<x,y>=GF(5)[]
sage: f=x^2+x+y
sage: f.polynomial(x)
x^2 + x + y
sage: f.polynomial(y)
y + x^2 + x
```

reduced_form(**kwds)

Return a reduced form of this polynomial.

The algorithm is from Stoll and Cremona’s “On the Reduction Theory of Binary Forms” [CS2003]. This takes a two variable homogeneous polynomial and finds a reduced form. This is a $SL(2, \mathbb{Z})$-equivalent binary form whose covariant in the upper half plane is in the fundamental domain. If the polynomial has multiple roots, they are removed and the algorithm is applied to the portion without multiple roots.

This reduction should also minimize the sum of the squares of the coefficients, but this is not always the case. By default the coefficient minimizing algorithm in [HS2018] is applied. The coefficients can be minimized either with respect to the sum of their squares or the maximum of their global heights.

A portion of the algorithm uses Newton’s method to find a solution to a system of equations. If Newton’s method fails to converge to a point in the upper half plane, the function will use the less precise $z_0$ covariant from the $Q_0$ form as defined on page 7 of [CS2003]. Additionally, if this polynomial has a root with multiplicity at least half the total degree of the polynomial, then we must also use the $z_0$ covariant. See [CS2003] for details.

Note that, if the covariant is within error_limit of the boundary but outside the fundamental domain, our function will erroneously move it to within the fundamental domain, hence our conjugation will be off by 1. If you don’t want this to happen, decrease your error_limit and increase your precision.

Implemented by Rebecca Lauren Miller as part of GSOC 2016. Smallest coefficients added by Ben Hutz July 2018.

INPUT:

keywords:

- prec – integer, sets the precision (default:300)
- return_conjugation – boolean. Returns element of $SL(2, \mathbb{Z})$ (default:True)
- error_limit – sets the error tolerance (default:0.000001)
• **smallest_coeffs** – (default: True), boolean, whether to find the model with smallest coefficients
• **norm_type** – either 'norm' or 'height'. What type of norm to use for smallest coefficients
• **emb** – (optional) embedding of based field into CC

**OUTPUT:**
• a polynomial (reduced binary form)
• a matrix (element of $SL(2, \mathbb{Z})$)

**TODO:** When Newton’s Method doesn’t converge to a root in the upper half plane. Now we just return $z_0$. It would be better to modify and find the unique root in the upper half plane.

**EXAMPLES:**

```sage
code not shown
```

An example where the multiplicity is too high:

```sage
code not shown
```

An example where Newton’s Method does not find the right root:

```sage
code not shown
```

An example with covariant on the boundary, therefore a non-unique form:

```sage
code not shown
```
An example where precision needs to be increased:

```python
sage: R.<x,y> = PolynomialRing(QQ)
sage: F=-16*x^7 - 114*x^6*y - 345*x^5*y^2 - 599*x^4*y^3 - 666*x^3*y^4 - 481*x^2*y^5 - 207*x*y^6 - 40*y^7
sage: F.reduced_form(prec=50, smallest_coeffs=False)
Traceback (most recent call last):
  ... ValueError: accuracy of Newton's root not within tolerance(0.0000124... > 1e-06), increase precision
sage: F.reduced_form(prec=100, smallest_coeffs=False)
([-1 -1]
 -x^5*y^2 - 24*x^3*y^4 - 3*x^2*y^5 - 2*x*y^6 + 16*y^7, [ 1  0])
```

```python
sage: R.<x,y> = PolynomialRing(QQ)
sage: F = - 8*x^4 - 3933*x^3*y - 725085*x^2*y^2 - 59411592*x*y^3 - 99*y^6
sage: F.reduced_form(return_conjugation=False)
x^4 + 9*x^3*y - 3*x*y^3 - 8*y^4
```

```python
sage: R.<x,y> = QQ[]
sage: F = -2*x^3 + 2*x^2*y + 3*x*y^2 + 127*y^3
sage: F.reduced_form()
([-2 -2]
 -2*x^3 - 22*x^2*y - 77*x*y^2 + 43*y^3, [0 1])
```

```python
sage: R.<x,y> = QQ[]
sage: F = -2*x^3 + 2*x^2*y + 3*x*y^2 + 127*y^3
sage: F.reduced_form(norm_type='height')
([-58 -58]
 -58*x^3 - 47*x^2*y + 52*x*y^2 + 43*y^3, [1 1])
```

```python
sage: R.<x,y,z> = PolynomialRing(QQ)
sage: F = x^4 + x^3*y*z + y^2*z
sage: F.reduced_form()
Traceback (most recent call last):
  ... ValueError: (=x^3*y*z + x^4 + y^2*z) must have two variables
```

```python
sage: R.<x,y> = PolynomialRing(ZZ)
sage: F = - 8*x^6 - 3933*x^3*y - 725085*x^2*y^2 - 59411592*x*y^3 - 99*y^6
sage: F.reduced_form(return_conjugation=False)
Traceback (most recent call last):
  ... ValueError: (=-8*x^6 - 99*y^6 - 3933*x^3*y - 725085*x^2*y^2 - 59411592*x*y^3) must be homogeneous
```


```python
sage: R.<x,y> = PolynomialRing(RR)
sage: F = 217.992172373276*x^3 + 96023.1505442490*x^2*y + 1.
      + 6.900160271132168*y^3
sage: F.reduced_form(smallest_coeffs=False) # tol 1e-8
([-39.5673942565918*x^3 + 111.874026298523*x^2*y + 231.052762985229*x*y^2 - 138.
      + 388829811096*y^3,
   [-147  -148]
   [  1   1])
```

```python
sage: R.<x,y> = PolynomialRing(CC)
sage: F = (0.759099196558145 + 0.845425869641446*CC.0)*x^3 + (84.8317207268542 +
      + 93.8840848648033*CC.0)*x^2*y + (3159.07040755858 + 3475.33037377779*CC.0)*x*y^2 + (39202.5965389079 + 42882.
      + 513729462*CC.0)*y^3
sage: F.reduced_form(smallest_coeffs=False) # tol 1e-11
([-0.759099196558145 - 0.845425869641446*I)*x^3 + (-0.571709908900118 - 0.
      + 754252314465703*I)*y^3,
   [-1  37]
   [ 0  -1])
```

### specialization($D=None, phi=None$)

Specialization of this polynomial.

Given a family of polynomials defined over a polynomial ring. A specialization is a particular member of that family. The specialization can be specified either by a dictionary or a `SpecializationMorphism`.

**INPUT:**

- **D** – dictionary (optional)
- **phi** – `SpecializationMorphism` (optional)

**OUTPUT:** a new polynomial

**EXAMPLES:**

```python
sage: R.<c> = PolynomialRing(QQ)
sage: S.<x,y> = PolynomialRing(R)
sage: F = x^2 + c*y^2
sage: F.specialization({c:2})
x^2 + 2*y^2
```

```python
sage: S.<a,b> = PolynomialRing(QQ)
sage: P.<x,y,z> = PolynomialRing(S)
sage: RR.<c,d> = PolynomialRing(P)
sage: f = a^2*x^2 + b*y^3 + c*y^2 - b*a*d + d^2 - a*c*b*z^2
```

sage: f.specialization({a:2, z:4, d:2})
(y^2 - 32*b)*c + b*y^3 + 2*x^2 - 4*b + 4

Check that we preserve multi- versus uni-variate:

sage: R.<l> = PolynomialRing(QQ, 1)
sage: S.<k> = PolynomialRing(R)
sage: K.<a, b, c> = PolynomialRing(S)
sage: F = a*k^2 + b*l + c^2
sage: F.specialization({b:56, c:5}).parent()
Univariate Polynomial Ring in a over Univariate Polynomial Ring in k
over Multivariate Polynomial Ring in l over Rational Field

subresultants\((other, variable=None)\)

Return the nonzero subresultant polynomials of \(self\) and \(other\).

INPUT:

• other – a polynomial

OUTPUT: a list of polynomials in the same ring as \(self\)

EXAMPLES:

sage: R.<x,y> = QQ[]
sage: p = (y^2 + 6)*(x - 1) - y*(x^2 + 1)
sage: q = (x^2 + 6)*(y - 1) - x*(y^2 + 1)
sage: p.subresultants(q, y)
[2*x^6 - 22*x^5 + 102*x^4 - 274*x^3 + 488*x^2 - 552*x + 288,
-x^3 - x^2*y + 6*x^2 + 5*x*y - 11*x - 6*y + 6]
sage: p.subresultants(q, x)
[2*y^6 - 22*y^5 + 102*y^4 - 274*y^3 + 488*y^2 - 552*y + 288,
x*y^2 + y^3 - 5*x*y - 6*y^2 + 6*x + 11*y - 6]

sylvester_matrix\((right, variable=None)\)

Given two nonzero polynomials \(self\) and \(right\), returns the Sylvester matrix of the polynomials with respect to a given variable.

Note that the Sylvester matrix is not defined if one of the polynomials is zero.

INPUT:

• self, right: multivariate polynomials

• variable: optional, compute the Sylvester matrix with respect to this variable. If variable is not provided, the first variable of the polynomial ring is used.

OUTPUT:

• The Sylvester matrix of \(self\) and \(right\).

EXAMPLES:

sage: R.<x, y> = PolynomialRing(ZZ)
sage: f = (y + 1)*x + 3*x^2
sage: g = (y + 2)*x + 4*x^2
sage: M = f.sylvester_matrix(g, x)
sage: M

(continues on next page)
If the polynomials share a non-constant common factor then the determinant of the Sylvester matrix will be zero:

```
sage: M.determinant()
0
sage: f.sylvester_matrix(1 + g, x).determinant()
y^2 - y + 7
```

If both polynomials are of positive degree with respect to variable, the determinant of the Sylvester matrix is the resultant:

```
sage: f = R.random_element(4)
sage: g = R.random_element(4)
sage: f.sylvester_matrix(g, x).determinant() == f.resultant(g, x)
True
```

`truncate(var, n)`
Returns a new multivariate polynomial obtained from self by deleting all terms that involve the given variable to a power at least n.

`weighted_degree(*weights)`
Return the weighted degree of self, which is the maximum weighted degree of all monomials in self; the weighted degree of a monomial is the sum of all powers of the variables in the monomial, each power multiplied with its respective weight in weights.

This method is given for convenience. It is faster to use polynomial rings with weighted term orders and the standard `degree` function.

**INPUT:**
- `weights` - Either individual numbers, an iterable or a dictionary, specifying the weights of each variable. If it is a dictionary, it maps each variable of self to its weight. If it is a sequence of individual numbers or a tuple, the weights are specified in the order of the generators as given by `self.parent().gens()`.

**EXAMPLES:**
```
sage: R.<x,y,z> = GF(7)[]
sage: p = x^3 + y + x^2*z^2
sage: p.weighted_degree({z:0, x:1, y:2})
3
sage: p.weighted_degree(1, 2, 0)
3
sage: p.weighted_degree((1, 4, 2))
5
sage: p.weighted_degree((1, 4, 1))
4
sage: p.weighted_degree(2**64, 2**50, 2**128)
68056473384186926945195958937245974528
```
You may also work with negative weights

```python
sage: p.weighted_degree(-1, -2, -1)
-2
```

Note that only integer weights are allowed

```python
sage: p.weighted_degree(x,1,1)
Traceback (most recent call last):
  ...TypeError: unable to convert non-constant polynomial x to Integer Ring
sage: p.weighted_degree(2/1,1,1)
6
```

The `weighted_degree` coincides with the `degree` of a weighted polynomial ring, but the later is faster.

```python
sage: K = PolynomialRing(QQ, 'x,y', order=TermOrder('wdegrevlex', (2,3)))
sage: p = K.random_element(10)
sage: p.degree() == p.weighted_degree(2,3)
True
```

sage.rings.polynomial.multi_polynomial.is_MPolynomial(x)

### 3.1.4 Multivariate Polynomial Rings over Generic Rings

Sage implements multivariate polynomial rings through several backends. This generic implementation uses the classes `PolyDict` and `ETuple` to construct a dictionary with exponent tuples as keys and coefficients as values.

**AUTHORS:**

- David Joyner and William Stein
- Kiran S. Kedlaya (2006-02-12): added Macaulay2 analogues of Singular features
- Martin Albrecht (2006-04-21): reorganize class hierarchy for singular rep
- Martin Albrecht (2007-04-20): reorganized class hierarchy to support Pyrex implementations

**EXAMPLES:**

We construct the Frobenius morphism on \( \mathbb{F}_5[x, y, z] \) over \( \mathbb{F}_5 \):

```python
sage: R.<x,y,z> = GF(5)[]
sage: frob = R.hom([x^5, y^5, z^5])
sage: frob(x^2 + 2*y - z^4)
-z^20 + x^10 + 2*y^5
sage: frob((x + 2*y)^3)
x^15 + x^10*y^5 + 2*x^5*y^10 - 2*y^15
sage: frob((x^5 + 2*y^5)^3)
x^15 + x^10*y^5 + 2*x^5*y^10 - 2*y^15
```
We make a polynomial ring in one variable over a polynomial ring in two variables:

```python
sage: R.<x, y> = PolynomialRing(QQ, 2)
sage: S.<t> = PowerSeriesRing(R)
sage: t*(x+y)
(x + y)*t
```

```python
class sage.rings.polynomial.multi_polynomial_ring.MPolynomialRing_macaulay2_repr
    Bases: object

    A mixin class for polynomial rings that support conversion to Macaulay2.

class sage.rings.polynomial.multi_polynomial_ring.MPolynomialRing_polydict(base_ring, n, names, order)
    Bases: sage.rings.polynomial.multi_polynomial_ring.MPolynomialRing_macaulay2_repr,
            sage.rings.polynomial.polynomial_singular_interface.PolynomialRing_singular_repr,
            sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base

    Multivariable polynomial ring.

    EXAMPLES:

    sage: R = PolynomialRing(Integers(12), 'x', 5); R
    Multivariate Polynomial Ring in x0, x1, x2, x3, x4 over Ring of integers modulo 12
    sage: loads(R.dumps()) == R
    True

    monomial_all_divisors(t)
        Return a list of all monomials that divide t, coefficients are ignored.

        INPUT:

        • t - a monomial.

        OUTPUT: a list of monomials.

        EXAMPLES:

    sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_
    --polydict_domain
    sage: P.<x,y,z> = MPolynomialRing_polydict_domain(QQ,3, order='degrevlex')
    sage: P.monomial_all_divisors(x^2*z^3)
    [x^2, z, x^2*z, x^2*z^2, x^2*z^3, z^2, x*z^2, x^2*z^2, z^3, x*z^3, x^2*z^3]

    ALGORITHM: addwithcarry idea by Toon Segers

    monomial_divides(a, b)
        Return False if a does not divide b and True otherwise.

        INPUT:

        • a – monomial
        • b – monomial

        OUTPUT: Boolean

        EXAMPLES:
```
sage: P.<x,y,z> = PolynomialRing(ZZ,3, order='degrevlex')
sage: P.monomial_divides(x*y*z, x^3*y^2*z^4)
True
sage: P.monomial_divides(x^3*y^2*z^4, x*y*z)
False
```

### monomial_lcm(f, g)

LCM for monomials. Coefficients are ignored.

**INPUT:**

- f - monomial.
- g - monomial.

**OUTPUT:** monomial.

**EXAMPLES:**

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domain
sage: P.<x,y,z> = MPolynomialRing_polydict_domain(QQ,3, order='degrevlex')
sage: P.monomial_lcm(3/2*x*y, x)
x*y
```

### monomial_pairwise_prime(h, g)

Return True if h and g are pairwise prime.

Both are treated as monomials.

**INPUT:**

- h - monomial.
- g - monomial.

**OUTPUT:** Boolean.

**EXAMPLES:**

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domain
sage: P.<x,y,z> = MPolynomialRing_polydict_domain(QQ,3, order='degrevlex')
sage: P.monomial_pairwise_prime(x^2*z^3, y^4)
True
sage: P.monomial_pairwise_prime(1/2*x^3*y^2, 3/4*y^3)
False
```

### monomial_quotient(f, g, coeff=False)

Return \( f/g \), where both \( f \) and \( g \) are treated as monomials.

Coefficients are ignored by default.

**INPUT:**

- f - monomial.
- g - monomial.
- coeff - divide coefficients as well (default: False).
OUTPUT: monomial.

EXAMPLES:

```python
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_
    polydict_domain
sage: P.<x,y,z> = MPolynomialRing_polydict_domain(QQ, 3, order='degrevlex')
sage: P.monomial_quotient(3/2*x*y, x)
y
sage: P.monomial_quotient(3/2*x*y, 2*x, coeff=True)
3/4*y
```

Note: Assumes that the head term of f is a multiple of the head term of g and return the multiplicant m. If this rule is violated, funny things may happen.

\[ \text{monomial\_reduce}(f, G) \]

Try to find a \( g \) in \( G \) where \( g.\text{lm}() \) divides \( f \).

If found, \((\text{flt}, g)\) is returned, \((0,0)\) otherwise, where \( \text{flt} = f/g.\text{lm}() \). It is assumed that \( G \) is iterable and contains ONLY elements in this ring.

INPUT:
- \( f \) - monomial
- \( G \) - list/set of mpolynomials

EXAMPLES:

```python
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_
    polydict_domain
sage: P.<x,y,z> = MPolynomialRing_polydict_domain(QQ, 3, order='degrevlex')
sage: f = x*y^2
sage: G = [3/2*x^3 + y^2 + 1/2, 1/4*x*y + 2/7, P(1/2)]
sage: P.monomial_reduce(f,G)
(y, 1/4*x*y + 2/7)
```

```python
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_
    polydict_domain
sage: P.<x,y,z> = MPolynomialRing_polydict_domain(Zmod(23432), 3, order=
    'degrevlex')
sage: f = x*y^2
sage: G = [3*x^3 + y^2 + 2, 4*x*y + 7, P(2)]
sage: P.monomial_reduce(f,G)
(y, 4*x*y + 7)
```

class sage.rings.polynomial.multi_polynomial_ring.MPolynomialRing_polydict_domain(base_ring, n, names, order)

MPolynomialRing_polydict

ideal(*gens, **kwds)
Create an ideal in this polynomial ring.
3.1.5 Generic Multivariate Polynomials

AUTHORS:

• David Joyner: first version
• William Stein: use dict’s instead of lists
• Martin Albrecht malb@informatik.uni-bremen.de: some functions added
• Kiran S. Kedlaya (2006-02-12): added Macaulay2 analogues of some Singular features
• William Stein (2006-04-19): added e.g., f[1,3] to get coeff of \( xy^3 \); added examples of the new \( R.x,y = \) PolynomialRing(QQ,2) notation.
• Martin Albrecht: improved singular coercions (restructured class hierarchy) and added ETuples
• Robert Bradshaw (2007-08-14): added support for coercion of polynomials in a subset of variables (including multi-level univariate rings)
• Joel B. Mohler (2008-03): Refactored interactions with ETuples.

EXAMPLES:

We verify Lagrange’s four squares identity:

```sage
R.<a0,a1,a2,a3,b0,b1,b2,b3> = QQbar[]
(f0 = a0^2 + a1^2 + a2^2 + a3^2)* (f1 = b0^2 + b1^2 + b2^2 + b3^2) ==
(f2 = (a0*b0 - a1*b1 - a2*b2 - a3*b3)^2 + (a0*b1 + a1*b0 + a2*b3 - a3*b2)^2 +
(a0*b2 - a1*b3 + a2*b0 + a3*b1)^2 + (a0*b3 + a1*b2 - a2*b1 + a3*b0)^2)
True
```

```python
class sage.rings.polynomial.multi_polynomial_element.MPolynomial_element(parent, x)
    Bases: sage.rings.polynomial.multi_polynomial.MPolynomial

EXAMPLES:

sage: K.<cuberoot2> = NumberField(x^3 - 2)
sage: L.<cuberoot3> = K.extension(x^3 - 3)
sage: S.<sqrt2> = L.extension(x^2 - 2)
sage: S
Number Field in sqrt2 with defining polynomial x^2 - 2 over its base field
sage: P.<x,y,z> = PolynomialRing(S) # indirect doctest
```

```python
change_ring(R)
Change the base ring of this polynomial to R.

INPUT:

• R – ring or morphism.

OUTPUT: a new polynomial converted to R.

EXAMPLES:
```
```
```
sage: R.<x,y> = QQ[]
sage: f = x^2 + 5*y
sage: f.change_ring(GF(5))
x^2

sage: K.<w> = CyclotomicField(5)
sage: R.<x,y> = K[]
sage: f = x^2 + w*y
sage: f.change_ring(K.embeddings(QQbar)[1])
x^2 + (-0.8090169943749474? + 0.5877852522924731?*I)*y
```

**element()**

Return the number of non-zero coefficients of this polynomial.

This is also called weight, `hamming_weight()` or sparsity.

**EXAMPLES:**

```
sage: R.<x, y> = CC[]
sage: f = x^3 - y
sage: f.number_of_terms()
2
sage: R(0).number_of_terms()
0
sage: f = (x+y)^100
sage: f.number_of_terms()
101
```

The method `hamming_weight()` is an alias:

```
sage: f.hamming_weight()
101
```

**number_of_terms()**

Return the number of non-zero coefficients of this polynomial.

This is also called weight, `hamming_weight()` or sparsity.

**EXAMPLES:**

```
sage: R.<x, y> = CC[]
sage: f = x^3 - y
sage: f.number_of_terms()
2
sage: R(0).number_of_terms()
0
sage: f = (x+y)^100
sage: f.number_of_terms()
101
```

The method `hamming_weight()` is an alias:

```
sage: f.hamming_weight()
101
```
class sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict(parent, x)

Bases: sage.rings.polynomial.polynomial_singular_interface.Polynomial_singular_repr, sage.rings.polynomial.multi_polynomial_element.MPolynomial_element

Multivariate polynomials implemented in pure python using polydicts.

coefficient(degrees)

Return the coefficient of the variables with the degrees specified in the python dictionary degrees. Mathematically, this is the coefficient in the base ring adjoined by the variables of this ring not listed in degrees. However, the result has the same parent as this polynomial.

This function contrasts with the function monomial_coefficient which returns the coefficient in the base ring of a monomial.

INPUT:

• degrees - Can be any of:
  – a dictionary of degree restrictions
  – a list of degree restrictions (with None in the unrestricted variables)
  – a monomial (very fast, but not as flexible)

OUTPUT: element of the parent of self

See also:

For coefficients of specific monomials, look at monomial_coefficient().

EXAMPLES:

```python
sage: R.<x, y> = QQbar[]
sage: f = 2 * x * y
sage: c = f.coefficient({x:1,y:1}); c
2
sage: c.parent()
Multivariate Polynomial Ring in x, y over Algebraic Field
sage: c in PolynomialRing(QQbar, 2, names = ['x', 'y'])
True
sage: f = y^2 - x^9 - 7*x + 5*x*y
sage: f.coefficient({y:1})
5*x
sage: f.coefficient({y:0})
-x^9 + (-7)*x
sage: f.coefficient({x:0,y:0})
0
sage: f=(1+y+y^2)*(1+x+x^2)
```
```
sage: f.coefficient({x:0})
y^2 + y + 1
sage: f.coefficient(x^0) # outputs the full polynomial
x^2*y^2 + x^2*y + x^2*y^2 + x^2 + x*y + y^2 + x + y + 1
```

3.1. Multivariate Polynomials and Polynomial Rings 311
AUTHORS:

• Joel B. Mohler (2007-10-31)

code_list
constant_coefficient()
Return the constant coefficient of this multivariate polynomial.

EXAMPLES:
code_list
degree(x=None, std_grading=False)
Return the degree of self in x, where x must be one of the generators for the parent of self.

INPUT:

• x - multivariate polynomial (a generator of the parent of self). If x is not specified (or is None), return the total degree, which is the maximum degree of any monomial. Note that a weighted term ordering alters the grading of the generators of the ring; see the tests below. To avoid this behavior, set the optional argument std_grading=True.

OUTPUT: integer

EXAMPLES:
code_list

Note that total degree takes into account if we are working in a polynomial ring with a weighted term order.

sage: x, y = ZZ['x', 'y'].gens()
sage: GF(3037000453)['x', 'y'].gen(0).degree(x)
1
sage: x0, y0 = QQ['x', 'y'].gens()
sage: GF(3037000453)['x', 'y'].gen(0).degree(x0)
Traceback (most recent call last):
  ... TypeError: x must canonically coerce to parent
sage: GF(3037000453)['x', 'y'].gen(0).degree(x^2)
Traceback (most recent call last):
  ... TypeError: x must be one of the generators of the parent

degrees()

Returns a tuple (precisely - an ETuple) with the degree of each variable in this polynomial. The list of degrees is, of course, ordered by the order of the generators.

EXAMPLES:

sage: R.<x,y,z>=PolynomialRing(QQbar)
sage: f = 3^x^2 + 2^y + 7^x^2*y^2 + 5
sage: f.degrees()
(2, 2, 0)
sage: f = x^2+z^2
sage: f.degrees()
(2, 0, 2)
sage: f.total_degree()  # this simply illustrates that total degree is not the sum of the degrees
2
sage: R.<x,y,z,u>=PolynomialRing(QQbar)
sage: f=(1-x)*(1+y+z+x^3)^5
sage: f.degrees()
(16, 5, 5, 0)
sage: R(0).degrees()
(0, 0, 0, 0)
dict()
Return underlying dictionary with keys the exponents and values the coefficients of this polynomial.

exponents(as_ETuples=True)
Return the exponents of the monomials appearing in self.

INPUT:
• as_ETuples – (default: True): return the list of exponents as a list of ETuples

OUTPUT:
The list of exponents as a list of ETuples or tuples.

EXAMPLES:

```
sage: R.<a,b,c> = PolynomialRing(QQbar, 3)
sage: f = a^3 + b + 2*b^2
sage: f.exponents()
[(3, 0, 0), (0, 2, 0), (0, 1, 0)]
```

By default the list of exponents is a list of ETuples:

```
sage: type(f.exponents()[0])
<type 'sage.rings.polynomial.polydict.ETuple'>
sage: type(f.exponents(as_ETuples=False)[0])
<... 'tuple'>
```

factor(proof=None)
Compute the irreducible factorization of this polynomial.

INPUT:
• proof'' - insist on provably correct results (default: ´`True unless explicitly
disabled for the "polynomial" subsystem with sage.structure.proof.proof.WithProof.)

integral(var=None)
Integrates self with respect to variable var.

**Note:** The integral is always chosen so the constant term is 0.

If var is not one of the generators of this ring, integral(var) is called recursively on each coefficient of this
polynomial.

EXAMPLES:

On polynomials with rational coefficients:

```
sage: x, y = PolynomialRing(QQ, ´x, y´).gens()
sage: ex = x*y + x - y
sage: it = ex.integral(x); it
1/2*x^2*y + 1/2*x^2 - x*y
sage: it.parent() == x.parent()
True
```

On polynomials with coefficients in power series:
sage: R.<t> = PowerSeriesRing(QQbar)
sage: S.<x, y> = PolynomialRing(R)
sage: f = (t^2 + 0(t^3))*x^2*y^3 + (37*t^4 + 0(t^5))*x^3
sage: f.parent()
Multivariate Polynomial Ring in x, y over Power Series Ring in t over Algebraic
˓→Field
sage: f.integral(x)  # with respect to x
(1/3*t^2 + O(t^3))*x^3*y^3 + (37/4*t^4 + O(t^5))*x^4
sage: f.integral(x).parent()
Multivariate Polynomial Ring in x, y over Power Series Ring in t over Algebraic
˓→Field
sage: f.integral(y)  # with respect to y
(1/4*t^2 + O(t^3))*x^2*y^4 + (37*t^4 + O(t^5))*x^3*y
sage: f.integral(t)  # with respect to t (recurses into base ring)
(1/3*t^3 + O(t^4))*x^2*y^3 + (37/5*t^5 + O(t^6))*x^3

inverse_of_unit()
Return the inverse of a unit in a ring.

is_constant()
Return True if self is a constant and False otherwise.

EXAMPLES:

sage: R.<x,y> = QQbar[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.is_constant()
False
sage: g = 0^x^0
sage: g.is_constant()
True

is_generator()
Return True if self is a generator of its parent.

EXAMPLES:

sage: R.<x,y>=QQbar[]
sage: x.is_generator()
True
sage: (x+y-y).is_generator()
True
sage: (x*y).is_generator()
False

is_homogeneous()
Return True if self is a homogeneous polynomial.

EXAMPLES:

sage: R.<x,y> = QQbar[]
sage: (x+y).is_homogeneous()
True
sage: (x.parent()(0)).is_homogeneous()
True
sage: (x*y^2).is_homogeneous()
False
sage: (x^2 + y^2).is_homogeneous()
True
sage: (x^2 + y^2*x).is_homogeneous()
False
sage: (x^2*y + y^2*x).is_homogeneous()
True

is_monomial()
Return True if self is a monomial, which we define to be a product of generators with coefficient 1.
Use is_term() to allow the coefficient to not be 1.

EXAMPLES:
sage: R.<x,y>=QQbar[
  sage: x.is_monomial()
  True
  sage: (x+2*y).is_monomial()
  False
  sage: (2*x).is_monomial()
  False
  sage: (x*y).is_monomial()
  True

To allow a non-1 leading coefficient, use is_term():
sage: (2*x*y).is_term()
True
sage: (2*x*y).is_monomial()
False

is_term()
Return True if self is a term, which we define to be a product of generators times some coefficient, which
need not be 1.
Use is_monomial() to require that the coefficient be 1.

EXAMPLES:
sage: R.<x,y>=QQbar[
  sage: x.is_term()
  True
  sage: (x+2*y).is_term()
  False
  sage: (2*x).is_term()
  True
  sage: (7*x^5*y).is_term()
  True

To require leading coefficient 1, use is_monomial():
sage: (2*x^y).is_monomial()
False
sage: (2*x^y).is_term()
True

is_univariate()
Returns True if this multivariate polynomial is univariate and False otherwise.

EXAMPLES:
sage: R.<x,y> = QQbar[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.is_univariate()
False
sage: g = f.subs({x:10}); g
700*y^2 + (-2)*y + 305
sage: g.is_univariate()
True
sage: f = x^0
sage: f.is_univariate()
True

iterator_exp_coeff(as_ETuples=True)
Iterate over self as pairs of ((E)Tuple, coefficient).

INPUT:
• as_ETuples – (default: True) if True iterate over pairs whose first element is an ETuple, otherwise as a tuples

EXAMPLES:
sage: R.<x,y,z> = PolynomialRing(QQbar, order='lex')
sage: f = (x^1*y^5*z^2 + x^2*z + x^4*y^1*z^3)

sage: list(f.iterator_exp_coeff())
[((4, 1, 3), 1), ((2, 0, 1), 1), ((1, 5, 2), 1)]
sage: R.<x,y,z> = PolynomialRing(QQbar, order='deglex')
sage: f = (x^1*y^5*z^2 + x^2*z + x^4*y^1*z^3)

sage: list(f.iterator_exp_coeff(as_ETuples=False))
[((4, 1, 3), 1), ((1, 5, 2), 1), ((2, 0, 1), 1)]

lc()
Returns the leading coefficient of self i.e., self.coefficient(self.lm())

EXAMPLES:
sage: R.<x,y,z>=QQbar[]
sage: f=3*x^2-y^2-x*y
sage: f.lc()
3

lift(I)
given an ideal I = (f_1,...,f_r) and some g (== self) in I, find s_1,...,s_r such that g = s_1 f_1 + ... + s_r f_r

ALGORITHM: Use Singular.
EXAMPLES:

```python
sage: A.<x,y> = PolynomialRing(CC,2,order='degrevlex')
sage: I = A.ideal([x^10 + x^9*y^2, y^8 - x^2*y^7 ])
sage: f = x^y^13 + y^12
sage: M = f.lift(I)
sage: M
[y^7, x^7*y^2 + x^8 + x^5*y^3 + x^6*y + x^3*y^4 + x^4*y^2 + x^y^5 + x^2*y^3 + y^-
˓→4]
sage: sum( map( mul , zip( M, I.gens() ) ) ) == f
True
```

**lm()**

Returns the lead monomial of self with respect to the term order of self.parent().

EXAMPLES:

```python
sage: R.<x,y,z>=PolynomialRing(GF(7),3,order='lex')
sage: (x^1*y^2 + y^3*z^4).lm()
x^y^2
sage: (x^3*y^2*z^4 + x^3*y^2*z^1).lm()
x^3*y^2*z^4
sage: R.<x,y,z>=PolynomialRing(QQbar,3,order='deglex')
```

```python
sage: (x^1*y^2*z^3 + x^3*y^2*z^0).lm()
x^y^2*z^3
sage: (x^1*y^2*z^4 + x^1*y^1*z^5).lm()
x^y^2*z^4
```

```python
sage: R.<x,y,z>=PolynomialRing(QQbar,3,order='degrevlex')
sage: (x^1*y^5*z^2 + x^4*y^1*z^3).lm()
x^y^5*z^2
sage: (x^4*y^7*z^1 + x^4*y^2*z^3).lm()
x^y^7*z
```

**lt()**

Returns the leading term of self i.e., self.lc()*self.lm(). The notion of “leading term” depends on the ordering defined in the parent ring.

EXAMPLES:

```python
sage: R.<x,y,z>=PolynomialRing(QQbar)
sage: f=3*x^2-y^2-x*y
sage: f.lt()
3*x^2
sage: R.<x,y,z>=PolynomialRing(QQbar,order="invlex")
```

```python
sage: f=3*x^2-y^2-x*y
sage: f.lt()
-y^2
```

**monomial_coefficient(mon)**

Return the coefficient in the base ring of the monomial mon in self, where mon must have the same parent as self.

This function contrasts with the function coefficient which returns the coefficient of a monomial viewing this polynomial in a polynomial ring over a base ring having fewer variables.
INPUT:
* `mon` - a monomial

OUTPUT: coefficient in base ring

See also:
For coefficients in a base ring of fewer variables, look at `coefficient()`.

EXAMPLES:
The parent of the return is a member of the base ring.

```sage
c = f.monomial_coefficient(x*y); c
2
```

The parent of the return is a member of the base ring.

```sage:
f = y^2 + y^2*x - x^9 + 5*x*y
f.monomial_coefficient(y^2)
1
f.monomial_coefficient(x*y)
5
f.monomial_coefficient(x^9)
-1
f.monomial_coefficient(x^10)
0
```

```sage:
var('a')
a
K.<a> = NumberField(a^2+a+1)
P.<x,y> = K[]
f=(a*x-1)*((a+1)*y-1); f
-x*y + (-a)*x + (-a - 1)*y + 1
f.monomial_coefficient(x)
-a
```

`monomials()`
Returns the list of monomials in self. The returned list is decreasingly ordered by the term ordering of `self.parent()`.

OUTPUT: list of MPolynomials representing Monomials

EXAMPLES:

```sage:
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.monomials()
[x^2*y^2, x^2, y, 1]
```
sage: R.<fx, fy, gx, gy> = QQbar[]
sage: F = ((fx*gy - fy*gx)^3)
sage: F
\(-fy^3*gx^3 + 3*fx*fy^2*gx^2*gy + (-3)*fx^2*fy*gx*gy^2 + fx^3*gy^3\)
sage: F.monomials()
\([fy^3*gx^3, fx*fy^2*gx^2*gy, fx^2*fy*gx*gy^2, fx^3*gy^3]\)
sage: F.coefficients()
\([-1, 3, -3, 1]\)
sage: sum(map(mul, zip(F.coefficients(), F.monomials()))) == F
True

nvariables()
Number of variables in this polynomial

EXAMPLES:

sage: R.<x, y> = QQbar[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.nvariables()
2
sage: g = f.subs({x: 10}); g
700*y^2 + (-2)*y + 305
sage: g.nvariables()
1

quo_rem(right)
Returns quotient and remainder of self and right.

EXAMPLES:

sage: R.<x, y> = CC[]
sage: f = y^2*x^2 + x + 1
sage: f.quo_rem(x)
\((x*y + 1.00000000000000, 1.00000000000000)\)
sage: R = QQ['a', 'b']['x', 'y', 'z']
sage: p1 = R('a + (1+2*b)*x*y + (3-a^2)*z')
sage: p2 = R('x-1')
sage: p1.quo_rem(p2)
\((2*b + 1)*y, (2*b + 1)*y + (-a^2 + 3)*z + a)\)
sage: R.<x, y> = Qp(5)[]
sage: x.quo_rem(y)
Traceback (most recent call last):
...
TypeError: no conversion of this ring to a Singular ring defined

ALGORITHM: Use Singular.

reduce(I)
Reduce this polynomial by the polynomials in I.

INPUT:

- I - a list of polynomials or an ideal

EXAMPLES:
```python
sage: P.<x,y,z> = QQbar[]
sage: f1 = -2 * x^2 + x^3
sage: f2 = -2 * y + x*y
sage: f3 = -x^2 + y^2
sage: F = Ideal([f1,f2,f3])
sage: g = x*y - 3*x*y^2
sage: g.reduce(F)
(-6)*y^2 + 2*y
sage: g.reduce(F.gens())
(-6)*y^2 + 2*y
sage: f = 3*x
sage: f.reduce([2*x,y])
0

sage: k.<w> = CyclotomicField(3)
sage: A.<y9,y12,y13,y15> = PolynomialRing(k)
sage: J = [ y9 + y12]
sage: f = y9 - y12; f.reduce(J)
-2*y12
sage: f = y13*y15; f.reduce(J)
y13*y15
sage: f = y13*y15 + y9 - y12; f.reduce(J)
y13*y15 - 2*y12

Make sure the remainder returns the correct type, fixing trac ticket #13903:

```python
sage: R.<y1,y2>=PolynomialRing(Qp(5),2, order='lex')
sage: G=[y1^2 + y2^2, y1*y2 + y2^2, y2^3]
sage: type((y2^3).reduce(G))
<class 'sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict'>
```

**resultant**(other, variable=None)

Compute the resultant of self and other with respect to variable.

If a second argument is not provided, the first variable of self.parent() is chosen.

For inexact rings or rings not available in Singular, this computes the determinant of the Sylvester matrix.

**INPUT:**

- other – polynomial in self.parent()
- variable – (optional) variable (of type polynomial) in self.parent()

**EXAMPLES:**

```python
sage: P.<x,y> = PolynomialRing(QQ, 2)
sage: a = x + y
sage: b = x^3 - y^3
sage: a.resultant(b)
-2*y^3
sage: a.resultant(b, y)
2*x^3
```

---

3.1. Multivariate Polynomials and Polynomial Rings 321
**subresultants**(*other, variable=None*)

Return the nonzero subresultant polynomials of `self` and `other`.

**INPUT:**

- `other` – a polynomial

**OUTPUT:** a list of polynomials in the same ring as `self`

**EXAMPLES:**

```sage
csage: R.<x,y> = QQbar[]
csage: p = (y^2 + 6)*(x - 1) - y*(x^2 + 1)
csage: q = (x^2 + 6)*(y - 1) - x*(y^2 + 1)
csage: p.subresultants(q, y)
[2*x^6 + (-22)*x^5 + 102*x^4 + (-274)*x^3 + 488*x^2 + (-552)*x + 288,
 -x^3 - x^2*y + 6*x^2 + 5*x*y + (-11)*x + (-6)*y + 6]
csage: p.subresultants(q, x)
[2*y^6 + (-22)*y^5 + 102*y^4 + (-274)*y^3 + 488*y^2 + (-552)*y + 288,
 x*y^2 + y^3 + (-5)*x*y + (-6)*y^2 + 6*x + 11*y - 6]
```

**subs**(*fixed=None, **kw*)

Fixes some given variables in a given multivariate polynomial and returns the changed multivariate polynomials. The polynomial itself is not affected. The variable,value pairs for fixing are to be provided as a dictionary of the form `{variable:value}`.

This is a special case of evaluating the polynomial with some of the variables constants and the others the original variables.

**INPUT:**

- `fixed` - (optional) dictionary of inputs
- `**kw` - named parameters

**OUTPUT:** new MPolynomial

**EXAMPLES:**

```sage
csage: R.<x,y> = QQbar[]
csage: f = x^2 + y + x^2*y^2 + 5
csage: f((5,y))
25*y^2 + y + 30
csage: f.subs({x:5})
25*y^2 + y + 30
```

**total_degree**()

Return the total degree of `self`, which is the maximum degree of any monomial in `self`.

**EXAMPLES:**

```sage
csage: R.<x,y,z> = QQbar[]
csage: f=2*x*y^3*z^2
csage: f.total_degree()
6
csage: f=4*x^2*y^2*z^3
csage: f.total_degree()
7
csage: f=99*x^6*y^3*z^9
```
sage: f.total_degree()
18
sage: f=x*y^3*z^6+3*x^2
sage: f.total_degree()
10
sage: f=z^3+8*x^4*y^5+z
sage: f.total_degree()
10
sage: f=z^9+10*x^4+y^8*x^2
sage: f.total_degree()
10

univariate_polynomial(R=None)
Returns a univariate polynomial associated to this multivariate polynomial.

INPUT:

• R - (default: None) PolynomialRing

If this polynomial is not in at most one variable, then a ValueError exception is raised. This is checked using the is_univariate() method. The new Polynomial is over the same base ring as the given MPolynomial.

EXAMPLES:

sage: R.<x,y> = QQbar[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.univariate_polynomial()
Traceback (most recent call last):
...
TypeError: polynomial must involve at most one variable
sage: g = f.subs({x:10}); g
700*y^2 + (-2)*y + 305
sage: g.univariate_polynomial()
700*y^2 - 2*y + 305
sage: g.univariate_polynomial(PolynomialRing(QQ,'z'))
700*z^2 - 2*z + 305

variable(i)
Returns i-th variable occurring in this polynomial.

EXAMPLES:

sage: R.<x,y> = QQbar[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.variable(0)
x
sage: f.variable(1)
y

variables()
Returns the tuple of variables occurring in this polynomial.

EXAMPLES:

sage: R.<x,y> = QQbar[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5

(continues on next page)

sage: f.variables()
(x, y)
sage: g = f.subs({x:10}); g
700*y^2 + (-2)*y + 305
sage: g.variables()
(y,)

sage.rings.polynomial.multi_polynomial_element.degree_lowest_rational_function(r, x)
Return the difference of valuations of r with respect to variable x.

INPUT:
• r – a multivariate rational function
• x – a multivariate polynomial ring generator x

OUTPUT:
• integer – the difference \( \text{val}_x(p) - \text{val}_x(q) \) where \( r = \frac{p}{q} \)

Note: This function should be made a method of the FractionFieldElement class.

EXAMPLES:

sage: R1 = PolynomialRing(FiniteField(5), 3, names=['a','b','c'])
sage: F = FractionField(R1)
sage: a,b,c = R1.gens()
sage: f = 3*a*b^2*c^3+4*a*b*c
sage: g = a^2*b*c^2+2*a^2*b^4*c^7

Consider the quotient \( f/g = \frac{4+3b^2}{a^2+2ab^3} \) (note the cancellation).

sage: r = f/g; r
(-2*b^2 - 1)/(2*a^2*b^3 + 3*a*b + a*c)
sage: degree_lowest_rational_function(r,a)
-1
sage: degree_lowest_rational_function(r,b)
0
sage: degree_lowest_rational_function(r,c)
-1

sage.rings.polynomial.multi_polynomial_element.is_MPolynomial(x)

3.1.6 Ideals in multivariate polynomial rings

Sage has a powerful system to compute with multivariate polynomial rings. Most algorithms dealing with these ideals are centered on the computation of Groebner bases. Sage mainly uses Singular to implement this functionality. Singular is widely regarded as the best open-source system for Groebner basis calculation in multivariate polynomial rings over fields.

EXAMPLES:

We compute a Groebner basis for some given ideal. The type returned by the groebner_basis method is PolynomialSequence, i.e. it is not a MPolynomialIdeal:
Groebner bases can be used to solve the ideal membership problem:

\[
\text{sage: } f, g, h = B \\
\text{sage: } (2 \times f + g).\text{reduce}(B) \\
\text{0}
\]

\[
\text{sage: } (2 \times f + g) \text{ in } I \\
\text{True}
\]

\[
\text{sage: } (2 \times f + 2 \times h + y^3).\text{reduce}(B) \\
y^3
\]

\[
\text{sage: } (2 \times f + 2 \times h + y^3) \text{ in } I \\
\text{False}
\]

We compute a Groebner basis for Cyclic 6, which is a standard benchmark and test ideal.

\[
\text{sage: } R.<x,y,z,t,u,v> = \mathbb{Q}[x,y,z,t,u,v] \\
\text{sage: } I = \text{sage.rings.ideal.Cyclic}(R,6) \\
\text{sage: } B = I.\text{groebner_basis()} \\
\text{sage: } \text{len}(B) \\
45
\]

We compute in a quotient of a polynomial ring over \( \mathbb{Z}/17\mathbb{Z} \):

\[
\text{sage: } R.<x,y> = \mathbb{Z}[] \\
\text{sage: } S.<a,b> = R.\text{quotient}((x^2 + y^2, 17)) \\
\text{sage: } S \\
\text{Quotient of Multivariate Polynomial Ring in x, y over Integer Ring by the ideal (x^2 + y^2, 17)}
\]

\[
\text{sage: } a^2 + b^2 == 0 \\
\text{True}
\]

\[
\text{sage: } a^3 - b^2 \\
-a*b^2 - b^2
\]

Note that the result of a computation is not necessarily reduced:

\[
\text{sage: } (a+b)^{17} \\
256*a*b^{16} + 256*b^17
\]

\[
\text{sage: } S(17) == 0 \\
\text{True}
\]

Or we can work with \( \mathbb{Z}/17\mathbb{Z} \) directly:

\[
\text{sage: } R.<x,y> = \text{Zmod}(17)[] \\
\text{sage: } S.<a,b> = R.\text{quotient}((x^2 + y^2,)) \\
\text{sage: } S
\]
Quotient of Multivariate Polynomial Ring in x, y over Ring of integers modulo 17 by the ideal (x^2 + y^2)

```
sage: a^2 + b^2 == 0
True
sage: a^3 - b^2 == -a*b^2 - b^2 == 16*a*b^2 + 16*b^2
True
sage: (a+b)^17
a*b^16 + b^17
sage: S(17) == 0
True
```

Working with a polynomial ring over \( \mathbb{Z} \):

```
sage: R.<x,y,z,w> = ZZ[]
sage: I = ideal(x^2 + y^2 - z^2 - w^2, x-y)
sage: J = I^2
sage: J.groebner_basis()
[4*y^4 - 4*y^2*z^2 + z^4 - 4*y^2*w^2 + 2*z^2*w^2 + w^4,
  2*x*y^2 - 2*y^3 - x*z^2 + y*z^2 - x*w^2 + y*w^2,
  x^2 - x*y + y^2]

sage: y^2 - 2*x*y + x^2 in J
True
sage: 0 in J
True
```

We do a Groebner basis computation over a number field:

```
sage: K.<zeta> = CyclotomicField(3)
sage: R.<x,y,z> = K[]; R
Multivariate Polynomial Ring in x, y, z over Cyclotomic Field of order 3 and degree 2

sage: i = ideal(x - zeta*y + 1, x^3 - zeta*y^3); i
Ideal (x + (-zeta)*y + 1, x^3 + (-zeta)*y^3) of Multivariate Polynomial Ring in x, y, z over Cyclotomic Field of order 3 and degree 2

sage: i.groebner_basis()
[y^3 + (2*zeta + 1)*y^2 + (zeta - 1)*y + (-1/3*zeta - 2/3), x + (-zeta)*y + 1]

sage: S = R.quotient(i); S
Quotient of Multivariate Polynomial Ring in x, y, z over Cyclotomic Field of order 3 and degree 2 by the ideal (x + (-zeta)*y + 1, x^3 + (-zeta)*y^3)

sage: S.0 - zeta*S.1
-1
sage: S.0^3 - zeta*S.1^3
0
```

Two examples from the Mathematica documentation (done in Sage):

We compute a Groebner basis:
We show that three polynomials have no common root:

```python
sage: R.<x,y> = QQ[]
sage: ideal(x^2 - 2*y^2, x*y - 3).groebner_basis()
[1]
```

The next example shows how we can use Groebner bases over \( \mathbb{Z} \) to find the primes modulo which a system of equations has a solution, when the system has no solutions over the rationals.

We first form a certain ideal \( I \) in \( \mathbb{Z}[x, y, z] \), and note that the Groebner basis of \( I \) over \( \mathbb{Q} \) contains 1, so there are no solutions over \( \mathbb{Q} \) or an algebraic closure of it (this is not surprising as there are 4 equations in 3 unknowns).

```python
sage: P.<x,y,z> = PolynomialRing(ZZ,order='lex')
sage: I = ideal(-y^2 - 3*y + z^2 + 3, -2*y*z + z^2 + 2*z + 1, 
x^2 + y^2 + 3*y + 17220, y^2 + 3*y + 26532, 2*y + 15864, z^2 +
˓→17220, 2*z + 41856, 164878)
sage: I.groebner_basis()
[x + y + 57119*z + 4, y^2 + 3*y + 17220, y*z + y + 26532, 2*y + 15864, z^2 +]
```

Now for each prime \( p \) dividing this integer 164878, the Groebner basis of \( I \) modulo \( p \) will be non-trivial and will thus give a solution of the original system modulo \( p \).

```python
sage: factor(164878)
2 * 7 * 11777
sage: I.change_ring(P.change_ring( GF(2) )).groebner_basis()
[x + y + z, y^2 + y, y*z + y, z^2 + 1]
sage: I.change_ring(P.change_ring( GF(7) )).groebner_basis()
[x - 1, y + 3, z - 2]
sage: I.change_ring(P.change_ring( GF(11777 ))).groebner_basis()
[x + 5633, y - 3007, z - 2626]
```

The Groebner basis modulo any product of the prime factors is also non-trivial:

```python
sage: I.change_ring(P.change_ring( IntegerModRing(2*7) )).groebner_basis()
[x + 9*y + 13*z, y^2 + 3*y, y*z + 7*y + 6, 2*y + 6, z^2 + 3, 2*z + 10]
```

Modulo any other prime the Groebner basis is trivial so there are no other solutions. For example:

```python
sage: I.change_ring( P.change_ring( GF(3) ) ).groebner_basis()
[1]
```

**Note:** Sage distinguishes between lists or sequences of polynomials and ideals. Thus an ideal is not identified
with a particular set of generators. For sequences of multivariate polynomials see \texttt{sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic}.

AUTHORS:

- William Stein: initial version
- Kiran S. Kedlaya (2006-02-12): added Macaulay2 analogues of some Singular features
- Martin Albrecht (2009): added Groebner basis over rings functionality from Singular 3.1
- John Perry (2012): bug fixing equality & containment of ideals

\texttt{class sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal}(\texttt{ring}, \texttt{gens}, \texttt{coerce=True})

Bases: \texttt{sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr}, \texttt{sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_macaulay2_repr}, \texttt{sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_magma_repr}, \texttt{sage.ideal.Ideal_generic}

Create an ideal in a multivariate polynomial ring.

INPUT:

- \texttt{ring} - the ring the ideal is defined in
- \texttt{gens} - a list of generators for the ideal
- \texttt{coerce} - coerce elements to the ring

EXAMPLES:

\begin{verbatim}
sage: R.<x,y> = PolynomialRing(IntegerRing(), 2, order='lex')
sage: R.ideal([x, y])
Ideal (x, y) of Multivariate Polynomial Ring in x, y over Integer Ring
sage: R.<x0,x1> = GF(3)[]
sage: R.ideal([x0^2, x1^3])
Ideal (x0^2, x1^3) of Multivariate Polynomial Ring in x0, x1 over Finite Field of size 3
\end{verbatim}

\texttt{basis}

Shortcut to \texttt{gens()}. 

EXAMPLES:

\begin{verbatim}
sage: P.<x,y> = PolynomialRing(QQ,2)
sage: I = Ideal([x,y+1])
sage: I.basis
[x, y + 1]
\end{verbatim}

\texttt{change_ring}(\texttt{P})

Return the ideal \texttt{I} in \texttt{P} spanned by the generators \(g_1, \ldots, g_n\) of \texttt{self} as returned by \texttt{self.gens()}. 

INPUT:

- \texttt{P} - a multivariate polynomial ring 

EXAMPLES:
degree_of_semi_regularity()

Return the degree of semi-regularity of this ideal under the assumption that it is semi-regular.

Let \( \{f_1, \ldots, f_m\} \subset K[x_1, \ldots, x_n] \) be homogeneous polynomials of degrees \( d_1, \ldots, d_m \) respectively. This sequence is semi-regular if:

- \( \{f_1, \ldots, f_m\} \neq K[x_1, \ldots, x_n] \)
- for all \( 1 \leq i \leq m \) and \( g \in K[x_1, \ldots, x_n] \): \( \deg(g \cdot p_i) < D \) and \( g \cdot f_i < f_1, \ldots, f_{i-1} \) implies that \( g \in < f_1, \ldots, f_{i-1} > \) where \( D \) is the degree of regularity.

This notion can be extended to affine polynomials by considering their homogeneous components of highest degree.

The degree of regularity of a semi-regular sequence \( f_1, \ldots, f_m \) of respective degrees \( d_1, \ldots, d_m \) is given by the index of the first non-positive coefficient of:

\[
\sum c_k z^k = \prod \frac{1}{(1-z^{d_i})^{n_i}}
\]

EXAMPIES:

We consider a homogeneous example:
From this, we expect a Groebner basis computation to reach at most degree 4. For homogeneous systems this is equivalent to the largest degree in the Groebner basis:

```python
sage: max(f.degree() for f in I.groebner_basis())
4
```

We increase the number of polynomials and observe a decrease the degree of regularity:

```python
sage: for i in range(2*n):
    ....: f = P.random_element(degree=2, terms=binomial(n,2))
    ....: f -= f(*s)
    ....: L.append(f.homogenize())
sage: I = Ideal(L)
sage: I.degree_of_semi_regularity()
3
sage: max(f.degree() for f in I.groebner_basis())
3
```

The degree of regularity approaches 2 for quadratic systems as the number of polynomials approaches $n^2$:

```python
sage: for i in range((n-4)*n):
    ....: f = P.random_element(degree=2, terms=binomial(n,2))
    ....: f -= f(*s)
    ....: L.append(f.homogenize())
sage: I = Ideal(L)
sage: I.degree_of_semi_regularity()
2
sage: max(f.degree() for f in I.groebner_basis())
2
```

**Note:** It is unknown whether semi-regular sequences exist. However, it is expected that random systems are semi-regular sequences. For more details about semi-regular sequences see [BFS2004].

gens()

Return a set of generators / a basis of this ideal. This is usually the set of generators provided during object creation.

**EXAMPLES:**

```python
sage: P.<x,y> = PolynomialRing(QQ,2)
sage: I = Ideal([x,y+1]); I
Ideal (x, y + 1) of Multivariate Polynomial Ring in x, y over Rational Field
sage: I.gens()
[x, y + 1]
```

groebner_basis(algorithm=",", deg_bound=None, mult_bound=None, prot=False, *args, **kwds)

Return the reduced Groebner basis of this ideal.

A Groebner basis $g_1,...,g_n$ for an ideal $I$ is a generating set such that $<LM(g_i)> = LM(I)$, i.e., the leading monomial ideal of $I$ is spanned by the leading terms of $g_1,...,g_n$. Groebner bases are the key concept in computational ideal theory in multivariate polynomial rings which allows a variety of problems to be solved.
Additionally, a reduced Groebner basis $G$ is a unique representation for the ideal $< G >$ with respect to the chosen monomial ordering.

INPUT:

- **algorithm** - determines the algorithm to use, see below for available algorithms.
- **deg_bound** - only compute to degree $deg\_bound$, that is, ignore all S-polynomials of higher degree. (default: None)
- **mult_bound** - the computation is stopped if the ideal is zero-dimensional in a ring with local ordering and its multiplicity is lower than $mult\_bound$. Singular only. (default: None)
- **prot** - if set to True the computation protocol of the underlying implementation is printed. If an algorithm from the singular: or magma: family is used, prot may also be sage in which case the output is parsed and printed in a common format where the amount of information printed can be controlled via calls to set_verbose().
- ***args** - additional parameters passed to the respective implementations
- ****kwds** - additional keyword parameters passed to the respective implementations

ALGORITHMS:

- ‘autoselect’ (default)
- ‘singular:groebner’ Singular’s groebner command
- ‘singular:std’ Singular’s std command
- ‘singular:stdhilb’ Singular’s stdhilb command
- ‘singular:stdfglm’ Singular’s stdfglm command
- ‘singular:slimgb’ Singular’s slimgb command
- ‘libsingular:groebner’ libSingular’s groebner command
- ‘libsingular:std’ libSingular’s std command
- ‘libsingular:slimgb’ libSingular’s slimgb command
- ‘libsingular:stdhilb’ libSingular’s stdhilb command
- ‘libsingular:stdfglm’ libSingular’s stdfglm command
- ‘toy:buchberger’ Sage’s toy/educational buchberger without Buchberger criteria
- ‘toy:buchberger2’ Sage’s toy/educational buchberger with Buchberger criteria
- ‘toy:d_basis’ Sage’s toy/educational algorithm for computation over PIDs
- ‘macaulay2:gb’ Macaulay2’s gb command (if available)
- ‘macaulay2:f4’ Macaulay2’s GroebnerBasis command with the strategy “F4” (if available)
- ‘macaulay2:mgb’ Macaulay2’s GroebnerBasis command with the strategy “MGB” (if available)
- ‘magma:GroebnerBasis’ Magma’s GroebnerBasis command (if available)
- ‘ginv:TQ’, ‘ginv:TQBlockHigh’, ‘ginv:TQBlockLow’ and ‘ginv:TQDegree’ One of GINV’s implementations (if available)
- ‘giac:gbasis’ Giac’s gbasis command (if available)

If only a system is given - e.g. ‘magma’ - the default algorithm is chosen for that system.
**Note:** The Singular and libSingular versions of the respective algorithms are identical, but the former calls an external Singular process while the latter calls a C function, i.e. the calling overhead is smaller. However, the libSingular interface does not support pretty printing of computation protocols.

**EXAMPLES:**

Consider Katsura-3 over \( \mathbb{Q} \) with lexicographical term ordering. We compute the reduced Groebner basis using every available implementation and check their equality.

```python
sage: P.<a,b,c> = PolynomialRing(QQ,3, order='lex')
sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis()
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c^2 + 1/84*c]
```

```python
sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis('libsingular:groebner')
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c^2 + 1/84*c]
```

```python
sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis('libsingular:std')
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c^2 + 1/84*c]
```

```python
sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis('libsingular:stdhilb')
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c^2 + 1/84*c]
```

```python
sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis('libsingular:stdfglm')
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c^2 + 1/84*c]
```

```python
sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis('libsingular:slimgb')
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c^2 + 1/84*c]
```

Although Giac does support lexicographical ordering, we use degree reverse lexicographical ordering here, in order to test against trac ticket #21884:

```python
sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.change_ring(P.change_ring(order='degrevlex'))
sage: gb = I.groebner_basis('giac') # random
sage: gb
[c^3 - 79/210*b^2 + 1/30*b + 1/70*c, b^2 - 3/5*c^2 - 1/5*b + 1/5*c, b*c + 6/5*c^2 - 2 - 1/10*b - 2/5*c, a + 2*b + 2*c - 1]
```

```python
sage: J.groebner_basis.set_cache(gb)
```
sage: ideal(J.transformed_basis()).change_ring(P).interreduced_basis()  # testing trac 21884
...
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/
21*c^3 + 1/84*c^2 + 1/84*c]

Giac's gbasis over \( \mathbb{Q} \) can benefit from a probabilistic lifting and multi threaded operations:

sage: A9=PolynomialRing(QQ,9,'x')
sage: I9=sage.rings.ideal.Katsura(A9)
sage: print("possible output from giac", flush=True); I9.groebner_basis("giac",
    proba_epsilon=1e-7)  # long time (3s)
possible output...
Polynomial Sequence with 143 Polynomials in 9 Variables

The list of available Giac options is provided at \( \text{sage.libs.giac.groebner_basis}() \).

Note that \texttt{toy:buchberger} does not return the reduced Groebner basis,

sage: I = sage.rings.ideal.Katsura(P,3)  # regenerate to prevent caching
sage: gb = I.groebner_basis('toy:buchberger')
True
sage: gb == gb.reduced()
False

but that \texttt{toy:buchberger2} does.

sage: I = sage.rings.ideal.Katsura(P,3)  # regenerate to prevent caching
sage: gb = I.groebner_basis('toy:buchberger2'); gb
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/
21*c^3 + 1/84*c^2 + 1/84*c]
sage: gb == gb.reduced()
True

Here we use Macaulay2 with three different strategies over a finite field.

sage: R.<a,b,c> = PolynomialRing(GF(101), 3)
sage: I = sage.rings.ideal.Katsura(R,3)  # regenerate to prevent caching
sage: I.groebner_basis('macaulay2:gb')  # optional - macaulay2
[c^3 + 28*c^2 - 37*b + 13*c, b^2 - 41*c^2 + 20*b - 20*c, b*c - 19*c^2 + 10*b +
40*c, a + 2*b + 2*c - 1]
sage: I = sage.rings.ideal.Katsura(R,3)  # regenerate to prevent caching
sage: I.groebner_basis('macaulay2:f4')  # optional - macaulay2
[c^3 + 28*c^2 - 37*b + 13*c, b^2 - 41*c^2 + 20*b - 20*c, b*c - 19*c^2 + 10*b +
40*c, a + 2*b + 2*c - 1]
sage: I = sage.rings.ideal.Katsura(R,3)  # regenerate to prevent caching
sage: I.groebner_basis('macaulay2:mgb')  # optional - macaulay2
[c^3 + 28*c^2 - 37*b + 13*c, b^2 - 41*c^2 + 20*b - 20*c, b*c - 19*c^2 + 10*b +
40*c, a + 2*b + 2*c - 1]
Sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis('magma:GroebnerBasis') # optional - magma
[\begin{align*}
   a - 60c^3 &+ 158/7c^2 - 8/7c - 1, \\
   b + 30c^3 &- 79/7c^2 + 3/7c, \\
   c^4 &- 21c^3 + 1/84c^2 + 1/84c
\end{align*}]

Singular and libSingular can compute Groebner basis with degree restrictions.

Sage: R.<x,y> = QQ[]
Sage: I = R*[x^3+y^2,x^2*y+1]
Sage: I.groebner_basis(algorithm='singular')
[x^3 + y^2, x^2*y + 1]
Sage: I.groebner_basis(algorithm='singular',deg_bound=2)
[x^3 + y^2, x^2*y + 1]
Sage: I.groebner_basis()
[x^3 + y^2, x^2*y + 1, y^3 - x]
Sage: I.groebner_basis(deg_bound=2)
[x^3 + y^2, x^2*y + 1]

A protocol is printed, if the verbosity level is at least 2, or if the argument prot is provided. Historically, the protocol did not appear during doctests, so, we skip the examples with protocol output.

Sage: from sage.misc.verbose import set_verbose
Sage: set_verbose(2)
Sage: I = R*[x^3+y^2,x^2*y+1]
Sage: I.groebner_basis() # not tested
std in (QQ),(x,y),(dp(2),C)
[...:2]3ss4s6
(S:2)--
product criterion:1 chain criterion:0
[x^3 + y^2, x^2*y + 1, y^3 - x]
Sage: I.groebner_basis(prot=False)
std in (QQ),(x,y),(dp(2),C)
[...:2]3ss4s6
(S:2)--
product criterion:1 chain criterion:0
[x^3 + y^2, x^2*y + 1, y^3 - x]
Sage: set_verbose(0)
Sage: I.groebner_basis(prot=True) # not tested
std in (QQ),(x,y),(dp(2),C)
[...:2]3ss4s6
(S:2)--
product criterion:1 chain criterion:0
[x^3 + y^2, x^2*y + 1, y^3 - x]

The list of available options is provided at LibSingularOptions.

Note that Groebner bases over \( \mathbb{Z} \) can also be computed.

Sage: P.<a,b,c> = PolynomialRing(ZZ,3)
Sage: I = P *[a + 2*b + 2*c - 1, a^2 - a + 2*b^2 + 2*c^2, 2*a*b + 2*b*c - b]
Sage: I.groebner_basis()
[b^3 + b*c^2 + 12*c^3 + b^2 + b*c - 4*c^2, \\
2*b*c^2 - 6*c^3 - b^2 - b*c + 2*c^2, \\
42*c^3 + b^2 + 2*b*c - 14*c^2 + b, 
(continues on next page)
\[2*b^2 + 6*b*c + 6*c^2 - b - 2*c,\]
\[10*b*c + 12*c^2 - b - 4*c,\]
\[a + 2*b + 2*c - 1\]

**Sage**: `I.groebner_basis('macaulay2') # optional - macaulay2`

\[
[b^3 + b*c^2 + 12*c^3 + b^2 + b*c - 4*c^2,\]
\[2*b*c^2 - 6*c^3 + b^2 + 5*b*c + 8*c^2 - b - 2*c,\]
\[42*c^3 + b^2 + 2*b*c - 14*c^2 + b,\]
\[2*b^2 - 4*b*c - 6*c^2 + 2*c, 10*b*c + 12*c^2 - b - 4*c,\]
\[a + 2*b + 2*c - 1\]

Groebner bases over \(\mathbb{Z}/n\mathbb{Z}\) are also supported:

**Sage**: `P.<a,b,c> = PolynomialRing(Zmod(1000),3)`

```
sage: I = P * (a + 2*b + 2*c - 1, a^2 - a + 2*b^2 + 2*c^2, 2*a*b + 2*b*c - b)
sage: I.groebner_basis()
[b^2 + 732*b*c + 808*b, a + 2*b + 2*c - 1, b^2 + 438*b*c + 281*b, 5*b*c + 156*c^2 + 112*b + 948*c, 50*c^2 + 600*b + 650*c, a + 2*b + 2*c + 999, 125*b]
```

Sage also supports local orderings:

**Sage**: `P.<x,y,z> = PolynomialRing(QQ,3,order='negdegrevlex')`

```
sage: I = P * ( x*y*z + z^5, 2*x^2 + y^3 + z^7, 3*z^5 + y^5 )
sage: I.groebner_basis()
[x^2 + 1/2*y^3, x*y*z + z^5, y^5 + 3*z^5, y^4*z - 2*x*z^5, z^6]
```

We can represent every element in the ideal as a combination of the generators using the `lift()` method:

**Sage**: `P.<x,y,z> = PolynomialRing(QQ,3)`

```
sage: I = P * ( x*y*z + z^5, 2*x^2 + y^3 + z^7, 3*z^5 + y^5 )
sage: J = Ideal(I.groebner_basis())
sage: f = sum(P.random_element(terms=2)*f for f in I.gens())
sage: f
# random
1/2*y^2*z^7 - 1/4*y*z^8 + 2*x*z^5 + 95*z^6 + 1/2*y^5 - 1/4*y^4*z + x^2*y^2 + 3/2*x^2*y*z + 95*x*y*z^2
sage: f.lift(I.gens()); l
# random
[2*x + 95*z, 1/2*y^2 - 1/4*y*z, 0]
sage: sum(map(mul, zip(l,J.gens()))) == f
```

(continues on next page)
Groebner bases over fraction fields of polynomial rings are also supported:

```
sage: P.<t> = QQ[]
sage: F = Frac(P)
sage: R.<X,Y,Z> = F[
 sage: I = Ideal([f + P.random_element() for f in sage.rings.ideal.Katsura(R).gens()])
sage: I.groebner_basis()
```

```
[Z^3 + (-79/105*t - 79/70)*Z^2 + (2/15*t^2 - 74/315*t + 94/105)*Y + (2/35*t^2 + 194/315*t + 1/105)*Z - 4/15*t + 1/2, Y^2 + (-3/5)*Z^2 + (-2/5*t - 3/5)*Y + (2/5*t + 3/5)*Z - 4/15*t + 1/2, Y*Z + 6/5*Z^2 + (-1/5*t - 3/10)*Y + (-4/5*t - 6/5)*Z + 8/15*t - 1/2, X + 2*Y + 2*Z - t - 2]
```

In cases where a characteristic cannot be determined, we use a toy implementation of Buchberger’s algorithm (see trac ticket #6581):

```
sage: R.<a,b> = QQ[]; I = R.ideal(a^2+b^2-1)
sage: Q = QuotientRing(R,I); K = Frac(Q)
sage: R2.<x,y> = K[]; J = R2.ideal([(a^2+b^2)*x + y, x+y])
sage: J.groebner_basis()
```

```
verbose 0 (...: multi_polynomial_ideal.py, groebner_basis) Warning: falling back to very slow toy implementation.
[x + y]
```

ALGORITHM:

Uses Singular, Magma (if available), Macaulay2 (if available), Giac (if available), or a toy implementation.

```
groebner_fan(is_groebner_basis=False, symmetry=None, verbose=False)
```

Return the Groebner fan of this ideal.

The base ring must be \( \mathbb{Q} \) or a finite field \( \mathbb{F}_p \) of with \( p \leq 32749 \).

EXAMPLES:

```
sage: P.<x,y> = PolynomialRing(QQ)
sage: i = ideal(x^2 - y^2 + 1)
sage: g = i.groebner_fan()
sage: g.reduced_groebner_bases()
```

```
[[x^2 - y^2 + 1], [-x^2 + y^2 - 1]]
```

INPUT:

- **is_groebner_basis** - bool (default False). if True, then \( \text{I.gens()} \) must be a Groebner basis with respect to the standard degree lexicographic term order.

- **symmetry** - default: None; if not None, describes symmetries of the ideal

- **verbose** - default: False; if True, printout useful info during computations

```
homogenize(var='h')
```

Return homogeneous ideal spanned by the homogeneous polynomials generated by homogenizing the generators of this ideal.

INPUT:
• h - variable name or variable in cover ring (default: ‘h’)

EXAMPLES:

```python
sage: P.<x,y,z> = PolynomialRing(GF(2))
sage: I = Ideal([x^2*y + z + 1, x + y^2 + 1]); I
Ideal (x^2*y + z + 1, y^2 + x + 1) of Multivariate Polynomial Ring in x, y, z over Finite Field of size 2
```

```python
sage: I.homogenize()
Ideal (x^2*y + z*h^2 + h^3, y^2 + x*h + h^2) of Multivariate Polynomial Ring in x, y, z, h over Finite Field of size 2
```

```python
sage: I.homogenize(y)
Ideal (x^2*y + y^3 + y^2*z, x*y) of Multivariate Polynomial Ring in x, y, z over Finite Field of size 2
```

```python
sage: I = Ideal([x^2*y + z^3 + y^2*x, x + y^2 + 1])
sage: I.is_homogeneous()
False
```

```python
sage: J = I.homogenize()
sage: J
Ideal (x + 2*y + 2*z - h, x^2 + 2*y^2 + 2*z^2 - x, 2*x*y + 2*y*z - y) of Multivariate Polynomial Ring in x, y, z, h over Rational Field
```

```python
sage: J.is_homogeneous()
True
```

```python
plot(*args, **kwds)
Plot the real zero locus of this principal ideal.

INPUT:

• self - a principal ideal in 2 variables
```
• **algorithm** - set this to ‘surf’ if you want ‘surf’ to plot the ideal (default: None)
• **args** - optional tuples (variable, minimum, maximum) for plotting dimensions
• **kwds** - optional keyword arguments passed on to **implicit_plot**

EXAMPLES:

Implicit plotting in 2-d:

```python
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: I = R.ideal([y^3 - x^2])
sage: I.plot()  # cusp
Graphics object consisting of 1 graphics primitive
```

```python
sage: I = R.ideal([y^2 - x^2 - 1])
sage: I.plot((x,-3, 3), (y, -2, 2))  # hyperbola
Graphics object consisting of 1 graphics primitive
```

```python
sage: I = R.ideal([y^2 + x^2*(1/4) - 1])
sage: I.plot()  # ellipse
Graphics object consisting of 1 graphics primitive
```

```python
sage: I = R.ideal([y^2-(x^2-1)*(x-2)])
sage: I.plot()  # elliptic curve
Graphics object consisting of 1 graphics primitive
```

```python
sage: f = ((x+3)^3 + 2*(x+3)^2 - y^2)*(x^3 - y^2)*((x-3)^3-2*(x-3)^2-y^2)
sage: I = R.ideal(f)
sage: I.plot()  # the Singular logo
Graphics object consisting of 1 graphics primitive
```

```python
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: I = R.ideal([x - 1])
sage: I.plot((y, -2, 2))  # vertical line
Graphics object consisting of 1 graphics primitive
```

```python
sage: I = R.ideal([-x^2*y + 1])
sage: I.plot()  # blow up
Graphics object consisting of 1 graphics primitive
```

**random_element**(degree, compute_gb=False, *args, **kwds)

Return a random element in this ideal as \( r = \sum h_i \cdot f_i \).

INPUT:

• **compute_gb** - if True then a Gröbner basis is computed first and \( f_i \) are the elements in the Gröbner basis. Otherwise whatever basis is returned by self.gens() is used.
• **args** and **kwds** are passed to R.random_element() with \( R = \text{self.ring()} \).

EXAMPLES:

We compute a uniformly random element up to the provided degree.
```
sage: P.<x,y,z> = GF(127)[]
sage: I = sage.rings.ideal.Katsura(P)
sage: I.random_element(degree=4, compute_gb=True, terms=infinity)
34*x^4 - 33*x^3*y + 45*x^2*y^2 - 51*x*y^3 - 55*y^4 + 43*x^3*z ... - 28*y - 33*z
+ 45
```

Note that sampling uniformly at random from the ideal at some large enough degree is equivalent to computing a Gröbner basis. We give an example showing how to compute a Gröbner basis if we can sample uniformly at random from an ideal:

```
sage: n = 3; d = 4
sage: P = PolynomialRing(GF(127), n, 'x')
sage: I = sage.rings.ideal.Cyclic(P)

1. We sample $n^d$ uniformly random elements in the ideal:
   ```
sage: F = Sequence(I.random_element(degree=d, compute_gb=True, terms=infinity) for _ in range(n^d))
   ```

2. We linearize and compute the echelon form:
   ```
sage: A,v = F.coefficient_matrix()
sage: A.echelonize()
   ```

3. The result is the desired Gröbner basis:
   ```
sage: G = Sequence((A*v).list())
sage: G.is_groebner()
True
sage: Ideal(G) == I
True
   ```

We return some element in the ideal with no guarantee on the distribution:

```
sage: P = PolynomialRing(GF(127), 10, 'x')
sage: I = sage.rings.ideal.Katsura(P)
sage: f = I.random_element(degree=3)
sage: f
# random
-25*x0^2*x1 + 14*x1^3 + 57*x0*x1*x2 + ... + 19*x7*x9 + 40*x8*x9 + 49*x1
sage: f.degree()
3
```

We show that the default method does not sample uniformly at random from the ideal:

```
sage: P.<x,y,z> = GF(127)[]
sage: G = Sequence([x+7, y-2, z+110])
sage: I = Ideal([sum(P.random_element() * g for g in G) for _ in range(4)])
sage: all(I.random_element(degree=1) == 0 for _ in range(100))
True
```

If degree equals the degree of the generators a random linear combination of the generators is returned:
reduce(f)
Reduce an element modulo the reduced Groebner basis for this ideal. This returns 0 if and only if the element is in this ideal. In any case, this reduction is unique up to monomial orders.

EXAMPLES:

```sage
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: I = (x^3 + y, y)*R
sage: I.reduce(y)
0
sage: I.reduce(x^3)
0
sage: I.reduce(x - y)
x
sage: I = (y^2 - (x^3 + x))*R
sage: I.reduce(x^3)
y^2 - x
sage: I.reduce(x^6)
y^4 + 2*x*y^2 + x^2
sage: (y^2 - x)^2
y^4 - 2*x*y^2 + x^2
```

Note: Requires computation of a Groebner basis, which can be a very expensive operation.

subs(in_dict=None, **kwds)
Substitute variables.
This method substitutes some variables in the polynomials that generate the ideal with given values. Variables that are not specified in the input remain unchanged.

INPUT:

- `in_dict` – (optional) dictionary of inputs
- `**kwds` – named parameters

OUTPUT:
A new ideal with modified generators. If possible, in the same polynomial ring. Raises a TypeError if no common polynomial ring of the substituted generators can be found.

EXAMPLES:

```sage
sage: R.<x,y> = PolynomialRing(ZZ,2, 'xy')
sage: I = R.ideal(x^5+y^5, x^2 + y + x^2*y^2 + 5); I
Ideal (x^5 + y^5, x^2*y^2 + x^2 + y + 5) of Multivariate Polynomial Ring in x, y over Integer Ring
sage: I.subs(x=y)
```

(continues on next page)
Ideal \((2*y^5, y^4 + y^2 + y + 5)\) of Multivariate Polynomial Ring in \(x, y\) over \(\text{Integer Ring}\)

\[
\text{sage: I.subs({x:y})} \quad \# \text{ same substitution but with dictionary}
\]
Ideal \((2*y^5, y^4 + y^2 + y + 5)\) of Multivariate Polynomial Ring in \(x, y\) over \(\text{Integer Ring}\)

The new ideal can be in a different ring:

\[
\text{sage: R.<a,b> = PolynomialRing(QQ,2)}
\]
\[
\text{sage: S.<x,y> = PolynomialRing(QQ,2)}
\]
\[
\text{sage: I = R.ideal(a^2+b^2-a-b+2); I}
\]
Ideal \((a^2 + b^2 + a - b + 2)\) of Multivariate Polynomial Ring in \(a, b\) over \(\text{Rational Field}\)

\[
\text{sage: I.subs(a=x, b=y)}
\]
Ideal \((x^2 + y^2 + x - y + 2)\) of Multivariate Polynomial Ring in \(x, y\) over \(\text{Rational Field}\)

The resulting ring need not be a multivariate polynomial ring:

\[
\text{sage: T.<t> = PolynomialRing(QQ)}
\]
\[
\text{sage: I.subs(a=t, b=t)}
\]
Principal ideal \((t^2 + 1)\) of Univariate Polynomial Ring in \(t\) over Rational Field

\[
\text{sage: var("z")}
\]
z
\[
\text{sage: I.subs(a=z, b=z)}
\]
Principal ideal \((2*z^2 + 2)\) of Symbolic Ring

Variables that are not substituted remain unchanged:

\[
\text{sage: R.<x,y> = PolynomialRing(QQ,2)}
\]
\[
\text{sage: I = R.ideal(x^2+y^2+x-y+2); I}
\]
Ideal \((x^2 + y^2 + x - y + 2)\) of Multivariate Polynomial Ring in \(x, y\) over \(\text{Rational Field}\)

\[
\text{sage: I.subs(x=1)}
\]
Ideal \((y^2 - y + 4)\) of Multivariate Polynomial Ring in \(x, y\) over Rational Field

\text{weil_restriction}()

Computes the Weil restriction of this ideal over some extension field. If the field is a finite field, then this computes the Weil restriction to the prime subfield.

A Weil restriction of scalars - denoted \(\text{Res}_{L/k}\) - is a functor which, for any finite extension of fields \(L/k\) and any algebraic variety \(X\) over \(L\), produces another corresponding variety \(\text{Res}_{L/k}(X)\), defined over \(k\). It is useful for reducing questions about varieties over large fields to questions about more complicated varieties over smaller fields.

This function does not compute this Weil restriction directly but computes on generating sets of polynomial ideals:

Let \(d\) be the degree of the field extension \(L/k\), let \(a\) a generator of \(L/k\) and \(p\) the minimal polynomial of \(L/k\). Denote this ideal by \(I\).

Specifically, this function first maps each variable \(x\) to its representation over \(k\): \(\sum_{i=0}^{d-1} a^i x_i\). Then each generator of \(I\) is evaluated over these representations and reduced modulo the minimal polynomial \(p\). The result is interpreted as a univariate polynomial in \(a\) and its coefficients are the new generators of the returned ideal.

3.1. Multivariate Polynomials and Polynomial Rings
If the input and the output ideals are radical, this is equivalent to the statement about algebraic varieties above.

**OUTPUT:** MPolynomial Ideal

**EXAMPLES:**

```sage
ek.<a> = GF(2^2)
P.<x,y> = PolynomialRing(k,2)
I = Ideal([x*y + 1, a*x + 1])
I.variety()
[(y: a, x: a + 1)]
J = I.weil_restriction()
J
Ideal (x0*y0 + x1*y1 + 1, x1*y0 + x0*y1 + x1*y1, x1 + 1, x0 + x1) of
Multivariate Polynomial Ring in x0, x1, y0, y1 over Finite Field of size
2
J += sage.rings.ideal.FieldIdeal(J.ring())  # ensure radical ideal
J.variety()  # py2
[(y1: 1, x1: 1, x0: 1, y0: 0)]
J.variety()  # py3
[(y1: 1, y0: 0, x1: 1, x0: 1)]
J.weil_restriction()  # returns J
Ideal (x0*y0 + x1*y1 + 1, x1*y0 + x0*y1 + x1*y1, x1 + 1, x0 + x1, x0^2 +
x0, x1^2 + 1, y0^2 + y0, y1^2 + y1) of Multivariate Polynomial Ring in
x0, x1, y0, y1 over Finite Field of size 2
k.<a> = GF(3^5)
P.<x,y,z> = PolynomialRing(k)
I = sage.rings.ideal.Katsura(P)
I.dimension()
0
I.variety()  # py2
[(y: 0, z: 0, x: 1)]
I.variety()  # py3
[(z: 0, y: 0, x: 1)]
J = I.weil_restriction(); J
Ideal (x0 - y0 - z0 - 1, x1 - y1 - z1, x2 - y2 - z2, x3 - y3 - z3, x4 -
y4 - z4, x0^2 + x2*x3 + x1*x4 - y0^2 - y2*y3 - y1*y4 - z0^2 - z2*z3 -
z1*z4 - x0, -x0*x1 - x2*x3 - x3^2 - x1*x4 + x2*x4 + y0*y1 + y2*y3 + y3^2 +
y1*y4 - y2*x4 + z0*z1 + z2*z3 + z3^2 + z1*z4 - z2*z4 - x1, x1^2 -
x0*x2 + x3^2 - x2*x4 + x3*x4 - y1^2 + y0*y2 - y3^2 + y2*y4 - y3*y4 -
z1^2 + z0*z2 - z3^2 + z2*z4 - z3*z4 - x2, -x1*x2 - x0*x3 - x3*x4 - x4^2 +
y1*y2 + y0*y3 + y3*y4 + y4^2 + z1*z2 + z0*z3 + z3*z4 + z4^2 - x3, x2^2 -
x1*x3 - x0*x4 + x4^2 - y2^2 + y1*y3 + y0*y4 - y4^2 - z2*z2 + z1*z3 +
z0*z4 - z4^2 - x4, -x0*y0 + x4*y1 + x3*y2 + x2*y3 + x1*y4 - y0*z0 +
y4*z1 + y3*z2 + y2*z3 + y1*z4 - y0, -x1*y0 - x0*y1 - x4*y1 - x3*y2 +
x4*y2 - x2*y3 + x3*y3 - x1*y4 + x2*y4 - y1*z0 - y0*z1 - y4*z1 - y3*z2 +
y4*z2 - y2*z3 + y3*z3 - y1*z4 + y2*z4 - y1, -x2*y0 - x1*y1 - x0*y2 -
x4*y2 - x3*y3 + x4*y3 - x2*y4 + x3*y4 - y2*z0 - y1*z1 - y0*z2 - y4*z2 -
y3*z3 + y4*z3 - y2*z4 + y3*z4 - y2, -x3*y0 - x2*y1 - x1*y2 - x0*y3 -
x4*y3 - x3*y4 + x4*y4 - y3*z0 - y2*z1 - y1*z2 - y0*z3 - y4*z3 - y3*z4 +
```

(continues on next page)
Weil restrictions are often used to study elliptic curves over extension fields so we give a simple example involving those:

```python
sage: K.<a> = QuadraticField(1/3)
sage: E = EllipticCurve(K,[1,2,3,4,5])
```

We pick a point on $E$:

```python
sage: p = E.lift_x(1); p
(1 : 2 : 1)
sage: I = E.defining_ideal(); I
Ideal (-x^3 - 2*x^2*z + x*y*z + y^2*z - 4*x*z^2 + 3*y*z^2 - 5*z^3) of Multivariate Polynomial Ring in x, y, z over Number Field in a with defining polynomial x^2 - 1/3 with a = 0.5773502691896258?
```

Of course, the point $p$ is a root of all generators of $I$:

```python
sage: I.subs(x=1,y=2,z=1)
Ideal (0) of Multivariate Polynomial Ring in x, y, z over Number Field in a with defining polynomial x^2 - 1/3 with a = 0.5773502691896258?
```

$I$ is also radical:

```python
sage: I.radical() == I
True
```

So we compute its Weil restriction:

```python
sage: J = I.weil_restriction()
sage: J
Ideal (-x0^3 - x0*x1^2 - 2*x0^2*z0 - 2/3*x1^2*z0 + x0*y0*z0 + y0^2*z0 + 1/3*x1*y1*z0 + 1/3*y1^2*z0 - 4*x0^2*z0^2 + 3*y0*z0^2 - 5*z0^3 - 4/3*x0*x1*z1 + 1/3*x1*y0*z1 + 1/3*x0*y1*z1 + 2/3*y0*y1*z1 - 8/3*x1*z0^2 + 2*y1*z0^2 - 5*z0^2*z1 - 1/3*x1^3 - 4*x0*x1^2*z0 + x1*y0*z0 + x0*y1*z0 + 2*y0*y1*z0 - 4*x1*z0^2 + 3*y1*z0^2 - 2*x0^2*z1 - 2/3*x1^2*z1 + x0*y0*z1 + y0^2*z1 + 1/3*x1*y1*z1 + 1/3*y1^2*z1 - 8*x0^2*z0*z1 + 6*y0*z0^2*z1 - 15*z0^2*z1^2 - 4/3*x1*z1^2 + y1*z1^2 - 5/3*z1^3) of Multivariate Polynomial Ring in x0, x1, y0, y1, z0, z1 over Rational Field
```

We can check that the point $p$ is still a root of all generators of $J$:  

```python
3.1. Multivariate Polynomials and Polynomial Rings 343
```
Example for relative number fields:

```
sage: R.<x> = QQ[]
sage: K.<w> = NumberField(x^5-2)
sage: R.<x> = K[]
sage: L.<v> = K.extension(x^2+1)
sage: S.<x,y> = L[]
sage: I = S.ideal([y^2-x^3-1])
sage: I.weil_restriction()
Ideal (-x0^3 + 3*x0*x1^2 + y0^2 - y1^2 - 1, -3*x0^2*x1 + x1^3 + 2*y0*y1)
of Multivariate Polynomial Ring in x0, x1, y0, y1 over Number Field in w
with defining polynomial x^5 - 2
```

Note: Based on a Singular implementation by Michael Brickenstein

---

**class** sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_macaulay2_repr

Bases: object

An ideal in a multivariate polynomial ring, which has an underlying Macaulay2 ring associated to it.

**EXAMPLES:**

```
sage: R.<x,y,z,w> = PolynomialRing(ZZ, 4)
sage: I = ideal(x*y-z^2, y^2-w^2)
sage: I
Ideal (x*y - z^2, y^2 - w^2) of Multivariate Polynomial Ring in x, y, z, w over Integer Ring
```

**class** sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_magma_repr

Bases: object

**class** sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_base_repr

Bases: object

**syzygy_module()**

Computes the first syzygy (i.e., the module of relations of the given generators) of the ideal.

**EXAMPLES:**

```
sage: R.<x,y> = PolynomialRing(QQ)
sage: f = 2*x^2 + y
sage: g = y
sage: h = 2*f + g
sage: I = Ideal([f,g,h])
sage: M = I.syzygy_module(); M
[ -2 -1 1]
[ -y 2*x^2 + y 0]
sage: G = vector(I.gens())
sage: M*G
(0, 0)
```
ALGORITHM: Uses Singular’s syz command

```python
class sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr
    Bases: sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_base_repr

An ideal in a multivariate polynomial ring, which has an underlying Singular ring associated to it.

associated_primes(algorithm='sy')
Return a list of the associated primes of primary ideals of which the intersection is \( I = \text{self} \).

An ideal \( Q \) is called primary if it is a proper ideal of the ring \( R \) and if whenever \( ab \in Q \) and \( a \notin Q \) then \( b^n \in Q \) for some \( n \in \mathbb{Z} \).

If \( Q \) is a primary ideal of the ring \( R \), then the radical ideal \( P \) of \( Q \), i.e. \( P = \{ a \in R, a^n \in Q \} \) for some \( n \in \mathbb{Z} \), is called the associated prime of \( Q \).

If \( I \) is a proper ideal of the ring \( R \) then there exists a decomposition in primary ideals \( Q_i \) such that

* their intersection is \( I \)
* none of the \( Q_i \) contains the intersection of the rest, and
* the associated prime ideals of \( Q_i \) are pairwise different.

This method returns the associated primes of the \( Q_i \).

INPUT:

* algorithm - string:
  * 'sy' - (default) use the Shimoyama-Yokoyama algorithm
  * 'gtz' - use the Gianni-Trager-Zacharias algorithm

OUTPUT:

* list - a list of associated primes

EXAMPLES:
```
sage: R.<x,y,z> = PolynomialRing(QQ, 3, order='lex')
sage: p = z^2 + 1; q = z^3 + 2
sage: I = (p*q^2, y-z^2)*R
sage: pd = I.associated_primes(); sorted(pd, key=str)

[Ideal (z^2 + 1, y + 1) of Multivariate Polynomial Ring in x, y, z over Rational Field,
  Ideal (z^3 + 2, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational Field]
```

ALGORITHM:
Uses Singular.

REFERENCES:


basis_is_groebner(singular=Singular)
Return True if the generators of this ideal (\( \text{self.gens()} \)) form a Groebner basis.
Let $I$ be the set of generators of this ideal. The check is performed by trying to lift $\text{Syz}(LM(I))$ to $\text{Syz}(I)$ as $I$ forms a Groebner basis if and only if for every element $S$ in $\text{Syz}(LM(I))$:

$$S \ast G = \sum_{i=0}^{m} h_i g_i > G \ast 0.$$

**ALGORITHM:**

Uses Singular.

**EXAMPLES:**

```python
sage: R.<a,b,c,d,e,f,g,h,i,j> = PolynomialRing(GF(127),10)
sage: I = sage.rings.ideal.Cyclic(R,4)
sage: I.basis_is_groebner()
False
sage: I2 = Ideal(I.groebner_basis())
sage: I2.basis_is_groebner()
True
```

A more complicated example:

```python
sage: R.<U6,U5,U4,U3,U2, u6,u5,u4,u3,u2, h> = PolynomialRing(GF(7583))
sage: I = [u6 + u5 + u4 + u3 + u2 - 3791*h, 
        U6 + U5 + U4 + U3 + U2 - 3791*h, 
        U2*u2 - h^2, U3*u3 - h^2, U4*u4 - h^2, 
        U5*u4 + U5*u3 + U4*u3 + U5*u2 + U4*u2 + U3*u2 - 3791*U5*h -
            → 3791*U4*h - 3791*U3*h - 3791*U2*h - 2842*h^2, 
        U4*u5 + U3*u5 + U2*u5 + U3*u4 + U2*u4 + U2*u3 - 3791*U5*h -
            → 3791*u4*h - 3791*u3*h - 3791*u2*h - 2842*h^2, 
        U5*u5 - h^2, U4*U2*u3 + U5*U3*u2 + U4*U3*u2 + U3^2*u2 - 3791*U5*U3*h,
            → - 3791*U4*U3*h - 3791*U3^2*h - 3791*U5*U2*h
                - 3791*U4*U2*h + U3*U2*h - 3791*U2^2*h - 3791*U4*U3*h -
            → 3791*U4*u2*h - 3791*U3*u2*h - 2843*U5*h^2 + 1897*U4*h^2 - 946*U3*h^2 -
                → 947*U2*h^2 + 2376*h^3, 
        U3*u5*u4 + U2*u5*u4 + U3*u4^2 + U2*u4^2 + U2*u4*u3 - 3791*U5*u4*h -
            → 3791*u4^2*h - 3791*u4*u3*h - 3791*u4*u2*h + U5*h^2 - 2842*u4*h^2, 
        U2*u5*u4*u3 + U2*u4*u2*u3 + U2*u4*u3^2 - 3791*u5*u4*u3*h - 3791*u4^*
            → 2*u3*h - 3791*u4*u3^2*h - 3791*u3*u2*h + U5*u4*h^2 + U4^2*h^2 + U5*u3^2*h -
                → 2842*u4*u3^2*h, 
        U5^2*U4*u3 + U5*U4^2*u3 + U5^2*u4*u3 + U5^2*U4^2*u2 + U5^2*U3*u2 +
            → 2*U5*U4*U3*u2 + U5*U3^2*u2 - 3791*U5*U2^2*h - 3791*U5*U4*u2*h - 3791*U5*U3*U2*h,
            → \left. + U5*U4*U3*h - 3791*U5*U3^2*h - 3791*U5*U2^2*h + U5*U4*U2*h +
                    → U5*U3*U2*h - 3791*U5*U2*h - 2842*U5*U2^2*h + 1897*U5*U4*h^2
                → 2 \left. - U4*U2^2*h - 947*U5*U3*h^2 - U4*U3*h^2 - 948*U5*U2*h^2 - U4*U2*h^2 -
                    → 1422*U5*U3*h^3 + 3791*U4*U3*h^3, \right.
        u5*u4^3*u3^2*u2*h + u4^2*u3^2*u2*h + u4*u3^2*u2^2*h + u4*u3^2*u2^2*h +
            → 2*u5*u4*u3^2*h^2 + 2*u4^2*u3^2*h^2 + 2*u4*u3^2*u2^2*h + 2*u4*u3^2*u2^2*h + 2*u4^2*u2^2*h^2 -
                → 2 \left. + 2*u5*u3*u2^2*h + 1899*u4*u3*u2^2*h^2, \right.
        U5^2*U4*U3*u2 + U5*U4^2*U3*u2 + U5*U4*U3^2*u2 - 3791*U5*U2*U3*h,
            → 3791*U5*U4^2*U3*h - 3791*U5*U4*U3^2*h - 3791*U5*U4*U3^2*h - 3791*U5*U4^2*U3*h,
            → \left. + 3791*U5*U4*U3*u2^2*h + U5*U4^2*u2*h + U5*U4^2*u2*h + U5*U4^2*U3*h^2 - U4^*
                    → 2*U3^2*h^2 - 115*U3^2*h^2 + 2*U3^2*h^2 - 115*U4*U2^2*h^2 \right)
```

(continues on next page)
- U5*U3*U2*h^2 - U4*U3*U2*h^2 + 3791*U5*U4*h^3 + 3791*U5*U3*h^3 +,\n˓→3791*U4*U3*h^3, \n˓→ u4^2*u3*u2*h^2 + 1515*u5*u3^2*u2*h^2 + u4*u3^2*u2*h^2 +,\n˓→1515*u5*u4*u2^2*h^2 + 1515*u5*u3*u2^2*h^2 + 1521*u5*u4*u3*h^3 - 3028*u4^2*u3*h^3 - 3028*u4*u3^2*h^3 +,\n˓→1521*u5*u4*u2*h^3 - 3028*u4^2*u2*h^3 + 1521*u5*u3*u2*h^3 + 3420*u4*u3*u2*h^3, \n˓→ \n˓→ U5^2*U4*U3*U2*h + U5*U4^2*U3*U2*h + U5*U4*U3^2*U2*h + U5*U4*U3*U2^2*h + 2*U5^2*U4*U2*h^2 + 2*U5*U4^2*U2*h^2 + 2*U5*U4*U3^2*h^2 + 2*U5^2*U3*U2*h^2 - 2*U4^2*U3*U2^2*h^2 - 2*U4*U3^2*U2*h^2 - 2*U5*U4*U2^2*h^2 - 2*U5*U3*U2^2*h^2 - 2*U5*U4*U3*h^3 - 2*U5*U4*U2*h^3 - 2*U5*U3*U2*h^3 - U5*U4*U3*U2*h^3 - U5*U4*U3*U2*h^3 - U4*U3*U2*h^3]

sage: Ideal(l).basis_is_groebner()
False
sage: gb = Ideal(l).groebner_basis()
sage: Ideal(gb).basis_is_groebner()
True

Note: From the Singular Manual for the reduce function we use in this method: ‘The result may have no meaning if the second argument (self) is not a standard basis’. I (malb) believe this refers to the mathematical fact that the results may have no meaning if self is not a standard basis, i.e., Singular doesn’t ‘add’ any additional ‘nonsense’ to the result. So we may actually use reduce to determine if self is a Groebner basis.

**complete_primary_decomposition**(object)
A decorator that creates a cached version of an instance method of a class.

Note: For proper behavior, the method must be a pure function (no side effects). Arguments to the method must be hashable or transformed into something hashable using key or they must define **sage.structure.sage_object.SageObject._cache_key()**.

EXAMPLES:

```
sage: class Foo(object):
    ....:     @cached_method
    ....:     def f(self, t, x=2):
    ....:         print('computing')
    ....:         return t**x
sage: a = Foo()
```

The example shows that the actual computation takes place only once, and that the result is identical for equivalent input:

```
sage: res = a.f(3, 2); res
computing 9
sage: a.f(t = 3, x = 2) is res
True
```
Note, however, that the CachedMethod is replaced by a CachedMethodCaller or CachedMethodCallerNoArgs as soon as it is bound to an instance or class:

```
sage: P.<a,b,c,d> = QQ[]
sage: I = P*[a,b]
sage: type(I._.class_.gens)
<type 'sage.misc.cachefunc.CachedMethodCallerNoArgs'>
```

So, you would hardly ever see an instance of this class alive.

The parameter key can be used to pass a function which creates a custom cache key for inputs. In the following example, this parameter is used to ignore the algorithm keyword for caching:

```
sage: class A(object):
    ....:     def _f_normalize(self, x, algorithm):
    ....:         return x
    ....:     @cached_method(key=_f_normalize)
    ....:     def f(self, x, algorithm='default'):
    ....:         return x

sage: a = A()
sage: a.f(1, algorithm="default") is a.f(1) is a.f(1, algorithm="algorithm")
True
```

The parameter do_pickle can be used to enable pickling of the cache. Usually the cache is not stored when pickling:

```
sage: class A(object):
    ....:     @cached_method
    ....:     def f(self, x):
    ....:         return None

sage: import __main__
sage: __main__.A = A
sage: a = A()
sage: a.f(1)
1
sage: len(a.f.cache)
1
sage: b = loads(dumps(a))
sage: len(b.f.cache)
0
```

When do_pickle is set, the pickle contains the contents of the cache:

```
sage: class A(object):
    ....:     @cached_method(do_pickle=True)
    ....:     def f(self, x):
    ....:         return None

sage: __main__.A = A
sage: a = A()
sage: a.f(1)
1
sage: len(a.f.cache)
1
sage: b = loads(dumps(a))
sage: len(b.f.cache)
1
```
Cached methods cannot be copied like usual methods, see trac ticket #12603. Copying them can lead to very surprising results:

```python
sage: class A:
    ....:     @cached_method
    ....:     def f(self):
    ....:         return 1
sage: class B:
    ....:     g=A.f
    ....:     def f(self):
    ....:         return 2
sage: b=B()
sage: b.f()  # return 2
sage: b.g()  # return 1
sage: b.f()  # return 1
```

**dimension** *(singular='singular_default')*

The dimension of the ring modulo this ideal.

**EXAMPLES:**

```python
sage: P.<x,y,z> = PolynomialRing(GF(32003),order='degrevlex')
sage: I = ideal(x^2-y,x^3)
sage: I.dimension()
1
```

If the ideal is the total ring, the dimension is $-1$ by convention.

For polynomials over a finite field of order too large for Singular, this falls back on a toy implementation of Buchberger to compute the Groebner basis, then uses the algorithm described in Chapter 9, Section 1 of Cox, Little, and O’Shea’s “Ideals, Varieties, and Algorithms”.

**EXAMPLES:**

```python
sage: R.<x,y> = PolynomialRing(GF(2147483659),order='lex')
sage: I = R.ideal([x*y,x*y+1])
sage: I.dimension()  # very slow toy implementation.
-1
sage: I=ideal([x^(x*y+1),y^(x*y+1)])
sage: I.dimension()  # very slow toy implementation.
1
sage: I = R.ideal([x^3*y,x^y^2])
sage: I.dimension()  # very slow toy implementation.
1
sage: R.<x,y> = PolynomialRing(GF(2147483659),order='lex')
```

(continues on next page)
sage: I = R.ideal(0)
sage: I.dimension()
verbose 0 (....: multi_polynomial_ideal.py, dimension) Warning: falling back to very slow toy implementation.
2

ALGORITHM:
Uses Singular, unless the characteristic is too large.

Note: Requires computation of a Groebner basis, which can be a very expensive operation.

elimination_ideal(variables, algorithm=None, *args, **kwds)
Return the elimination ideal of this ideal with respect to the variables given in variables.

INPUT:

• variables – a list or tuple of variables in self.ring()
• algorithm - determines the algorithm to use, see below for available algorithms.

ALGORITHMS:

• 'libsingular:eliminate' – libSingular’s eliminate command (default)
• 'giac:eliminate' – Giac’s eliminate command (if available)

If only a system is given - e.g. ‘giac’ - the default algorithm is chosen for that system.

EXAMPLES:

sage: R.<x,y,t,s,z> = PolynomialRing(QQ,5)
sage: I = R * [x-t,y-t^2,z-t^3,s-x+y^3]
sage: J = I.elimination_ideal([t,s]); J
Ideal (y^2 - x*z, x*y - z, x^2 - y) of Multivariate Polynomial Ring in x, y, t, s, z over Rational Field

You can use Giac to compute the elimination ideal:

sage: print("possible output from giac", flush=True); I.elimination_ideal([t,˓→s], algorithm="giac") == J
possible output...
True

The list of available Giac options is provided at sage.libs.giac.groebner_basis().

ALGORITHM:
Uses Singular, or Giac (if available).

Note: Requires computation of a Groebner basis, which can be a very expensive operation.

genus(object)
A decorator that creates a cached version of an instance method of a class.
**Note:** For proper behavior, the method must be a pure function (no side effects). Arguments to the method must be hashable or transformed into something hashable using `key` or they must define `sage.structure.sage_object.SageObject._cache_key()`.

**EXAMPLES:**

```python
sage: class Foo(object):
    ....:     @cached_method
    ....:     def f(self, t, x=2):
    ....:         print('computing')
    ....:         return t**x
sage: a = Foo()
```

The example shows that the actual computation takes place only once, and that the result is identical for equivalent input:

```python
sage: res = a.f(3, 2); res
computing
9
sage: a.f(t = 3, x = 2) is res
True
sage: a.f(3) is res
True
```

Note, however, that the `CachedMethod` is replaced by a `CachedMethodCaller` or `CachedMethodCallerNoArgs` as soon as it is bound to an instance or class:

```python
sage: P.<a,b,c,d> = QQ[]
sage: I = P*[a,b]
sage: type(I.__class__.gens)
<type 'sage.misc.cachefunc.CachedMethodCallerNoArgs'>
```

So, you would hardly ever see an instance of this class alive.

The parameter `key` can be used to pass a function which creates a custom cache key for inputs. In the following example, this parameter is used to ignore the `algorithm` keyword for caching:

```python
sage: class A(object):
    ....:     def _f_normalize(self, x, algorithm):
    ....:         return x
    ....:     @cached_method(key=_f_normalize)
    ....:     def f(self, x, algorithm='default'):
    ....:         return x
sage: a = A()
sage: a.f(1, algorithm="default") is a.f(1) is a.f(1, algorithm="algorithm")
True
```

The parameter `do_pickle` can be used to enable pickling of the cache. Usually the cache is not stored when pickling:

```python
sage: class A(object):
    ....:     @cached_method
    ....:     def f(self, x):
    ....:         return None
sage: import __main__
sage: __main__.A = A
```

(continues on next page)
When do\_pickle is set, the pickle contains the contents of the cache:

```python
sage: class A(object):
    ....:     @cached_method(do_pickle=True)
    ....:     def f(self, x):
    ....:         return None

sage: __main__.A = A
sage: a = A()
sage: a.f(1)
1
sage: b = loads(dumps(a))
sage: len(b.f.cache)
1
```

Cached methods cannot be copied like usual methods, see trac ticket #12603. Copying them can lead to very surprising results:

```python
sage: class A:
    ....:     @cached_method
    ....:     def f(self):
    ....:         return 1

sage: class B:
    ....:     g=A.f
    ....:     def f(self):
    ....:         return 2

sage: b=B()
sage: b.f()
2
sage: b.g()
1
sage: b.f()
1
```

**hilbert\_numerator** *(grading=None, algorithm='sage')*

Return the Hilbert numerator of this ideal.

**INPUT:**

- **grading** – (optional) a list or tuple of integers
- **algorithm** – (default: 'sage') must be either 'sage' or 'singular'

Let \( I \) (which is self) be a homogeneous ideal and \( R = \bigoplus_d R_d \) (which is self.ring()) be a graded commutative algebra over a field \( K \). Then the **Hilbert function** is defined as \( H(d) = \dim_K R_d \) and the **Hilbert series** of \( I \) is defined as the formal power series \( HS(t) = \sum_{d=0}^{\infty} H(d)t^d \).
This power series can be expressed as \( HS(t) = Q(t)/(1 - t^n) \) where \( Q(t) \) is a polynomial over \( Z \) and \( n \) the number of variables in \( R \). This method returns \( Q(t) \), the numerator; hence the name, \texttt{hilbert_numerator}. An optional grading can be given, in which case the graded (or weighted) Hilbert numerator is given.

**EXAMPLES:**

```python
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: I = Ideal([x^3*y^2 + 3*x^2*y^2*z + y^3*z^2 + z^5])
sage: I.hilbert_numerator()
-t^5 + 1
sage: R.<a,b> = PolynomialRing(QQ)
sage: J = R.ideal([a^2*b,a*b^2])
sage: J.hilbert_numerator()
t^4 - 2*t^3 + 1
sage: J.hilbert_numerator(grading=(10,3))
t^26 - t^23 - t^16 + 1
```

\texttt{hilbert_polynomial(algorithm='sage')}

Return the Hilbert polynomial of this ideal.

**INPUT:**

- \texttt{algorithm} – (default: 'sage') must be either 'sage' or 'singular'

Let \( I \) (which is self) be a homogeneous ideal and \( R = \bigoplus_d R_d \) (which is self.ring()) be a graded commutative algebra over a field \( K \). The *Hilbert polynomial* is the unique polynomial \( HP(t) \) with rational coefficients such that \( HP(d) = \dim_K R_d \) for all but finitely many positive integers \( d \).

**EXAMPLES:**

```python
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: I = Ideal([x^3*y^2 + 3*x^2*y^2*z + y^3*z^2 + z^5])
sage: I.hilbert_polynomial()
5*t - 5
Of course, the Hilbert polynomial of a zero-dimensional ideal is zero:

```python
sage: J0 = Ideal([x^3*y^2 + 3*x^2*y^2*z + y^3*z^2 + z^5, y^3-2*x*z^2+x*y,x^→4+x*y-y*z^2])
sage: J = P*[m.lm() for m in J0.groebner_basis()]
sage: J.dimension()
0
sage: J.hilbert_polynomial()
0
```

It is possible to request a computation using the Singular library:

```python
sage: I.hilbert_polynomial(algorithm = 'singular') == I.hilbert_polynomial()
True
sage: J.hilbert_polynomial(algorithm = 'singular') == J.hilbert_polynomial()
True
```

Here is a bigger examples:
```python
sage: n = 4; m = 11; P = PolynomialRing(QQ, n * m, "x"); x = P.gens(); M =
    → Matrix(n, x)
sage: Minors = P.ideal(M.minors(2))
sage: hp = Minors.hilbert_polynomial(); hp
1/21772800*t^13 + 61/21772800*t^12 + 1661/21772800*t^11
+ 26681/21772800*t^10 + 93841/7257600*t^9 + 685421/7257600*t^8
+ 152489/3110400*t^7 + 9780323/21772800*t^6 + 6638071/1360800*t^5
+ 12509761/1360800*t^4 + 26890323/21772800*t^3 + 1494509/151200*t^2
+ 12001/2520*t + 1
```

Because Singular uses 32-bit integers, the above example would fail with Singular. We don’t test it here, as it has a side-effect on other tests that is not understood yet (see trac ticket #26300):

```python
sage: Minors.hilbert_polynomial(algorithm = 'singular')  # not tested
Traceback (most recent call last):
...
RuntimeError: error in Singular function call 'hilbPoly':
int overflow in hilb 1
error occurred in or before poly.lib::hilbPoly line 58: `intvec v=hilb(I,2);
expected intvec-expression. type 'help intvec';
```

Note that in this example, the Hilbert polynomial gives the coefficients of the Hilbert-Poincaré series in all degrees:

```python
sage: P = PowerSeriesRing(QQ, 't', default_prec = 50)
sage: hs = Minors.hilbert_series()
sage: list(P(hs.numerator()) / P(hs.denominator())) == [hp(t = k) for k in
    → range(50)]
True
```

### hilbert_series(grading=None, algorithm='sage')

Return the Hilbert series of this ideal.

**INPUT:**

- **grading** – (optional) a list or tuple of integers
- **algorithm** – (default: 'sage') must be either 'sage' or 'singular'

Let $I$ (which is self) be a homogeneous ideal and $R = \bigoplus_d R_d$ (which is self.ring()) be a graded commutative algebra over a field $K$. Then the Hilbert function is defined as $H(d) = \dim_K R_d$ and the Hilbert series of $I$ is defined as the formal power series $H_S(t) = \sum_{d=0}^{\infty} H(d) t^d$.

This power series can be expressed as $H_S(t) = Q(t)/(1 - t)^n$ where $Q(t)$ is a polynomial over $Z$ and $n$ the number of variables in $R$. This method returns $Q(t)/(1 - t)^n$, normalised so that the leading monomial of the numerator is positive.

An optional grading can be given, in which case the graded (or weighted) Hilbert series is given.

### EXAMPLES:

```python
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: I = Ideal([x^3*y^2 + 3*x^2*y^2*z + y^3*z^2 + z^5])
sage: I.hilbert_series()
(t^4 + t^3 + t^2 + t + 1)/(t^4 - 2*t^3 + 1)
sage: R.<a,b> = PolynomialRing(QQ)
sage: J = R.ideal([a^2*b, a*b^2])
```

(continues on next page)
integral_closure(p=0, r=True, singular='singular_default')

Let \(I = \text{self}\).

Return the integral closure of \(I, \ldots, I^p\), where \(sI\) is an ideal in the polynomial ring \(R = k[x(1), \ldots x(n)]\). If \(p\) is not given, or \(p = 0\), compute the closure of all powers up to the maximum degree in \(t\) occurring in the closure of \(R[It]\) (so this is the last power whose closure is not just the sum/product of the smaller). If \(r\) is given and \(r\) is True, \(I\).integral_closure() starts with a check whether \(I\) is already a radical ideal.

INPUT:

- \(p\) - powers of \(I\) (default: 0)
- \(r\) - check whether \(self\) is a radical ideal first (default: True)

EXAMPLES:

```python
sage: R.<x,y> = QQ[]
sage: I = ideal([x^2,x*y^4,y^5])
sage: I.integral_closure()
[x^2, x*y^4, y^5, x*y^3]
```

ALGORITHM:

Uses libSINGULAR.

interreduced_basis()

If this ideal is spanned by \((f_1, \ldots, f_n)\) this method returns \((g_1, \ldots, g_s)\) such that:

- \((f_1, \ldots, f_n) = (g_1, \ldots, g_s)\)
- \(LT(g_i)! = LT(g_j)\) for all \(i! = j\)
- \(LT(g_i)\) does not divide \(m\) for all monomials \(m\) of \([g_1, \ldots, g_i-1, g_{i+1}, \ldots, g_s]\)
- \(LC(g_i) == 1\) for all \(i\) if the coefficient ring is a field.

EXAMPLES:

```python
sage: R.<x,y,z> = PolynomialRing(QQ)
sage: I = Ideal([z^2*x+y+z, y^3, z+x])
sage: I.interreduced_basis()
[y^3 + z, x^2*y + z, x*z - z]
```

Note that tail reduction for local orderings is not well-defined:
```python
sage: R.<x,y,z> = PolynomialRing(QQ, order='negdegrevlex')
sage: I = Ideal([z^2*x+y^3, z+y^3, z+x*y])
sage: I.interreduced_basis()
[z + x^2*y, x^2*y - y^3, x^2*y - y^3]
```

A fixed error with nonstandard base fields:

```python
sage: R.<t>=QQ['t']
sage: K.<x,y>=R.fraction_field()['x,y']
sage: I=t*x*K
sage: I.interreduced_basis()
[x]
```

The interreduced basis of 0 is 0:

```python
sage: P.<x,y,z> = GF(2)[]
sage: Ideal(P(0)).interreduced_basis()
[0]
```

**ALGORITHM:**

Uses Singular’s interred command or `sage.rings.polynomial.toy_buchberger.inter_reduction()` if conversion to Singular fails.

**intersection(**-*others*)**

Return the intersection of the arguments with this ideal.

**EXAMPLES:**

```python
sage: R.<x,y> = PolynomialRing(QQ, 2, order='lex')
sage: I = x*R
sage: J = y*R
sage: I.intersection(J)
Ideal (x*y) of Multivariate Polynomial Ring in x, y over Rational Field
```

The following simple example illustrates that the product need not equal the intersection.

```python
sage: I = (x^2, y)*R
sage: J = (y^2, x)*R
sage: K = I.intersection(J); K
Ideal (y^2, x*y, x^2) of Multivariate Polynomial Ring in x, y over Rational Field
sage: IJ = I*J; IJ
Ideal (x^2*y^2, x^3, y^3, x*y) of Multivariate Polynomial Ring in x, y over Rational Field
sage: IJ == K
False
```

Intersection of several ideals:

```python
sage: R.<x,y,z> = PolynomialRing(QQ, 3, order='lex')
sage: I1 = x*R
sage: I2 = y*R
sage: I3 = (x, y)*R
sage: I4 = (x^2 + x*y*z, y^2 - z^3*y, z^3 + y^5*x^2*z)*R
```

(continues on next page)
The ideals must share the same ring:

```python
sage: R2.<x,y> = PolynomialRing(QQ, 2, order='lex')
sage: R3.<x,y,z> = PolynomialRing(QQ, 3, order='lex')
sage: I2 = x*R2
sage: I3 = x*R3
sage: I2.intersection(I3)
Traceback (most recent call last):
...  
TypeError: Intersection is only available for ideals of the same ring.
```

**is_prime(****kwds**)

Return True if this ideal is prime.

INPUT:

- keyword arguments are passed on to `complete_primary_decomposition`; in this way you can specify the algorithm to use.

EXAMPLES:

```python
sage: R.<x, y> = PolynomialRing(QQ, 2)
sage: I = (x^2 - y^2 - 1)*R
sage: I.is_prime()
True
sage: (I^2).is_prime()
False
sage: J = (x^2 - y^2)*R
sage: J.is_prime()
False
sage: (J^3).is_prime()
False
sage: (I * J).is_prime()
False
```

The following is trac ticket #5982. Note that the quotient ring is not recognized as being a field at this time, so the fraction field is not the quotient ring itself:

```python
sage: Q = R.quotient(I); Q
Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x^2 - y^2 - 1)
sage: Q.fraction_field()
Fraction Field of Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x^2 - y^2 - 1)
```

**minimal_associated_primes()**

**OUTPUT:**

- list - a list of prime ideals

**EXAMPLES:**
sage: R.<x,y,z> = PolynomialRing(QQ, 3, 'xyz')
sage: p = z^2 + 1; q = z^3 + 2
sage: I = (p*q^2, y-z^2)*R
sage: sorted(I.minimal_associated_primes(), key=str)
[Ideal (z^2 + 1, -z^2 + y) of Multivariate Polynomial Ring in x, y, z over Rational Field, Ideal (z^3 + 2, -z^2 + y) of Multivariate Polynomial Ring in x, y, z over Rational Field]

ALGORITHM:
Uses Singular.

normal_basis(degree=None, algorithm='libsingular', singular='singular_default')
Return a vector space basis of the quotient ring of this ideal.

INPUT:

• degree – integer (default: None)
• algorithm – string (default: "libsingular"); if not the default, this will use the kbase() or weightKB() command from Singular
• singular – the singular interpreter to use when algorithm is not "libsingular" (default: the default instance)

OUTPUT:
Monomials in the basis. If degree is given, only the monomials of the given degree are returned.

EXAMPLES:

Monomials in the basis. If degree is given, only the monomials of the given degree are returned.

sage: R.<x,y,z> = PolynomialRing(QQ)
sage: I = R.ideal(x^2+y^2+z^2-4, x^2+2*y^2-5, x*z-1)
sage: I.normal_basis()
[y*z^2, z^2, y*z, z, x*y, y, x, 1]
sage: I.normal_basis(algorithm='singular')
[y*z^2, z^2, y*z, z, x*y, y, x, 1]

The result can be restricted to monomials of a chosen degree, which is particularly useful when the quotient ring is not finite-dimensional as a vector space.

sage: J = R.ideal(x^2+y^2+z^2-4, x^2+2*y^2-5)
sage: J.dimension()
1
sage: [J.normal_basis(d) for d in (0..3)]
[[1], [z, y, x], [z^2, y*z, x*z, x*y], [z^3, y*z^2, x*z^2, x*y*z]]
sage: [J.normal_basis(d, algorithm='singular') for d in (0..3)]
[[1], [z, y, x], [z^2, y*z, x*z, x*y], [z^3, y*z^2, x*z^2, x*y*z]]

In case of a polynomial ring with a weighted term order, the degree of the monomials is taken with respect to the weights.

sage: T = TermOrder('wdegrevlex', (1, 2, 3))
sage: R.<x,y,z> = PolynomialRing(QQ, order=T)
sage: B = R.ideal(x*y^2 + x^5, z*y + x^3*y).normal_basis(9); B
[x^2*y^2*z, x^3*z^2, x*y*z^2, z^3]
plot(singular=Singular)

If you somehow manage to install surf, perhaps you can use this function to implicitly plot the real zero
locus of this ideal (if principal).

INPUT:

- self - must be a principal ideal in 2 or 3 vars over $\mathbb{Q}$.

EXAMPLES:

Implicit plotting in 2-d:

```python
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: I = R.ideal([y^3 - x^2])
sage: I.plot()  # cusp
Graphics object consisting of 1 graphics primitive
sage: I = R.ideal([y^2 - x^2 - 1])
sage: I.plot()  # hyperbola
Graphics object consisting of 1 graphics primitive
sage: I = R.ideal([y^2 + x^2*(1/4) - 1])
sage: I.plot()  # ellipse
Graphics object consisting of 1 graphics primitive
sage: I = R.ideal([y^2-(x^2-1)*(x-2)])
sage: I.plot()  # elliptic curve
Graphics object consisting of 1 graphics primitive
```

Implicit plotting in 3-d:

```python
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: I = R.ideal([y^2 + x^2*(1/4) - z])
sage: I.plot()  # a cone; optional - surf
sage: I = R.ideal([y^2 + z^2*(1/4) - x])
sage: I.plot()  # same code, from a different angle; optional - surf
sage: I = R.ideal([x^2*y^2+x^2*z^2+y^2*z^2-16*x*y*z])
sage: I.plot()  # Steiner surface; optional - surf
```

AUTHORS:

- David Joyner (2006-02-12)

primary_decomposition(algorithm='sy')

Return a list of primary ideals such that their intersection is self.

An ideal $Q$ is called primary if it is a proper ideal of the ring $R$, and if whenever $ab \in Q$ and $a \notin Q$, then $b^n \in Q$ for some $n \in \mathbb{Z}$.

If $Q$ is a primary ideal of the ring $R$, then the radical ideal $P$ of $Q$ (i.e. the ideal consisting of all $a \in R$ with $a^n \in Q$ for some $n \in \mathbb{Z}$), is called the associated prime of $Q$.

If $I$ is a proper ideal of a Noetherian ring $R$, then there exists a finite collection of primary ideals $Q_i$ such that the following hold:

- the intersection of the $Q_i$ is $I$;
- none of the $Q_i$ contains the intersection of the others;
the associated prime ideals of the \( Q_i \) are pairwise distinct.

**INPUT:**

- **algorithm** – string:
  - `'sy'` – (default) use the Shimoyama-Yokoyama algorithm
  - `'gtz'` – use the Gianni-Trager-Zacharias algorithm

**OUTPUT:**

- a list of primary ideals \( Q_i \) forming a primary decomposition of \( \text{self} \).

**EXAMPLES:**

```
R.<x,y,z> = PolynomialRing(QQ, 3, order='lex')
p = z^2 + 1; q = z^3 + 2
I = (p*q^2, y-z^2)*R
pd = I.primary_decomposition(); sorted(pd, key=str)
```

\[
\begin{array}{ll}
\text{Ideal (z^2 + 1, y + 1) of Multivariate Polynomial Ring in x, y, z over Rational Field}, \\
\text{Ideal (z^6 + 4*z^3 + 4, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational Field}
\end{array}
\]

```
sage: from functools import reduce
sage: reduce(lambda Qi,Qj: Qi.intersection(Qj), pd) == I
True
```

**ALGORITHM:**

Uses Singular.

**REFERENCES:**


**primary_decomposition_complete**(object)

A decorator that creates a cached version of an instance method of a class.

**Note:** For proper behavior, the method must be a pure function (no side effects). Arguments to the method must be hashable or transformed into something hashable using `key` or they must define `sage.structure.sage_object.SageObject._cache_key()`.

**EXAMPLES:**

```
sage: class Foo(object):
    ....:     @cached_method
    ....:     def f(self, t, x=2):
    ....:         print('computing')
    ....:         return t**x
sage: a = Foo()
```

The example shows that the actual computation takes place only once, and that the result is identical for equivalent input:
sage: res = a.f(3, 2); res
computing
9
sage: a.f(t = 3, x = 2) is res
True
sage: a.f(3) is res
True

Note, however, that the CachedMethod is replaced by a CachedMethodCaller or
CachedMethodCallerNoArgs as soon as it is bound to an instance or class:

sage: P.<a,b,c,d> = QQ[]
sage: I = P*[a,b]
sage: type(I.__class__.gens)
<type 'sage.misc.cachefunc.CachedMethodCallerNoArgs'>

So, you would hardly ever see an instance of this class alive.

The parameter key can be used to pass a function which creates a custom cache key for inputs. In the
following example, this parameter is used to ignore the algorithm keyword for caching:

sage: class A(object):
    ....: def _f_normalize(self, x, algorithm):
    ....:     return x
    ....: @cached_method(key=_f_normalize)
    ....: def f(self, x, algorithm='default'):
    ....:     return x
sage: a = A()
sage: a.f(1, algorithm="default") is a.f(1) is a.f(1, algorithm="algorithm")
True

The parameter do_pickle can be used to enable pickling of the cache. Usually the cache is not stored
when pickling:

sage: class A(object):
    ....: @cached_method
    ....: def f(self, x):
    ....:     return None
sage: import __main__
sage: __main__.A = A
sage: a = A()
sage: a.f(1)
sage: len(a.f.cache)
1
sage: b = loads(dumps(a))
sage: len(b.f.cache)
0

When do_pickle is set, the pickle contains the contents of the cache:

sage: class A(object):
    ....: @cached_method(do_pickle=True)
    ....: def f(self, x):
    ....:     return None
sage: __main__.A = A
sage: a = A()
sage: a.f(1)
sage: len(a.f.cache)

(continues on next page)
Cached methods cannot be copied like usual methods, see trac ticket #12603. Copying them can lead to very surprising results:

```python
sage: class A:
....:     @cached_method
....:     def f(self):
....:         return 1
sage: class B:
....:     g = A.f
....:     def f(self):
....:         return 2
sage: b = B()
sage: b.f()  # 2
sage: b.g()  # 1
sage: b.f()  # 1
```

**quotient**(J)

Given ideals \( I = \text{self} \) and \( J \) in the same polynomial ring \( P \), return the ideal quotient of \( I \) by \( J \) consisting of the polynomials \( a \) of \( P \) such that \( \{aJ \subset I\} \).

This is also referred to as the colon ideal \((I:J)\).

**INPUT:**

- \( J \) - multivariate polynomial ideal

**EXAMPLES:**

```python
sage: R.<x,y,z> = PolynomialRing(GF(181),3)
sage: I = Ideal([x^2+x*y*z, y^2-z^3*y, z^3+y^5*x*z])
sage: J = Ideal([x])
sage: Q = I.quotient(J)
sage: y*z + x in I  # False
sage: x in J  # True
sage: x * (y*z + x) in I  # True
```

**radical**()

The radical of this ideal.

**EXAMPLES:**

This is an obviously not radical ideal:
sage: R.<x,y,z> = PolynomialRing(QQ, 3)
sage: I = (x^2, y^3, (x*z)^4 + y^3 + 10*x^2)*R
sage: I.radical()
Ideal (y, x) of Multivariate Polynomial Ring in x, y, z over Rational Field

That the radical is correct is clear from the Groebner basis.

sage: I.groebner_basis()
[y^3, x^2]

This is the example from the Singular manual:

sage: p = z^2 + 1; q = z^3 + 2
sage: I = (p*q^2, y-z^2)*R
sage: I.radical()
Ideal (z^2 - y, y^2*z + y*z + 2*y + 2) of Multivariate Polynomial Ring in x, y, z over Rational Field

Note: From the Singular manual: A combination of the algorithms of Krick/Logar and Kemper is used. Works also in positive characteristic (Kemper’s algorithm).

sage: R.<x,y,z> = PolynomialRing(GF(37), 3)
sage: p = z^2 + 1; q = z^3 + 2
sage: I = (p*q^2, y-z^2)*R
sage: I.radical()
Ideal (z^2 - y, y^2*z + y*z + 2*y + 2) of Multivariate Polynomial Ring in x, y, z over Finite Field of size 37

### saturation(other)

Return the saturation (and saturation exponent) of the ideal self with respect to the ideal other

**INPUT:**

- other – another ideal in the same ring

**OUTPUT:**

- a pair (ideal, integer)

**EXAMPLES:**

sage: R.<x,y,z> = QQ[]
sage: I = R.ideal(x^4*z^3, x*y*z, y*z^4)
sage: J = R.ideal(z)
sage: I.saturation(J)
(Ideal (y, x^5) of Multivariate Polynomial Ring in x, y, z over Rational Field, 4)

### syzygy_module()

Computes the first syzygy (i.e., the module of relations of the given generators) of the ideal.

**EXAMPLES:**

sage: R.<x,y> = PolynomialRing(QQ)
sage: f = 2*x^2 + y
transformed_basis\(\) (algorithm='gwalk', other_ring=None, singular='singular_default')

Return a lex or other_ring Groebner Basis for this ideal.

INPUT:

- **algorithm** - see below for options.
- **other_ring** - only valid for algorithm ‘fglm’, if provided conversion will be performed to this ring. Otherwise a lex Groebner basis will be returned.

ALGORITHMS:

- **fglm** - FGLM algorithm. The input ideal must be given with a reduced Groebner Basis of a zero-dimensional ideal
- **gwalk** - Groebner Walk algorithm (default)
- **awalk1** - ‘first alternative’ algorithm
- **awalk2** - ‘second alternative’ algorithm
- **twalk** - Tran algorithm
- **fwalk** - Fractal Walk algorithm

EXAMPLES:

\[
\begin{align*}
sage: R.<x,y,z> & = PolynomialRing(QQ,3) \\
sage: I & = Ideal([y^3+x^2, x^2*y+x^2, x^3-x^2, z^4-x^2-y]) \\
sage: J & = Ideal(I.transformed_basis('fglm', S)) \\
sage: J \\
Ideal (z^4 + y^3 - y, x^2 + y^3, x*y^3 - y^3, y^4 + y^3)
\]
of Multivariate Polynomial Ring in z, x, y over Rational Field

\[
\begin{align*}
sage: R.<z,y,x> & = PolynomialRing(GF(32003),3,order='lex') \\
sage: I & = Ideal([y*z+y^8+x^2*y^2, x*y^3, x*y^3, y^4 + y^3]) \\
sage: I & = Ideal(I.transformed_basis('gwalk')) \\
[z^8+y^8*x^2 + y^8*x + 3, z^8+y^8*x^2 + 8297*y^8*x^2 + 8297*y^8*x + 3556*y^7 - 15409*y^6*x + 15409*y^6*x^3 - 8297*y^6*x^2 + 8297*y^5*x^5 - 3556*y^5*x^2 + 3556*y^5*x + 3556*y^4*x^3]
\]
(continues on next page)

(continued from previous page)

+ 3556*y^4*x^2 - 10668*y^4 - 10668*y^3*x - 8297*y^2*x^9 - 1185*y^2*x^8 +
    14224*y^2*x^7
- 1185*y^2*x^6 - 8297*y^2*x^5 - 14223*y*x^7 - 10666*y*x^6 - 10666*y*x^5 -
    14223*y*x^4 + x^5 + 2*x^4 + x^3,

y^9 - y^7*x^2 - y^7*x - y^6*x^3 - y^6*x^2 - 3*y^6 - 3*y^5*x - y^3*x^7 - 3*y^3*x^6
- 3*y^3*x^5 - y^3*x^4 - 9*y^2*x^5 - 18*y^2*x^4 - 9*y^2*x^3 - 27*y*x^3 - 27*y*x^2
- 2 - 27*x]

ALGORITHM:

Uses Singular.

\texttt{triangular\_decomposition}(algorithm=None, singular='\texttt{singular\_default}')

Decompose zero-dimensional ideal self into triangular sets.

This requires that the given basis is reduced w.r.t. to the lexicographical monomial ordering. If the basis of self does not have this property, the required Groebner basis is computed implicitly.

INPUT:

- algorithm - string or None (default: None)

ALGORITHMS:

- \texttt{singular:triangL} - decomposition of self into triangular systems (Lazard).
- \texttt{singular:triangLfak} - decomp. of self into tri. systems plus factorization.
- \texttt{singular:triangM} - decomposition of self into triangular systems (Moeller).

OUTPUT: a list $T$ of lists $t$ such that the variety of self is the union of the varieties of $t$ in $L$ and each $t$ is in triangular form.

EXAMPLES:

\begin{verbatim}sage: P.<e,d,c,b,a> = PolynomialRing(QQ,5,order='lex')
sage: I = sage.rings.ideal.Cyclic(P)
sage: GB = Ideal(I.groebner_basis('libsingular:stdfglm'))
sage: GB.triangular_decomposition('singular:triangLfak')
[Ideal (a - 1, b - 1, c - 1, d^2 + 3*d + 1, e + d + 3) of Multivariate Polynomial Ring in e, d, c, b, a over Rational Field, Ideal (a - 1, b - 1, c^2 + 3*c + 1, d + c + 3, e - 1) of Multivariate Polynomial Ring in e, d, c, b, a over Rational Field, Ideal (a - 1, b^2 + 3*b + 1, c + b + 3, d - 1, e - 1) of Multivariate Polynomial Ring in e, d, c, b, a over Rational Field, Ideal (a - 1, b^4 + b^3 + b^2 + b + 1, -c + b^2, -d + b^3, e + b^3 + b^2 + b + 1) of Multivariate Polynomial Ring in e, d, c, b, a over Rational Field, Ideal (a^2 + 3*a + 1, b - 1, c - 1, d - 1, e + a + 3) of Multivariate Polynomial Ring in e, d, c, b, a over Rational Field, Ideal (a^2 + 3*a + 1, b + a + 3, c - 1, d - 1, e - 1) of Multivariate Polynomial Ring in e, d, c, b, a over Rational Field, Ideal (a^4 - 4*a^3 + 6*a^2 + a + 1, -11*b^2 + 6*b*a^3 - 26*b*a^2 + 41*b*a - 4*b^2 - 8*a^3 + 31*a^2 - 40*a - 24, 11*c + 3*a^3 - 13*a^2 + 26*a - 2, 11*d + 3*a^3 - 13*a^2 + 26*a - 2, -11*e - 11*b + 6*a^3 - 26*a^2 + 41*a - 4) of Multivariate Polynomial Ring in e, d, c, b, a over Rational Field, Ideal (a^4 + a^3 + a^2 + a + 1, b - 1, c + a^3 + a^2 + a + 1, -d + a^3, -e + a^2 - 2) of Multivariate Polynomial Ring in e, d, c, b, a over Rational Field]
\end{verbatim}
\begin{verbatim}
Ideal (a^4 + a^3 + a^2 + a + 1, b - a, c - a, d^2 + 3*d*a + a^2, e + d + 3*a) → of Multivariate Polynomial Ring in e, d, c, b, a over Rational Field,
Ideal (a^4 + a^3 + a^2 + a + 1, b - a, c^2 + 3*c*a + a^2, d + c + 3*a, e - a) → of Multivariate Polynomial Ring in e, d, c, b, a over Rational Field,
Ideal (a^4 + a^3 + a^2 + a + 1, b^2 + 3*b*a + a^2, c + b + 3*a, d - a, e - a) → of Multivariate Polynomial Ring in e, d, c, b, a over Rational Field,
Ideal (a^4 + a^3 + a^2 + a + 1, b^3 + b^2*a + b^2 + b*a^2 + b*a + b + a^3 + a^2 → + a + 1, c + b^2*a^3 + b^2*a^2 + b^2*a + b^2, -d + b^2*a^2 + b^2*a + b^2 + → b^2*a^2 + b*a + a^2, -e + b^2*a^3 - b*a^2 - b*a - b - a^2 - a) of Multivariate → Polynomial Ring in e, d, c, b, a over Rational Field,
Ideal (a^4 + a^3 + 6*a^2 - 4*a + 1, -11*b^2 + 6*b*a^3 + 10*b*a^2 + 39*b*a + 2*b → + 16*a^3 + 23*a^2 + 104*a - 24, 11*c + 3*a^3 + 5*a^2 + 25*a + 1, 11*d + 3*a^3 → + 5*a^2 + 25*a + 1, -11*e - 11*b + 6*a^3 + 10*a^2 + 39*a + 2) of Multivariate → Polynomial Ring in e, d, c, b, a over Rational Field]
\end{verbatim}

```
sage: R.<x1,x2> = PolynomialRing(QQ, 2, 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([f1,f2])
sage: I.triangular_decomposition()
[ Ideal (x2, x1^2) of Multivariate Polynomial Ring in x1, x2 over Rational Field, 
  Ideal (x2, x1^2) of Multivariate Polynomial Ring in x1, x2 over Rational Field, 
  Ideal (x2^4 + 4*x2^3 - 6*x2^2 - 20*x2 + 5, 8*x1 - x2^3 + x2^2 + 13*x2 - 5) of Multivariate Polynomial Ring in x1, x2 over Rational Field]
```

\textbf{variety}(\texttt{ring=None})

Return the variety of this ideal.

Given a zero-dimensional ideal \( I \) (== \texttt{self}) of a polynomial ring \( P \) whose order is lexicographic, return the variety of \( I \) as a list of dictionaries with \texttt{(variable, value)} pairs. By default, the variety of the ideal over its coefficient field \( K \) is returned; \texttt{ring} can be specified to find the variety over a different ring.

These dictionaries have cardinality equal to the number of variables in \( P \) and represent assignments of values to these variables such that all polynomials in \( I \) vanish.

If \texttt{ring} is specified, then a triangular decomposition of \texttt{self} is found over the original coefficient field \( K \); then the triangular systems are solved using root-finding over \texttt{ring}. This is particularly useful when \( K \) is \texttt{QQ} (to allow fast symbolic computation of the triangular decomposition) and \texttt{ring} is \texttt{RR}, \texttt{AA}, \texttt{CC}, or \texttt{QQbar} (to compute the whole real or complex variety of the ideal).

Note that with \texttt{ring=RR} or \texttt{CC}, computation is done numerically and potentially inaccurately; in particular, the number of points in the real variety may be miscomputed. With \texttt{ring=AA} or \texttt{QQbar}, computation is done exactly (which may be much slower, of course).

\textbf{INPUT:}

\begin{itemize}
  \item \texttt{ring} - return roots in the \texttt{ring} instead of the base ring of this ideal (default: \texttt{None})
  \item \texttt{proof} - return a provably correct result (default: \texttt{True})
\end{itemize}

\textbf{EXAMPLES:}

```
sage: K.<w> = GF(27) # this example is from the MAGMA handbook
sage: P.<x,y> = PolynomialRing(K, 2, order='lex')
sage: I = Ideal([x^8 + y + 2, y^6 + x*y^5 + x^2 ])
```
```
sage: I = Ideal(I.groebner_basis()); I
Ideal (x - y^47 - y^45 + y^44 - y^43 + y^41 - y^39 - y^38 - y^37 - y^36 + y^35 - y^34 + y^33 - y^32 + y^31 + y^30 + y^28 + y^27 + y^26 + y^25 - y^23 + y^22 + y^21 - y^19 - y^18 - y^16 + y^15 + y^13 + y^12 - y^10 + y^9 + y^8 + y^7 - y^6 + y^4 + y^3 + y^2 + y - 1, y^48 + y^41 - y^40 + y^37 - y^36 - y^33 + y^32 - y^29 + y^28 - y^25 + y^24 + y^2 + y + 1) of Multivariate Polynomial Ring in x, y over Finite Field in w of size 3^3

sage: V = I.variety();
sage: sorted(V, key=str)
[{y: w^2 + 2*w, x: 2*w + 2}, {y: w^2 + 2, x: 2*w}, {y: w^2 + w, x: 2*w + 1}]

sage: [f.subs(v) for f in I.gens() for v in V] # check that all polynomials vanish
[0, 0, 0, 0, 0, 0]
sage: [I.subs(v).is_zero() for v in V] # same test, but nicer syntax
[True, True, True]

However, we only account for solutions in the ground field and not in the algebraic closure:

sage: I.vector_space_dimension()
48

Here we compute the points of intersection of a hyperbola and a circle, in several fields:

sage: K.<x, y> = PolynomialRing(QQ, 2, order='lex')
sage: I = Ideal([x*y - 1, (x-2)^2 + (y-1)^2 - 1])
sage: I = Ideal(I.groebner_basis()); I
Ideal (x + y^3 - 2*y^2 + 4*y - 4, y^4 - 2*y^3 + 4*y^2 - 4*y + 1) of Multivariate Polynomial Ring in x, y over Rational Field

These two curves have one rational intersection:

sage: I.variety()
[({y: 1, x: 1})]

There are two real intersections:

sage: sorted(I.variety(ring=RR), key=str)
[({y: 0.361103080528647, x: 2.76929235423863},
 {y: 1.000000000000000, x: 2.76929235423863})]
sage: I.variety(ring=AA) # py2
[({x: 1, y: 1},
 {x: 2.76929235423863, y: 0.361103080528647})]
sage: I.variety(ring=AA) # py3
[({y: 1, x: 1},
 {y: 0.361103080528647, x: 2.76929235423863})]

and a total of four intersections:

sage: sorted(I.variety(ring=CC), key=str)
[({y: 0.31944845973567... + 1.6331702409152...*I,}...}]
x: 0.11535382288068... - 0.58974280502220...*I},
{y: 0.31944845973567... - 1.633170240915220...*I},
{x: 0.11535382288068... + 0.58974280502220...*I},
{y: 0.36110308052864..., x: 2.7692923542386...},
{y: 1.0000000000000000, x: 1.0000000000000000}

sage: sorted(I.variety(ring=QQbar), key=str)
[[{y: 0.3194484597356763? + 1.633170240915238?*I,
x: 0.11535382288068429? - 0.5897428050222055?*I},
{y: 0.3194484597356763? - 1.633170240915238?*I,
x: 0.11535382288068429? + 0.5897428050222055?*I},
{y: 0.3611030805286474?, x: 2.769292354238632?},
{y: 1, x: 1}]

Computation over floating point numbers may compute only a partial solution, or even none at all. Notice
that x values are missing from the following variety:

sage: R.<x,y> = CC[]
sage: I = ideal([x^2+y^2-1,x*y-1])
sage: sorted(I.variety(), key=str)
verbose 0 (...: multi_polynomial_ideal.py, variety) Warning: computations in
the complex field are inexact; variety may be computed partially or
incorrectly.
verbose 0 (...: multi_polynomial_ideal.py, variety) Warning: falling back to
very slow toy implementation.
[[{y: -0.86602540378443... + 0.500000000000000*I},
{y: -0.86602540378443... - 0.500000000000000*I},
{y: 0.86602540378443... + 0.500000000000000*I},
{y: 0.86602540378443... - 0.500000000000000*I}]

This is due to precision error, which causes the computation of an intermediate Groebner basis to fail.
If the ground field’s characteristic is too large for Singular, we resort to a toy implementation:

sage: R.<x,y> = PolynomialRing(GF(2147483659),order='lex')
sage: I=ideal([x^3-2*y^2,3*x+y^4])
sage: I.variety()
verbose 0 (...: multi_polynomial_ideal.py, groebner_basis) Warning: falling back to
very slow toy implementation.
verbose 0 (...: multi_polynomial_ideal.py, dimension) Warning: falling back to
very slow toy implementation.
verbose 0 (...: multi_polynomial_ideal.py, variety) Warning: falling back to
very slow toy implementation.
[[{y: 0, x: 0}]

The dictionary expressing the variety will be indexed by generators of the polynomial ring after changing
to the target field. But the mapping will also accept generators of the original ring, or even generator names
as strings, when provided as keys:

sage: K.<x,y> = QQ[]
sage: I = ideal([x^2+2*y^2-5,x+y+3])
sage: v = I.variety(AA)[0]; v[x], v[y]
(4.4641016151377557, -7.4641016151377557)
sage: list(v)[0].parent()
Multivariate Polynomial Ring in x, y over Algebraic Real Field

sage: v[x]
4.464101615137755?
sage: v["y"]
-7.464101615137755?

ALGORITHM:
Uses triangular decomposition.

vector_space_dimension()
Return the vector space dimension of the ring modulo this ideal. If the ideal is not zero-dimensional, a
TypeError is raised.

ALGORITHM:
Uses Singular.

EXAMPLES:

sage: R.<u,v> = PolynomialRing(QQ)
sage: g = u^4 + v^4 + u^3 + v^3
sage: I = ideal(g) + ideal(g.gradient())
sage: I.dimension()
0
sage: I.vector_space_dimension()
4

When the ideal is not zero-dimensional, we return infinity:

sage: R.<x,y> = PolynomialRing(QQ)
sage: I = R.ideal(x)
sage: I.dimension()
1
sage: I.vector_space_dimension()
+Infinity

Due to integer overflow, the result is correct only modulo $2^{32}$, see trac ticket #8586:

sage: P.<x,y,z> = PolynomialRing(GF(32003),3)
sage: sage.rings.ideal.FieldIdeal(P).vector_space_dimension()  # known bug
3277216864027

class sage.rings.polynomial.multi_polynomial_ideal.NCPolynomialIdeal(ring, gens, coerce=True, side='left')

Bases: sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr,
sage.rings.noncommutative_ideals.Ideal_nc

Creates a non-commutative polynomial ideal.

INPUT:

- ring - the g-algebra to which this ideal belongs
- gens - the generators of this ideal
- coerce (optional - default True) - generators are coerced into the ring before creating the ideal
- side - optional string, either “left” (default) or “twosided”; defines whether this ideal is left of two-sided.
EXAMPLES:

```python
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H = A.g_algebra({y*x:x*y-z, z*x:x*z+2*x, z*y:y*z-2*y})
sage: H.inject_variables()
Defining x, y, z
sage: I = H.ideal([y^2, x^2, z^2-H.one()],coerce=False) # indirect doctest
sage: I
Left Ideal (y^2, x^2, z^2 - 1) of Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-relations: {z*x: x*z + 2*x, z*y: y*z - 2*y, y*x: x*y - z}
sage: sorted(I.gens(),key=str)
[x^2, y^2, z^2 - 1]
sage: H.ideal([y^2, x^2, z^2-H.one()], side="twosided") # random
Twosided Ideal (y^2, x^2, z^2 - 1) of Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-relations: {z*x: x*z + 2*x, z*y: y*z - 2*y, y*x: x*y - z}
sage: sorted(H.ideal([y^2, x^2, z^2-H.one()], side="twosided").gens(),key=str)
[x^2, y^2, z^2 - 1]
sage: H.ideal([y^2, x^2, z^2-H.one()], side="right")
Traceback (most recent call last):
  ... ValueError: Only left and two-sided ideals are allowed.
```

```python
definitions
elimination_ideal(variables)
Return the elimination ideal of this ideal with respect to the variables given in “variables”.

EXAMPLES:

```python
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H = A.g_algebra({y*x:x*y-z, z*x:x*z+2*x, z*y:y*z-2*y})
sage: H.inject_variables()
Defining x, y, z
sage: I = H.ideal([y^2, x^2, z^2-H.one()],coerce=False)
sage: I.elimination_ideal([x, z])
Left Ideal (y^2) of Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-relations: {...}
sage: J = I.twostd()
sage: J
Twosided Ideal (z^2 - 1, y*z - y, x*z + x, y^2, 2*x*y - z - 1, x^2) of Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-relations: {...}
sage: J.elimination_ideal([x, z])
Twosided Ideal (y^2) of Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-relations: {...}
```

ALGORITHM: Uses Singular's eliminate command

reduce(p)
Reduce an element modulo a Groebner basis for this ideal.

It returns 0 if and only if the element is in this ideal. In any case, this reduction is unique up to monomial orders.

NOTE:
There are left and two-sided ideals. Hence,
EXAMPLES:

```python
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H.<x,y,z> = A.g_algebra({y*x:x*y-z, z*x:x*z+2*x, z*y:y*z-2*y})
sage: I = H.ideal([y^2, x^2, z^2-H.one()],coerce=False, side='twosided')
sage: Q = H.quotient(I); Q
#random
Quotient of Noncommutative Multivariate Polynomial Ring in x, y, z
over Rational Field, nc-relations: {z*x: x*z + 2*x, z*y: y*z - 2*y, y*x: x*y - z} by the ideal (y^2, x^2, z^2 - 1)
sage: Q.2^2 == Q.one() # indirect doctest
True
```

Here, we see that the relation that we just found in the quotient is actually a consequence of the given relations:

```python
sage: H.2^2-H.one() in I.std().gens()
True
```

Here is the corresponding direct test:

```python
sage: I.reduce(z^2)
1
```

`res(length)`

Compute the resolution up to a given length of the ideal.

NOTE:

Only left syzygies can be computed. So, even if the ideal is two-sided, then the resolution is only one-sided. In that case, a warning is printed.

EXAMPLES:

```python
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H = A.g_algebra({y*x:x*y-z, z*x:x*z+2*x, z*y:y*z-2*y})
Defining x, y, z
sage: H.inject_variables()
Defining x, y, z
sage: I = H.ideal([y^2, x^2, z^2-H.one()],coerce=False)
sage: I.res(3)
<Resolution>
```

`std()`

Computes a GB of the ideal. It is two-sided if and only if the ideal is two-sided.

EXAMPLES:

```python
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H = A.g_algebra({y*x:x*y-z, z*x:x*z+2*x, z*y:y*z-2*y})
Defining x, y, z
sage: H.inject_variables()
Defining x, y, z
sage: I = H.ideal([y^2, x^2, z^2-H.one()],coerce=False)
sage: I.std() #random
Left Ideal (z^2 - 1, y*z - y, x*z + x, y^2, 2*x*y - z - 1, x^2) of...
˓
→Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, ...
˓→nc-relations: {z*x: x*z + 2*x, z*y: y*z - 2*y, y*x: x*y - z}
sage: sorted(I.std().gens(),key=str)
[2*x*y - z - 1, x*z + x, x^2, y*z - y, y^2, z^2 - 1]
```
If the ideal is a left ideal, then std returns a left Groebner basis. But if it is a two-sided ideal, then the output of std and twostd() coincide:

```
sage: JL = H.ideal([x^3, y^3, z^3 - 4*z])
sage: JL #random
Left Ideal (x^3, y^3, z^3 - 4*z) of Noncommutative Multivariate Polynomial Ring
˓→ in x, y, z over Rational Field, nc-relations: {z*x: x*z + 2*x, z*y: y*z - 2*y, 
˓→ y*x: x*y - z}
sage: sorted(JL.gens(),key=str)
[x^3, y^3, z^3 - 4*z]
sage: JL.std() #random
Left Ideal (z^3 - 4*z, y*z^2 - 2*y*z, x*z^2 + 2*x*z, 2*x*y*z - z^2 - 2*z, y^3, 
˓→ x^3) of Noncommutative Multivariate Polynomial Ring in x, y, z over Rational
˓→ Field, nc-relations: {z*x: x*z + 2*x, z*y: y*z - 2*y, y*x: x*y - z}
sage: sorted(JL.std().gens(),key=str)
[2*x*y*z - z^2 - 2*z, x*z^2 + 2*x*z, x^3, y*z^2 - 2*y*z, y^3, z^3 - 4*z]
sage: JT = H.ideal([x^3, y^3, z^3 - 4*z], side='twosided')
sage: JT #random
Twosided Ideal (x^3, y^3, z^3 - 4*z) of Noncommutative Multivariate Polynomial
˓→ Ring in x, y, z over Rational Field, nc-relations: {z*x: x*z + 2*x, z*y: y*z - 
˓→ 2*y, y*x: x*y - z}
sage: sorted(JT.gens(),key=str)
[x^3, y^3, z^3 - 4*z]
sage: JT.std() #random
Twosided Ideal (z^3 - 4*z, y*z^2 - 2*y*z, x*z^2 + 2*x*z, y^2*z - 2*y^2, 2*x*y*z 
˓→ - z^2 - 2*z, x^2*z + 2*x^2, y^3, x*y^2 - y*z, x^2*y - x*z - 2*x, x^3) of Noncommutative
˓→ Multivariate Polynomial Ring in x, y, z over Rational Field, nc-relations: {z*x: x*z + 2*x, z*y: y*z - 2*y, y*x: x*y - z}
sage: sorted(JT.std().gens(),key=str)
[2*x*y*z - z^2 - 2*z, x*y^2 - y*z, x*z^2 + 2*x*z, x^2*y - x*z - 2*x, x^2*z + 
˓→ 2*x^2, x^3, y*z^2 - 2*y*z, y^2*z - 2*y^2, y^3, z^3 - 4*z]
sage: JT.std() == JL.twostd()
True
```

ALGORITHM: Uses Singular’s std command

**syzygy_module()**

Computes the first syzygy (i.e., the module of relations of the given generators) of the ideal.

**NOTE:**

Only left syzygies can be computed. So, even if the ideal is two-sided, then the syzygies are only one-sided.

In that case, a warning is printed.

**EXAMPLES:**

```
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H = A.g_algebra({y*x:x*y-y*z, z*x:x*z+2*x, z*y:y*z-2*y})
sage: H.inject_variables()
Defining x, y, z
sage: I = H.ideal([y^2, x^2, z^2-H.one()],coerce=False)
sage: G = vector(I.gens()); G
d...: UserWarning: You are constructing a free module over a noncommutative ring. Sage does not have a concept of left/right and both sided modules, so be careful. It's also not guaranteed that all multiplications are
```
done from the right side.
d...: UserWarning: You are constructing a free module
over a noncommutative ring. Sage does not have a concept
of left/right and both sided modules, so be careful.
It's also not guaranteed that all multiplications are
done from the right side.
(y^2, x^2, z^2 - 1)
sage: M = I.syzygy_module(); M
[ [-z^2 - 8*z - 15]
  [ -y^2 ]
  [ 0 ]
  [ z^2 + 8*z - 15]
  [ x^2 ]
[ [ 2*z + 15*x^2 -y^2]
  [ 2*z^2 + 8*y^2*z - 15*y^2]
  [-4*x*y*z + 2*z^2 + 2*z]
  [x^2*y^2*z + 9*x^2*y^2*z - 6*x^2*z^3 + 20*x^2*y - 72*x^2*z^2]
  [282*x^2 - 360*x]
  [y^3*z^2 + 7*y^3*z - 12*y^3]
  [6*y^3*z^2]
  [x^3*z^2 + 7*x^3 + 12*x^3 -x^2*y^2*z + 9*x^2*y^2*z - 4*y^2*z^3 - 20*x*y^2 +]
  [52*y^2*z^2 - 224*y^2*z + 320*y]
  [-6*x^2*z^2]
  [x^2*y^2*z^2 + 4*x^2*y^2*z - 8*x^2*y^2*z - 48*x^2*y^2*z + 12*x^2*z^3 - 64*x*y + 108*z^2 +]
  [312*z^2 + 288]
  [-y^4*z + 4*y^4]
  [0]
[ [ 2*x^3*y*z + 8*x^3*y + 9*x^2 -x^2*y^2*z + 4*x^2*y^2*z - 4*y^2*z^2 + 32*x^2*y^2*z - 6*z^3 - 64*x*y^2 +]
  [1656*x^2 - 2052*x]
  [-x*y^4*z + 4*x*y^4]
  [8*y^3*z^2 + 62*y^3*z - 114*y^3]
  [48*y^3*z^2 - 36*y^3]
]
sage: M*G
(0, 0, 0, 0, 0, 0, 0, 0, 0)

ALGORITHM: Uses Singular's syz command
twostd()
Computes a two-sided GB of the ideal (even if it is a left ideal).
EXAMPLES:
sage: A.<x,y,z> = FreeAlgebra(QQ, 3)
sage: H = A.g_algebra({y*x:x*y-z, z*x:x*z+2*x, z*y:y*z-2*y})
sage: H.inject_variables()
Defining x, y, z
sage: I = H.ideal([y^2, x^2, z^2-H.one()],coerce=False)
sage: I.twostd()
Twosided Ideal (z^2 - 1, y*z - y, x*z + x, y^2, 2*x*y - z - 1, x^2) of Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field...
sage: sorted(I.twostd().gens(),key=str)
[2*x*y - z - 1, x*z + x, x^2, y*z - y, y^2, z^2 - 1]

ALGORITHM: Uses Singular's twostd command

class sage.rings.polynomial.multi_polynomial_ideal.RequireField(f)

Bases: sage.misc.method_decorator.MethodDecorator

Decorator which throws an exception if a computation over a coefficient ring which is not a field is attempted.

Note: This decorator is used automatically internally so the user does not need to use it manually.

sage.rings.polynomial.multi_polynomial_ideal.is_MPolynomialIdeal(x)

Return True if the provided argument x is an ideal in the multivariate polynomial ring.

INPUT:
• x - an arbitrary object

EXAMPLES:

sage: from sage.rings.polynomial.multi_polynomial_ideal import is_MPolynomialIdeal
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: I = [x + 2*y + 2*z - 1, x^2 + 2*y^2 + 2*z^2 - x, 2*x*y + 2*y*z - y]

Sage distinguishes between a list of generators for an ideal and the ideal itself. This distinction is inconsistent with Singular but matches Magma’s behavior.

sage: is_MPolynomialIdeal(I)
False

sage: I = Ideal(I)
sage: is_MPolynomialIdeal(I)
True

sage.rings.polynomial.multi_polynomial_ideal.require_field

alias of sage.rings.polynomial.multi_polynomial_ideal.RequireField
3.1.7 Polynomial Sequences

We call a finite list of polynomials a Polynomial Sequence.

Polynomial sequences in Sage can optionally be viewed as consisting of various parts or sub-sequences. These kind of polynomial sequences which naturally split into parts arise naturally for example in algebraic cryptanalysis of symmetric cryptographic primitives. The most prominent examples of these systems are: the small scale variants of the AES [CMR2005] (cf. sage.crypto.mq.sr.SR()) and Flurry/Curry [BPW2006]. By default, a polynomial sequence has exactly one part.

AUTHORS:

- Martin Albrecht (2007ff): initial version
- Martin Albrecht (2009): refactoring, clean-up, new functions
- Martin Albrecht (2011): refactoring, moved to sage.rings.polynomial
- Alex Raichev (2011-06): added algebraic_dependence()
- Charles Bouillaguet (2013-1): added solve()

EXAMPLES:

As an example consider a small scale variant of the AES:

```python
sage: sr = mq.SR(2,1,2,4,gf2=True,polybori=True)
sage: sr
SR(2,1,2,4)
```

We can construct a polynomial sequence for a random plaintext-ciphertext pair and study it:

```python
sage: set_random_seed(1)
sage: F, s = sr.polynomial_system()
sage: F
Polynomial Sequence with 112 Polynomials in 64 Variables
sage: r2 = F.part(2); r2
(w200 + k100 + x100 + x102 + x103,
 w201 + k101 + x100 + x101 + x103 + 1,
 w202 + k102 + x100 + x101 + x102 + 1,
 w203 + k103 + x101 + x102 + x103,
 w210 + k110 + x110 + x112 + x113,
 w211 + k111 + x110 + x111 + x113 + 1,
 w212 + k112 + x110 + x111 + x112 + 1,
 w213 + k113 + x111 + x112 + x113,
 x100*w100 + x100*w103 + x101*w102 + x102*w101 + x103*w100,
 x100*w100 + x100*w101 + x101*w100 + x101*w103 + x102*w102 + x103*w101,
 x100*w101 + x100*w102 + x101*w100 + x101*w102 + x102*w100 + x102*w103 + x103*w102,
 x100*w100 + x100*w102 + x100*w103 + x101*w100 + x101*w101 + x102*w102 + x103*w100 +
 x100*w101 + x100*w102 + x101*w101 + x101*w103 + x102*w100 + x102*w103 + x103*w102,
 x100*w100 + x100*w103 + x101*w102 + x102*w101 + x103*w100 +
 ...+w100,
 x100*w101 + x100*w102 + x101*w100 + x101*w102 + x102*w100 + x102*w103 + x103*w101 +
 x100*w100 + x100*w102 + x101*w100 + x101*w102 + x102*w100 + x102*w101 +
 ...+w100,
```

(continues on next page)
We separate the system in independent subsystems:

```python
sage: C = Sequence(r2).connected_components(); C
[[w213 + k113 + x111 + x112 + x113,
  w212 + k112 + x110 + x111 + x112 + 1,
  w211 + k111 + x110 + x111 + x113 + 1,
  w210 + k110 + x110 + x112 + x113,
  x110*w112 + x111*w111 + x112*w110 + x113*w113 + 1,
  x110*w112 + x111*w110 + x111*w111 + x112*w112 + x112*w113 + x113*w110 + x113*w112 +
  w111,
  x110*w111 + x111*w110 + x111*w111 + x112*w112 + x112*w113 + x113*w113,
  x110*w111 + x110*w113 + x111*w111 + x111*w112 + x112*w110 + x112*w112 + x113*w112 +
  w110,
  x110*w110 + x110*w111 + x110*w113 + x111*w111 + x111*w113 + x112*w112 + x112*w112 +
  x113*w112 + x112,
  x110*w110 + x110*w111 + x111*w110 + x111*w112 + x112*w111 + x113*w113 + x113*w113 +
  w113,
  x110*w112 + x111*w110 + x111*w111 + x112*w110 + x112*w112 + x112*w113 + x113*w113 +
  w113]],
```

(continues on next page)
w201 + k101 + x100 + x101 + x103 + 1,
w200 + k100 + x100 + x102 + x103,
w100*w102 + x101*w101 + x102*w100 + x103*w103 + 1,
w100*w102 + x101*w100 + x101*w103 + x102*w101 + x103*w100 + x103*w102 + w101,
w100*w101 + x101*w100 + x101*w102 + x102*w100 + x103*w101 + x103*w103 + w103,
w100*w101 + x100*w103 + x101*w101 + x102*w102 + x102*w100 + x102*w103 + x103*w101 + x101,
w100*w101 + x100*w102 + x101*w100 + x101*w103 + x102*w101 + x103*w103 + x103 + x103, x100*w101
  + x100*w102 + x101*w100 + x101*w101 + x102*w100 + x102*w103 + x103*w102,
x100*w100 + x100*w102 + x101*w101 + x103*w100,
x100*w100 + x100*w102 + x101*w100 + x101*w102 + x102*w100 + x102*w103 + x103*w102 + x102,
x100*w100 + x100*w102 + x101*w100 + x101*w103 + x102*w102 + x103*w100 + x100 + x100,
x100*w100 + x100*w102 + x101*w100 + x101*w103 + x102*w102 + x103*w100 + x100, x100
  + x100*w101 + x100*w102 + x101*w101 + x102*w102 + x103*w100 + x100 + x100,
x100*w100 + x100*w102 + x101*w100 + x101*w103 + x102*w102 + x103*w100 + x100 + x100,
x100*w100 + x100*w102 + x101*w101 + x102*w100 + x102*w101 + x102*w103 + x103*w100 + x100 + x100,
x100*w100 + x100*w102 + x101*w101 + x102*w100 + x102*w101 + x102*w103 + x103*w100 + x100 + x100
  + x103*w101 + w102]

Sage: C[0].groebner_basis()
Polynomial Sequence with 30 Polynomials in 16 Variables

and compute the coefficient matrix:

Sage: A,v = Sequence(r2).coefficient_matrix()
Sage: A.rank()
32

Using these building blocks we can implement a simple XL algorithm easily:

Sage: sr = mq.SR(1,1,4, gf2=True, polybori=True, order='lex')
Sage: F,s = sr.polynomial_system()
Sage: monomials = [a*b for a in F.variables() for b in F.variables() if a<b]
Sage: len(monomials)
190
Sage: F2 = Sequence(map(mul, cartesian_product_iterator((monomials, F))))
Sage: A,v = F2.coefficient_matrix(sparse=False)
Sage: A.echelonize()
Sage: A
6840 x 4474 dense matrix over Finite Field of size 2 (use the '.str()' method to see the...
  entries)
Sage: A.rank()
4056
Sage: A[4055]*v
(k001*k003)

Note: In many other computer algebra systems (cf. Singular) this class would be called Ideal but an ideal is a very distinct object from its generators and thus this is not an ideal in Sage.
Classes

`sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence(arg1, arg2=None, immutable=False, cr=False, cr_str=None)`

Construct a new polynomial sequence object.

**INPUT:**

- `arg1` - a multivariate polynomial ring, an ideal or a matrix
- `arg2` - an iterable object of parts or polynomials (default: `None`)
  - `immutable` - if True the sequence is immutable (default: `False`)
  - `cr` - print a line break after each element (default: `False`)
  - `cr_str` - print a line break after each element if ‘str’ is called (default: `None`)

**EXAMPLES:**

```
sage: P.<a,b,c,d> = PolynomialRing(GF(127),4)
sage: I = sage.rings.ideal.Katsura(P)
```

If a list of tuples is provided, those form the parts:

```
sage: F = Sequence([I.gens(),I.gens()], I.ring()); F # indirect doctest
[a + 2*b + 2*c + 2*d - 1,
a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a,
2*a*b + 2*b*c + 2*c*d - b,
b^2 + 2*a*c + 2*b*d - c,
a + 2*b + 2*c + 2*d - 1,
a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a,
2*a*b + 2*b*c + 2*c*d - b,
b^2 + 2*a*c + 2*b*d - c]
sage: F.nparts()
2
```

If an ideal is provided, the generators are used:

```
sage: Sequence(I)
[a + 2*b + 2*c + 2*d - 1,
a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a,
2*a*b + 2*b*c + 2*c*d - b,
b^2 + 2*a*c + 2*b*d - c]
```

If a list of polynomials is provided, the system has only one part:

```
sage: F = Sequence(I.gens(), I.ring()); F
[a + 2*b + 2*c + 2*d - 1,
a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a,
2*a*b + 2*b*c + 2*c*d - b,
b^2 + 2*a*c + 2*b*d - c]
sage: F.nparts()
1
```

We test that the ring is inferred correctly:
Construct a new system of multivariate polynomials.

**INPUT:**
- **part** - a list of lists with polynomials
- **ring** - a multivariate polynomial ring
- **immutable** - if True the sequence is immutable (default: False)
- **cr** - print a line break after each element (default: False)
- **cr_str** - print a line break after each element if ‘str’ is called (default: None)

**EXAMPLES:**

```sage```
sage: P.<a,b,c,d> = PolynomialRing(GF(127),4)
sage: I = sage.rings.ideal.Katsura(P)
sage: Sequence(I.gens(), I.ring())
[a + 2*b + 2*c + 2*d - 1, a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a, 2*a*b + 2*b*c + 2*c*d - b, b^2 + 2*a*c + 2*b*d - c]
```

If an ideal is provided, the generators are used:

```sage```
sage: Sequence(I)
[a + 2*b + 2*c + 2*d - 1, a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a, 2*a*b + 2*b*c + 2*c*d - b, b^2 + 2*a*c + 2*b*d - c]
```

If a list of polynomials is provided, the system has only one part:

```sage```
sage: Sequence(I.gens(), I.ring())
[a + 2*b + 2*c + 2*d - 1, a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a, 2*a*b + 2*b*c + 2*c*d - b, b^2 + 2*a*c + 2*b*d - c]
```

**algebraic_dependence()**

Returns the ideal of annihilating polynomials for the polynomials in self, if those polynomials are algebraically dependent. Otherwise, returns the zero ideal.

**OUTPUT:**

If the polynomials \(f_1, \ldots, f_r\) in self are algebraically dependent, then the output is the ideal \(\{F \in K[T_1, \ldots, T_r] : F(f_1, \ldots, f_r) = 0\}\) of annihilating polynomials of \(f_1, \ldots, f_r\). Here \(K\) is the coefficient
ring of polynomial ring of \( f_1, \ldots, f_r \) and \( T_1, \ldots, T_r \) are new indeterminates. If \( f_1, \ldots, f_r \) are algebraically independent, then the output is the zero ideal in \( K[T_1, \ldots, T_r] \).

EXAMPLES:

```python
sage: R.<x,y> = PolynomialRing(QQ)
sage: S = Sequence([x, x*y])
sage: I = S.algebraic_dependence(); I
Ideal (0) of Multivariate Polynomial Ring in T0, T1 over Rational Field
```

```python
sage: R.<x,y> = PolynomialRing(QQ)
sage: S = Sequence([x, (x^2 + y^2 - 1)^2, x*y - 2])
sage: I = S.algebraic_dependence(); I
Ideal (16 + 32*T2 - 8*T0^2 + 24*T2^2 - 8*T0^2*T2 + 8*T2^3 + 9*T0^4 - 2*T0^2*T2^→ 2 + T2^4 - T0^4*T1 + 8*T0^4*T2 - 2*T0^6 + 2*T0^4*T2^2 + T0^8) of Multivariate Polynomial Ring in T0, T1, T2 over Rational Field
```

```python
sage: [F(S) for F in I.gens()]
[0]
```

Note: This function’s code also works for sequences of polynomials from a univariate polynomial ring, but I don’t know where in the Sage codebase to put it to use it to that effect.

AUTHORS:

- Alex Raichev (2011-06-22)

coefficient_matrix(sparse=True)

Return tuple \((A, v)\) where \(A\) is the coefficient matrix of this system and \(v\) the matching monomial vector. Thus value of \(A[i,j]\) corresponds the coefficient of the monomial \(v[j]\) in the \(i\)-th polynomial in this system.

Monomials are order w.r.t. the term ordering of `self.ring()` in reverse order, i.e. such that the smallest entry comes last.

INPUT:

- `sparse` - construct a sparse matrix (default: True)

EXAMPLES:

```python
sage: P.<a,b,c,d> = PolynomialRing(GF(127),4)
sage: I = sage.rings.ideal.Katsura(P)
sage: I.gens()
[a + 2*b + 2*c + 2*d - 1,
 a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a,
 (continues on next page)```
\[ 2^a b + 2^b c + 2^c d - b, \]
\[ b^2 + 2^a c + 2^b d - c \]

\texttt{sage: F = Sequence(I)}
\texttt{sage: A,v = F.coefficient_matrix()}
\texttt{sage: A}
\begin{bmatrix}
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 2 & 2 & 126 \\
 1 & 0 & 2 & 0 & 0 & 2 & 0 & 0 & 2 & 126 & 0 & 0 & 0 \\
 0 & 2 & 0 & 0 & 2 & 0 & 0 & 2 & 0 & 0 & 126 & 0 & 0 \\
 0 & 0 & 1 & 2 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 126 & 0 & 0
\end{bmatrix}

\texttt{sage: v}
\begin{bmatrix}
 a^2 \\
 a b \\
 b^2 \\
 a c \\
 b c \\
 c^2 \\
 b d \\
 c d \\
 d^2 \\
 a \\
 b \\
 c \\
 d \\
 1
\end{bmatrix}

\texttt{sage: A*v}
\begin{bmatrix}
a + 2^b + 2^c + 2^d - 1 \\
 a^2 + 2^b^2 + 2^c^2 + 2^d^2 - a \\
 2^a^2 b + 2^b a^2 c + 2^c b c d - b \\
 b^2 + 2^a^2 c + 2^b d - c
\end{bmatrix}

\texttt{connected_components()}

Split the polynomial system in systems which do not share any variables.

EXAMPLES:

As an example consider one part of AES, which naturally splits into four subsystems which are independent:

\texttt{sage: sr = mq.SR(2,4,4,8,gf2=True,polybori=True)}
\texttt{sage: F,s = sr.polynomial_system()}
\texttt{sage: Fz = Sequence(F.part(2))}
\texttt{sage: Fz.connected_components()}

\begin{enumerate}
\item Polynomial Sequence with 128 Polynomials in 128 Variables,
\item Polynomial Sequence with 128 Polynomials in 128 Variables,
\item Polynomial Sequence with 128 Polynomials in 128 Variables,
\item Polynomial Sequence with 128 Polynomials in 128 Variables
\end{enumerate}

\texttt{connection_graph()}

Return the graph which has the variables of this system as vertices and edges between two variables if they appear in the same polynomial.

EXAMPLES:
sage: B.<x,y,z> = BooleanPolynomialRing()
sage: F = Sequence([x*y + y + 1, z + 1])
sage: F.connection_graph()
Graph on 3 vertices

`groebner_basis(*args, **kwargs)`

Compute and return a Groebner basis for the ideal spanned by the polynomials in this system.

**INPUT:**

- `args` - list of arguments passed to `MPolynomialIdeal.groebner_basis` call
- `kwargs` - dictionary of arguments passed to `MPolynomialIdeal.groebner_basis` call

**EXAMPLES:**

```python
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F, s = sr.polynomial_system()
sage: gb = F.groebner_basis()
sage: Ideal(gb).basis_is_groebner()
True
```

`ideal()`

Return ideal spanned by the elements of this system.

**EXAMPLES:**

```python
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F, s = sr.polynomial_system()
sage: P = F.ring()
sage: I = F.ideal()
sage: I.elimination_ideal(P('k002 + (a^3 + a + 1)*k003 + (a^2 + a + 1),
 k001 + (a^3)*k003, k000 + (a)*k003 + (a^2),
 k103 + k003 + (a^2 + a + 1),
 k102 + (a^3 + a + 1)*k003 + (a + 1),
 k101 + (a^3)*k003 + (a^2 + a + 1),
 k100 + (a)*k003 + (a),
 k003^2 + (a)*k003 + (a^2))
of Multivariate Polynomial Ring in k100, k101, k102, k103, x100, x101, x102, x103
 _ -> Finite Field in a of size 2^4
```

`is_groebner(singular=Singular)`

Returns `True` if the generators of this ideal (`self.gens()`) form a Groebner basis.

Let $I$ be the set of generators of this ideal. The check is performed by trying to lift $Syz(LM(I))$ to $Syz(I)$ as $I$ forms a Groebner basis if and only if for every element $S$ in $Syz(LM(I))$:

$$S * G = \sum_{i=0}^{m} h_i g_i \succ 0.$$ 

**EXAMPLES:**

```pythonsage: R.<a,b,c,d,e,f,g,h,i,j> = PolynomialRing(GF(127),10)
sage: I = sage.rings.ideal.Cyclic(R,4)
```
sage: I.basis.is_groebner()
False
sage: I2 = Ideal(I.groebner_basis())
sage: I2.basis.is_groebner()
True

**maximal_degree**

Return the maximal degree of any polynomial in this sequence.

**EXAMPLES:**

```sage
sage: P.<x,y,z> = PolynomialRing(GF(7))
sage: F = Sequence([x*y + x, x])
sage: F.maximal_degree()
2
sage: P.<x,y,z> = PolynomialRing(GF(7))
sage: F = Sequence([], universe=P)
sage: F.maximal_degree()
-1
```

**monomials**

Return an unordered tuple of monomials in this polynomial system.

**EXAMPLES:**

```sage
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system()
sage: len(F.monomials())
49
```

**nmonomials**

Return the number of monomials present in this system.

**EXAMPLES:**

```sage
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system()
sage: F.nmonomials()
49
```

**nparts**

Return number of parts of this system.

**EXAMPLES:**

```sage
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system()
sage: F.nparts()
4
```

**nvariables**

Return number of variables present in this system.

**EXAMPLES:**

```sage
```
part(i)
Return i-th part of this system.

EXAMPLES:

```
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system()
sage: R0 = F.part(1)
sage: R0
(k000^2 + k001, k001^2 + k002, k002^2 + k003, k003^2 + k000)
```

parts()
Return a tuple of parts of this system.

EXAMPLES:

```
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system()
sage: l = F.parts()
sage: len(l)
4
```

reduced()
If this sequence is \((f_1, ..., f_n)\) then this method returns \((g_1, ..., g_s)\) such that:

- \((f_1, ..., f_n) = (g_1, ..., g_s)\)
- \(LT(g_i)! = LT(g_j)\) for all \(i! = j\)
- \(LT(g_i)\) does not divide \(m\) for all monomials \(m\) of \(\{g_1, ..., g_i-1, g_i+1, ..., g_s\}\)
- \(LC(g_i) == 1\) for all \(i\) if the coefficient ring is a field.

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ)
sage: F = Sequence([z*x+y^3,z+y^3,z+x*y])
sage: F.reduced()
[y^3 + z, x*y + z, x*z - z]
```

Note that tail reduction for local orderings is not well-defined:

```
sage: R.<x,y,z> = PolynomialRing(QQ,order='negdegrevlex')
sage: F = Sequence([z*x+y^3,z+y^3,z+x*y])
sage: F.reduced()
[z + x*y, x*y - y^3, x^2*y - y^3]
```

A fixed error with nonstandard base fields:

```
sage: R.<t>=QQ['t']
sage: K.<x,y>=R.fraction_field()['x,y']
sage: I=t^x*K
```

(continues on next page)
The interreduced basis of 0 is 0:

```
sage: P.<x,y,z> = GF(2)[]
sage: Sequence([P(0)]).reduced()
[0]
```

Leading coefficients are reduced to 1:

```
sage: P.<x,y> = QQ[]
sage: Sequence([2*x,y]).reduced()
[x, y]
sage: P.<x,y> = CC[]
sage: Sequence([2*x,y]).reduced()
[x, y]
```

**ALGORITHM:**

Uses Singular’s interred command or `sage.rings.polynomial.toy_buchberger.inter_reduction()` if conversion to Singular fails.

**ring()**

Return the polynomial ring all elements live in.

**EXAMPLES:**

```
sage: sr = mq.SR(allow_zero_inversions=True,gf2=True,order='block')
sage: F,s = sr.polynomial_system(); F
Polynomial Sequence with 40 Polynomials in 20 Variables
```

**subs(**args, **kwargs)**

Substitute variables for every polynomial in this system and return a new system. See `MPolynomial.subs` for calling convention.

**INPUT:**

- `args` - arguments to be passed to `MPolynomial.subs`
- `kwargs` - keyword arguments to be passed to `MPolynomial.subs`

**EXAMPLES:**

```
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system(); F
Polynomial Sequence with 40 Polynomials in 20 Variables
```
sage: F = F.subs(s); F
Polynomial Sequence with 40 Polynomials in 16 Variables

universe()

Return the polynomial ring all elements live in.

EXAMPLES:

sage: sr = mq.SR(allow_zero_inversions=True,gf2=True,order='block')
sage: F,s = sr.polynomial_system()
sage: print(F.ring().repr_long())
Polynomial Ring
  Base Ring : Finite Field of size 2
  Size : 20 Variables
  Block  0 : Ordering : deglex
        Names : k100, k101, k102, k103, x100, x101, x102, x103, w100,...
          w101, w102, w103, s000, s001, s002, s003
  Block  1 : Ordering : deglex
        Names : k000, k001, k002, k003

variables()

Return all variables present in this system. This tuple may or may not be equal to the generators of the ring of this system.

EXAMPLES:

sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system()
sage: F.variables()[:10]
(k003, k002, k001, k000, s003, s002, s001, s000, w103, w102)

class sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_gf2(parts, ring, immutable=False, cr=False, cr_str=None)

Bases: sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic

Polynomial Sequences over \( \mathbb{F}_2 \).

eliminate_linear_variables(maxlength=\( +\infty \), skip=None, return_reductors=False, use_polybori=False)

Return a new system where linear leading variables are eliminated if the tail of the polynomial has length at most maxlength.

INPUT:

- `maxlength` - an optional upper bound on the number of monomials by which a variable is replaced. If `maxlength=\( +\infty \)` then no condition is checked. (default: \( +\infty \)).
- `skip` - an optional callable to skip eliminations. It must accept two parameters and return either `True` or `False`. The two parameters are the leading term and the tail of a polynomial (default: `None`).
- `return_reductors` - if `True` the list of polynomials with linear leading terms which were used for reduction is also returned (default: `False`).
- `use_polybori` - if `True` then `polybori.ll.eliminate` is called. While this is typically faster what is implemented here, it is less flexible (skip’ is not supported) and may increase the degree (default: `'False'`)
When `return_reductors==True`, then a pair of sequences of boolean polynomials are returned, along with the promises that:

1. The union of the two sequences spans the same boolean ideal as the argument of the method.
2. The second sequence only contains linear polynomials, and it forms a reduced Groebner basis (they all have pairwise distinct leading variables, and the leading variable of a polynomial does not occur anywhere in other polynomials).
3. The leading variables of the second sequence do not occur anywhere in the first sequence (these variables have been eliminated).

When `return_reductors==False`, only the first sequence is returned.

**EXAMPLES:**

```python
sage: B.<a,b,c,d> = BooleanPolynomialRing()
sage: F = Sequence([c + d + b + 1, a + c + d, a*b + c, b*c*d + c])
sage: F.eliminate_linear_variables()  # everything vanishes
[]
sage: F.eliminate_linear_variables(maxlength=2)
[b + c + d + 1, b*c + b*d + c, b*c*d + c]
sage: F.eliminate_linear_variables(skip=lambda lm,tail: str(lm)=='a')
[a + c + d, a*c + a*d + a + c, c*d + c]
```

The list of reductors can be requested by setting 'return_reductors' to True:

```python
sage: B.<a,b,c,d> = BooleanPolynomialRing()
sage: F = Sequence([a + b + d, a + b + c])
sage: F,R = F.eliminate_linear_variables(return_reductors=True)
sage: F
[]
sage: R
[a + b + d, c + d]
```

If the input system is detected to be inconsistent then [1] is returned and the list of reductors is empty:

```python
sage: R.<x,y,z> = BooleanPolynomialRing()
sage: S = Sequence([x*y*z+x*y+z*x*z, x+y+z+1, x+y+z])
sage: S.eliminate_linear_variables()
[1]
sage: R.<x,y,z> = BooleanPolynomialRing()
sage: S = Sequence([x*y*z+x*y+z*x*z, x+y+z+1, x+y+z])
sage: S.eliminate_linear_variables(return_reductors=True)
([1], [])
```

**Note:** This is called “massaging” in [BCJ2007].

---

3.1. Multivariate Polynomials and Polynomial Rings
• \(LT(g_i)\) does not divide \(m\) for all monomials \(m\) of \(g_1, \ldots, g_i-1, g_i+1, \ldots, g_s\).

EXAMPLES:

```python
sage: sr = mq.SR(1, 1, 1, 4, gf2=True, polybori=True)
sage: F, s = sr.polynomial_system()
sage: F.reduced()
\[k100 + 1, k101 + k001 + 1, k102, k103 + 1, \ldots, s002, s003 + k001 + 1, k000 + \alpha \rightarrow 1, k002 + 1, k003 + 1\]
```

`solve(algorithm=’polybori’, n=1, eliminate_linear_variables=True, verbose=False, **kwds)`

Find solutions of this boolean polynomial system.

This function provides a unified interface to several algorithms dedicated to solving systems of boolean equations. Depending on the particular nature of the system, some might be much faster than some others.

INPUT:

• `self` - a sequence of boolean polynomials
• `algorithm` - the method to use. Possible values are `polybori`, `sat` and `exhaustive_search`. (default: `polybori`, since it is always available)
• `n` - number of solutions to return. If \(n == +\infty\) then all solutions are returned. If \(n < \infty\) then \(n\) solutions are returned if the equations have at least \(n\) solutions. Otherwise, all the solutions are returned. (default: 1)
• `eliminate_linear_variables` - whether to eliminate variables that appear linearly. This reduces the number of variables (makes solving faster a priori), but is likely to make the equations denser (may make solving slower depending on the method).
• `verbose` - whether to display progress and (potentially) useful information while the computation runs. (default: False)

EXAMPLES:

Without argument, a single arbitrary solution is returned:

```python
sage: from sage.doctest.fixtures import reproducible_repr
sage: R.<x,y,z> = BooleanPolynomialRing()
sage: S = Sequence([x*y+z, y*z+x, x+y+z+1])
sage: sol = S.solve()
sage: print(reproducible_repr(sol))
\[{x: 0, y: 1, z: 0}\]

We check that it is actually a solution:

```python
sage: S.subs(sol[0])
\[0, 0, 0\]
```

We obtain all solutions:

```python
sage: sols = S.solve(n=Infinity)
sage: print(reproducible_repr(sols))
\[{x: 0, y: 1, z: 0}, {x: 1, y: 1, z: 1}\]
```

We can force the use of exhaustive search if the optional package FES is present:

```python
```

Chapter 3. Multivariate Polynomials
sage: sol = S.solve(algorithm='exhaustive_search') # optional - FES
sage: print(reproducible_repr(sol)) # optional - FES
[{x: 1, y: 1, z: 1}]
sage: S.subs(sol[0])
[0, 0, 0]

And we may use SAT-solvers if they are available:

sage: sol = S.solve(algorithm='sat') # optional - cryptominisat
sage: print(reproducible_repr(sol)) # optional - cryptominisat
[{x: 0, y: 1, z: 0}]
sage: S.subs(sol[0])
[0, 0, 0]

class sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_gf2e(parts, ring, immutable=False, cr=False, cr_str=None)

Bases: sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic

PolynomialSequence over $\mathbb{F}_2$, i.e extensions over $\mathbb{F}_2$.

weil_restriction()

Project this polynomial system to $\mathbb{F}_2$.

That is, compute the Weil restriction of scalars for the variety corresponding to this polynomial system and express it as a polynomial system over $\mathbb{F}_2$.

EXAMPLES:

sage: k.<a> = GF(2^2)
sage: P.<x,y> = PolynomialRing(k,2)
sage: a = P.base_ring().gen()
sage: F = Sequence([x*y + 1, a*x + 1], P)
sage: F2 = F.weil_restriction()
sage: F2
[x0*y0 + x1*y1 + 1, x1*y0 + x0*y1 + x1*y1, x1 + 1, x0 + x1, x0^2 + x0, x1^2 + x1, y0^2 + y0, y1^2 + y1]

Another bigger example for a small scale AES:

sage: sr = mq.SR(1,1,1,4,gf2=False)
sage: F,s = sr.polynomial_system(); F
Polynomial Sequence with 40 Polynomials in 20 Variables
sage: F2 = F.weil_restriction(); F2
Polynomial Sequence with 240 Polynomials in 80 Variables

sage.rings.polynomial.multi_polynomial_sequence.is_PolynomialSequence(F)

Return True if F is a PolynomialSequence.

INPUT:

- F - anything

EXAMPLES:
3.1.8 Multivariate Polynomials via libSINGULAR

This module implements specialized and optimized implementations for multivariate polynomials over many coefficient rings, via a shared library interface to SINGULAR. In particular, the following coefficient rings are supported by this implementation:

- the rational numbers $\mathbb{Q}$,
- the ring of integers $\mathbb{Z}$,
- $\mathbb{Z}/n\mathbb{Z}$ for any integer $n$,
- finite fields $\mathbb{F}_{p^n}$ for $p$ prime and $n > 0$,
- and absolute number fields $\mathbb{Q}(a)$.

EXAMPLES:

We show how to construct various multivariate polynomial rings:

```python
sage: P.<x,y> = PolynomialRing(QQ)
sage: I = [[x^2 + y^2], [x^2 - y^2]]
sage: F = Sequence(I, P); F
[x^2 + y^2, x^2 - y^2]
```

```python
sage: from sage.rings.polynomial.multi_polynomial_sequence import is_PolynomialSequence
sage: is_PolynomialSequence(F)
True
```

```python
sage: P.<x,y,z> = QQ[]
sage: P
Multivariate Polynomial Ring in x, y, z over Rational Field
sage: f = 27/113 * x^2 + y*z + 1/2; f
27/113*x^2 + y*z + 1/2
sage: P.term_order()
Degree reverse lexicographic term order
```

```python
sage: P = PolynomialRing(GF(127),3,names='abc', order='lex')
sage: P
Multivariate Polynomial Ring in a, b, c over Finite Field of size 127
sage: a,b,c = P.gens()
sage: f = 57*a^2*b + 43*c + 1; f
57*a^2*b + 43*c + 1
sage: P.term_order()
Lexicographic term order
```

```python
sage: z = QQ['z'].0
sage: K.<s> = NumberField(z^2 - 2)
sage: P.<x,y> = PolynomialRing(K, 2)
sage: 1/2*s*x^2 + 3/4*s
```

(continues on next page)
(1/2*s)*x^2 + (3/4*s)

**sage:** P.<x,y,z> = ZZ[]; P
Multivariate Polynomial Ring in x, y, z over Integer Ring

**sage:** P.<x,y,z> = Zmod(2^10)[]; P
Multivariate Polynomial Ring in x, y, z over Ring of integers modulo 1024

**sage:** P.<x,y,z> = Zmod(3^10)[]; P
Multivariate Polynomial Ring in x, y, z over Ring of integers modulo 59049

**sage:** P.<x,y,z> = Zmod(2^100)[]; P
Multivariate Polynomial Ring in x, y, z over Ring of integers modulo 126765060022822829401496703205376

**sage:** P.<x,y,z> = Zmod(2521352)[]; P
Multivariate Polynomial Ring in x, y, z over Ring of integers modulo 2521352

**sage:** type(P)
<type 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomialRing_libsingular'>

**sage:** P.<x,y,z> = Zmod(25213521351515232)[]; P
Multivariate Polynomial Ring in x, y, z over Ring of integers modulo 25213521351515232

**sage:** type(P)
<class 'sage.rings.polynomial.multi_polynomial_ring.MPolynomialRing_polydict_with_category'>

We construct the Frobenius morphism on $F_5[x, y, z]$ over $F_5$:

**sage:** R.<x,y,z> = PolynomialRing(GF(5), 3)
**sage:** frob = R.hom([x^5, y^5, z^5])
**sage:** frob(x^2 + 2*y - z^4)
-z^20 + x^10 + 2*y^5
**sage:** frob((x + 2*y)^3)
x^15 + x^10*y^5 + 2*x^5*y^10 - 2*y^15
**sage:** (x^5 + 2*y^5)^3
x^15 + x^10*y^5 + 2*x^5*y^10 - 2*y^15

We make a polynomial ring in one variable over a polynomial ring in two variables:

**sage:** R.<x, y> = PolynomialRing(QQ, 2)
**sage:** S.<t> = PowerSeriesRing(R)
**sage:** t*(x+y)
(x + y)*t

**Todo:** Implement Real, Complex coefficient rings via libSINGULAR

**AUTHORS:**
- Martin Albrecht (2007-01): initial implementation
- Joel Mohler (2008-01): misc improvements, polishing
- Martin Albrecht (2008-08): added $Q(\alpha)$ and $Z$ support

3.1. Multivariate Polynomials and Polynomial Rings
• Simon King (2009-04): improved coercion
• Martin Albrecht (2009-05): added \( \mathbb{Z}/n\mathbb{Z} \) support, refactoring
• Martin Albrecht (2009-06): refactored the code to allow better re-use
• Simon King (2011-03): use a faster way of conversion from the base ring.
• Volker Braun (2011-06): major cleanup, recfunc singular rings, bugfixes.

```python
class sage.rings.polynomial.multi_polynomial_libsingular.MPolynomialRing_libsingular
    Bases: sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base

Construct a multivariate polynomial ring subject to the following conditions:

INPUT:

- **base_ring** - base ring (must be either \( \text{GF}(q), \mathbb{Z}, \mathbb{Z}/n\mathbb{Z}, \mathbb{Q} \) or absolute number field)
- **n** - number of variables (must be at least 1)
- **names** - names of ring variables, may be string of list/tuple
- **order** - term order (default: degrevlex)

EXAMPLES:
```
sage: P.<x,y,z> = QQ[]
sage: P
Multivariate Polynomial Ring in x, y, z over Rational Field
```
sage: f = 27/113 * x^2 + y*z + 1/2; f
27/113*x^2 + y*z + 1/2
```
sage: P.term_order()
Degree reverse lexicographic term order
```
sage: P = PolynomialRing(GF(127),3,names='abc', order='lex')
sage: P
Multivariate Polynomial Ring in a, b, c over Finite Field of size 127
```
sage: a,b,c = P.gens()
sage: f = 57 * a^2*b + 43 * c + 1; f
57*a^2*b + 43*c + 1
```
sage: P.term_order()
Lexicographic term order
```
sage: z = QQ['z'].0
sage: K.<s> = NumberField(z^2 - 2)
sage: P.<x,y> = PolynomialRing(K, 2)
sage: f = 1/2*s*x^2 + 3/4*s
(1/2*s)*x^2 + (3/4*s)
```
sage: P.<x,y,z> = ZZ[]; P
Multivariate Polynomial Ring in x, y, z over Integer Ring
```
sage: P.<x,y,z> = Zmod(2^10)[]; P
Multivariate Polynomial Ring in x, y, z over Ring of integers modulo 1024
```
(continues on next page)
sage: P.<x,y,z> = Zmod(3^10)[]; P
Multivariate Polynomial Ring in x, y, z over Ring of integers modulo 59049

sage: P.<x,y,z> = Zmod(2^100)[]; P
Multivariate Polynomial Ring in x, y, z over Ring of integers modulo
1267650600228229401496703205376

sage: P.<x,y,z> = Zmod(2521352)[]; P
Multivariate Polynomial Ring in x, y, z over Ring of integers modulo 2521352

sage: type(P)
<type 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomialRing_libsingular'>

sage: P.<x,y,z> = Zmod(25213521351515232)[]; P
Multivariate Polynomial Ring in x, y, z over Ring of integers modulo 25213521351515232

sage: type(P)
<class 'sage.rings.polynomial.multi_polynomial_ring.MPolynomialRing_polydict_with_category'>

sage: P.<x,y,z> = PolynomialRing(Integers(2^32),order='lex')

3.1. Multivariate Polynomials and Polynomial Rings 393

Element
alias of MPolynomial_libsingular

gen(n=0)
Returns the n-th generator of this multivariate polynomial ring.

INPUT:
• n – an integer >= 0

EXAMPLES:

sage: P.<x,y,z> = QQ[]
sage: P.gen(),P.gen(1)
(x, y)

sage: P = PolynomialRing(GF(127),1000,'x')
sage: P.gen(500)
x500

sage: P.<SAGE,SINGULAR> = QQ[]  # weird names
sage: P.gen(1)
SINGULAR

ideal(*gens, **kwds)
Create an ideal in this polynomial ring.

INPUT:
• *gens - list or tuple of generators (or several input arguments)
• **coerce** - bool (default: True); this must be a keyword argument. Only set it to False if you are certain that each generator is already in the ring.

**EXAMPLES:**

```
sage: P.<x,y,z> = QQ[]
sage: sage.rings.ideal.Katsura(P)
Ideal (x + 2*y + 2*z - 1, x^2 + 2*y^2 + 2*z^2 - x, 2*x*y + 2*y*z - y) of
→Multivariate Polynomial Ring in x, y, z over Rational Field

sage: P.ideal([x + 2*y + 2*z - 1, 2*x*y + 2*y*z - y, x^2 + 2*y^2 + 2*z^2 - x])
Ideal (x + 2*y + 2*z - 1, 2*x*y + 2*y*z - y, x^2 + 2*y^2 + 2*z^2 - x) of
→Multivariate Polynomial Ring in x, y, z over Rational Field
```

**monomial_all_divisors(t)**

Return a list of all monomials that divide t.

Coefficients are ignored.

**INPUT:**

- **t** - a monomial

**OUTPUT:** a list of monomials

**EXAMPLES:**

```
sage: P.<x,y,z> = QQ[]
sage: P.monomial_all_divisors(x^2*z^3)
[x, x^2, z, x*z, x^2*z, z^2, x*z^2, x^2*z^2, z^3, x*z^3, x^2*z^3]
```

ALGORITHM: addwithcarry idea by Toon Segers

**monomial_divides(a, b)**

Return False if a does not divide b and True otherwise.

Coefficients are ignored.

**INPUT:**

- **a** – monomial
- **b** – monomial

**EXAMPLES:**

```
sage: P.<x,y,z> = QQ[]
sage: P.monomial_divides(x*y*z, x^3*y^2*z^4)
True
sage: P.monomial_divides(x^3*y^2*z^4, x*y*z)
False
```

**monomial_lcm(f, g)**

LCM for monomials. Coefficients are ignored.

**INPUT:**

- **f** - monomial
- **g** - monomial

**EXAMPLES:**

```
```python
sage: P.<x,y,z> = QQ[]
sage: P.monomial_lcm(3/2*x*y,x)
x^2*y
```

**monomial_pairwise_prime** *(g, h)*
Return True if h and g are pairwise prime. Both are treated as monomials.

Coefficients are ignored.

**INPUT:**
- h - monomial
- g - monomial

**EXAMPLES:**
```python
sage: P.<x,y,z> = QQ[]
sage: P.monomial_pairwise_prime(x^2*z^3, y^4)
True
sage: P.monomial_pairwise_prime(1/2*x^3*y^2, 3/4*y^3)
False
```

**monomial_quotient** *(f, g, coeff=False)*
Return \( \frac{f}{g} \), where both \( f \) and \( g \) are treated as monomials.

Coefficients are ignored by default.

**INPUT:**
- f - monomial
- g - monomial
- coeff - divide coefficients as well (default: False)

**EXAMPLES:**
```python
sage: P.<x,y,z> = QQ[]
sage: P.monomial_quotient(3/2*x*y,x)
y
sage: P.monomial_quotient(3/2*x*y,x,coeff=True)
3/2*y
```

Note, that \( \mathbb{Z} \) behaves different if coeff=True:
```python
sage: P.monomial_quotient(2*x,3*x)
1
sage: P.<x,y> = PolynomialRing(ZZ)
sage: P.monomial_quotient(2*x,3*x,coeff=True)
Traceback (most recent call last):
  ... ArithmeticError: Cannot divide these coefficients.
```
**Warning:** Assumes that the head term of f is a multiple of the head term of g and return the multiplicant m. If this rule is violated, funny things may happen.

**monomial_reduce(f, G)**

Try to find a g in G where g.lm() divides f. If found (flt, g) is returned, (0,0) otherwise, where flt is f/g.lm().

It is assumed that G is iterable and contains only elements in this polynomial ring.

Coefficients are ignored.

**INPUT:**

- f - monomial
- G - list/set of mpolynomials

**EXAMPLES:**

```sage
sage: P.<x,y,z> = QQ[]
sage: f = x*y^2
sage: G = [ 3/2*x^3 + y^2 + 1/2, 1/4*x*y + 2/7, 1/2 ]
sage: P.monomial_reduce(f,G)
(y, 1/4*x*y + 2/7)
```

**ngens()**

Returns the number of variables in this multivariate polynomial ring.

**EXAMPLES:**

```sage
sage: P.<x,y> = QQ[]
sage: P.ngens()
2
```

```sage
sage: k.<a> = GF(2^16)
sage: P = PolynomialRing(k,1000,'x')
sage: P.ngens()
1000
```

**class** `sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular`

Bases: `sage.rings.polynomial.multi_polynomial.MPolynomial`

A multivariate polynomial implemented using libSINGULAR.

**add_m_mul_q(m, q)**

Return self + m*q, where m must be a monomial and q a polynomial.

**INPUT:**

- m - a monomial
- q - a polynomial

**EXAMPLES:**

```sage
sage: P.<x,y,z>=PolynomialRing(QQ,3)
sage: x.add_m_mul_q(y,z)
y*z + x
```
**coefficient** *(degrees)*

Return the coefficient of the variables with the degrees specified in the python dictionary *degrees*. Mathematically, this is the coefficient in the base ring adjoined by the variables of this ring not listed in *degrees*. However, the result has the same parent as this polynomial.

This function contrasts with the function *monomial_coefficient* which returns the coefficient in the base ring of a monomial.

**INPUT:**

- *degrees* - Can be any of:
  - a dictionary of degree restrictions
  - a list of degree restrictions (with None in the unrestricted variables)
  - a monomial (very fast, but not as flexible)

**OUTPUT:** element of the parent of this element.

**Note:** For coefficients of specific monomials, look at *monomial_coefficient()*.

**EXAMPLES:**

```
sage: R.<x,y> = QQ[]
sage: f=x*y+y+5
sage: f.coefficient({x:0,y:1})
1
sage: f.coefficient({x:0})
y + 5
sage: f=(1+y+y^2)*(1+x+x^2)
sage: f.coefficient({x:0})
y^2 + y + 1
sage: f.coefficient([0,None])
y^2 + y + 1
sage: f.coefficient(x)
y^2 + y + 1

Note that exponents have all variables specified:
```
```
sage: x.coefficient(x.exponents()[0])
1
sage: f.coefficient([1,0])
1
sage: f.coefficient({x:1,y:0})
1

Be aware that this may not be what you think! The physical appearance of the variable x is deceiving – particularly if the exponent would be a variable.
```
```
sage: f.coefficient(x^0)  # outputs the full polynomial
x*y^2 + x*y + x*y^2 + x^2 + x*y + y^2 + x + y + 1
sage: R.<x,y> = GF(389)[]
sage: f=x*y+5
sage: c=f.coefficient({x:0,y:0}); c
```

(continues on next page)
AUTHOR:

- Joel B. Mohler (2007.10.31)

**coefficients()**

Return the nonzero coefficients of this polynomial in a list. The returned list is decreasingly ordered by the term ordering of the parent.

**EXAMPLES:**

```sage
sage: R.<x,y,z> = PolynomialRing(QQ, order='degrevlex')
sage: f=23*x^6*y^7 + x^3*y+6*x^7*z
sage: f.coefficients()
[23, 6, 1]
sage: R.<x,y,z> = PolynomialRing(QQ, order='lex')
sage: f=23*x^6*y^7 + x^3*y+6*x^7*z
sage: f.coefficients()
[6, 23, 1]
```

AUTHOR:

- Didier Deshommes

**constant_coefficient()**

Return the constant coefficient of this multivariate polynomial.

**EXAMPLES:**

```sage
sage: P.<x, y> = QQ[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.constant_coefficient()
5
sage: f = 3*x^2
sage: f.constant_coefficient()
0
```

**degree(x=None, std_grading=False)**

Return the degree of this polynomial.

**INPUT:**

- x – (default: None) a generator of the parent ring

**OUTPUT:**

If x is not given, return the maximum degree of the monomials of the polynomial. Note that the degree of a monomial is affected by the gradings given to the generators of the parent ring. If x is given, it is (or coercible to) a generator of the parent ring and the output is the maximum degree in x. This is not affected by the gradings of the generators.

**EXAMPLES:**
sage: R.<x, y> = QQ[
    sage: f = y^2 - x^9 - x
    sage: f.degree(x)
    9
    sage: f.degree(y)
    2
    sage: (y^10*x - 7*x^2*y^5 + 5*x^3).degree(x)
    3
    sage: (y^10*x - 7*x^2*y^5 + 5*x^3).degree(y)
    10

The term ordering of the parent ring determines the grading of the generators.

sage: T = TermOrder('wdegrevlex', (1,2,3,4))
    sage: R = PolynomialRing(QQ, 'x', 12, order=T+T+T)
    sage: [x.degree() for x in R.gens()]
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]

A matrix term ordering determines the grading of the generators by the first row of the matrix.

sage: m = matrix(3, [3,2,1,1,1,0,1,0,0])
    sage: m
    [3 2 1]
    [1 1 0]
    [1 0 0]
    sage: R.<x,y,z> = PolynomialRing(QQ, order=TermOrder(m))
    sage: x.degree(), y.degree(), z.degree()
(3, 2, 1)
    sage: f = x^3*y + x*z^4
    sage: f.degree()
11

If the first row contains zero, the grading becomes the standard one.

sage: m = matrix(3, [3,0,1,1,1,0,1,0,0])
    sage: m
    [3 0 1]
    [1 1 0]
    [1 0 0]
    sage: R.<x,y,z> = PolynomialRing(QQ, order=TermOrder(m))
    sage: x.degree(), y.degree(), z.degree()
(1, 1, 1)
    sage: f = x^3*y + x*z^4
    sage: f.degree()
5

To get the degree with the standard grading regardless of the term ordering of the parent ring, use `std_grading=True`.

sage: f.degree(std_grading=True)
5

degrees()
Returns a tuple with the maximal degree of each variable in this polynomial. The list of degrees is ordered
by the order of the generators.

EXAMPLES:

```python
sage: R.<y0,y1,y2> = PolynomialRing(QQ,3)
sage: q = 3*y0*y1^2*y2; q
3*y0*y1^2*y2
sage: q.degrees()
(1, 2, 1)
sage: (q + y0^5).degrees()
(5, 2, 1)
```

dict()
Return a dictionary representing self. This dictionary is in the same format as the generic MPolynomial: The dictionary consists of ETuple:coefficient pairs.

EXAMPLES:

```python
sage: R.<x,y,z> = QQ[]
sage: f=2*x*y^3*z^2 + 1/7*x^2 + 2/3
sage: f.dict()
{(0, 0, 0): 2/3, (1, 3, 2): 2, (2, 0, 0): 1/7}
```

divides(other)
Return True if this polynomial divides other.

EXAMPLES:

```python
sage: R.<x,y,z> = QQ[]
sage: p = 3*x*y + 2*y*z + x*z
sage: q = x + y + z + 1
sage: r = p * q
sage: p.divides(r)
True
sage: q.divides(p)
False
sage: r.divides(0)
True
sage: R.zero().divides(r)
False
sage: R.zero().divides(0)
True
```

exponents(as_ETuples=True)
Return the exponents of the monomials appearing in this polynomial.

INPUT:

* as_ETuples – (default: True) if True returns the result as an list of ETuples, otherwise returns a list of tuples

EXAMPLES:

```python
sage: R.<a,b,c> = QQ[]
sage: f = a^3 + b + 2*b^2
sage: f.exponents()
[(3, 0, 0), (0, 2, 0), (0, 1, 0)]
```
\begin{verbatim}
sage: f.exponents(as_ETuples=False)
[(3, 0, 0), (0, 2, 0), (0, 1, 0)]
\end{verbatim}

\textbf{\texttt{factor}} (\texttt{proof=None})

Return the factorization of this polynomial.

INPUT:

* proof - ignored.

EXAMPLES:

\begin{verbatim}
sage: R.<x, y> = QQ[]
sage: f = (x^3 + 2*y^2*x) * (x^2 + x + 1); f
x^5 + 2*x^3*y^2 + x^4 + 2*x^2*y^2 + x^3 + 2*x*y^2
sage: F = f.factor()
sage: F
x * (x^2 + x + 1) * (x^2 + 2*y^2)
\end{verbatim}

Next we factor the same polynomial, but over the finite field of order 3.:

\begin{verbatim}
sage: R.<x, y> = GF(3)[]
sage: f = (x^3 + 2*y^2*x) * (x^2 + x + 1); f
x^5 - x^3*y^2 + x^4 - x^2*y^2 + x^3 - x*y^2
sage: F = f.factor()
sage: F
# order is somewhat random
(-1) * x * (-x + y) * (x + y) * (x - 1)^2
\end{verbatim}

Next we factor a polynomial, but over a finite field of order 9.:

\begin{verbatim}
sage: K.<a> = GF(3^2)
sage: R.<x, y> = K[]
sage: f = (x^3 + 2*a*y^2*x) * (x^2 + x + 1); f
x^5 + (-a)*x^3*y^2 + x^4 + (-a)*x^2*y^2 + x^3 + (-a)*x*y^2
sage: F = f.factor()
sage: F
((-a)) * x * (x - 1)^2 * ((-a + 1)*x^2 + y^2)
sage: f - F
0
\end{verbatim}

Next we factor a polynomial over a number field.:

\begin{verbatim}
sage: p = var('p')
sage: K.<s> = NumberField(p^3-2)
sage: KXY.<x,y> = K[]
sage: factor(x^3 - 2*y^3)
(x + (-s)*y) * (x^2 + (s)*x*y + (s^2)*y^2)
sage: k = (x^3-2*y^3)^5*(x+s*y)^2*(2/3 + s^2)
sage: k.factor()
((s^2 + 2/3)) * (x + (s)*y)^2 * (x + (-s)*y)^5 * (x^2 + (s)*x*y + (s^2)*y^2)^5
\end{verbatim}

This shows that ticket trac ticket #2780 is fixed, i.e. that the unit part of the factorization is set correctly.

\section{Multivariate Polynomials and Polynomial Rings}

\index{multivariate polynomials}

\begin{verbatim}

(continued from previous page)

\end{verbatim}
Another example:

```python
sage: R.<x,y,z> = GF(32003)[]
sage: f = 9*(x-1)^2*(y+z)
sage: f.factor()
(9) * (y + z) * (x - 1)^2
```

Constant elements are factorized in the base rings.

```python
sage: P.<x,y> = ZZ[]
sage: P(2^3*7).factor()
2^3 * 7
sage: P.<x,y> = GF(2)[]
sage: P(1).factor()
1
```

Factorization for finite prime fields with characteristic > \(2^{29}\) is not supported

```python
sage: q = 1073741789
sage: T.<aa, bb> = PolynomialRing(GF(q))
sage: f = aa^2 + 12124343*bb*aa + 32434598*bb^2
sage: f.factor()
Traceback (most recent call last):
  ... NotImplementedError: Factorization of multivariate polynomials over prime fields with characteristic > 2^29 is not implemented.
```

Factorization over the integers is now supported, see trac ticket #17840:

```python
sage: P.<x,y> = PolynomialRing(ZZ)
sage: f = 12 * (3*x*y + 4) * (5*x - 2) * (2*y + 7)^2
sage: f.factor()
2^2 * 3 * (2*y + 7)^2 * (5*x - 2) * (3*x*y + 4)
```
sage: g = -12 * (x^2 - y^2)
sage: g.factor()
(-1) * 2^2 * 3 * (x - y) * (x + y)
sage: factor(-4*x*y - 2*x + 2*y + 1)
(-1) * (2*y + 1) * (2*x - 1)

Factorization over non-integral domains is not supported

sage: R.<x,y> = PolynomialRing(Zmod(4))
sage: f = (2*x + 1) * (x^2 + x + 1)
sage: f.factor()
Traceback (most recent call last):
...  
NotImplementedError: Factorization of multivariate polynomials over Ring of integers modulo 4 is not implemented.

\textbf{gcd}(right, algorithm=None, **kwds)

Return the greatest common divisor of self and right.

INPUT:

- right - polynomial
- algorithm - ezgcd - EZGCD algorithm - modular - multi-modular algorithm (default)
- **kwds - ignored

EXAMPLES:

sage: P.<x,y,z> = QQ[]
sage: f = (x*y*z)^6 - 1
sage: g = (x*y*z)^4 - 1
sage: f.gcd(g)
x^2*y^2*z^2 - 1
sage: GCD([x^3 - 3*x + 2, x^4 - 1, x^6 -1])
x - 1
sage: R.<x,y> = QQ[]
sage: f = (x^3 + 2*y^2*x)^2
sage: g = x^2*y^2
sage: f.gcd(g)
x^2
We compute a gcd over a finite field:

sage: F.<u> = GF(31^2)
sage: R.<x,y,z> = F[]
sage: p = x^3 + (1+u)*y^3 + z^3
sage: q = p^3 * (x - y + z*u)
sage: gcd(p,q)
x^3 + (u + 1)*y^3 + z^3
sage: gcd(p,q)  # yes, twice -- tests that singular ring is properly set.
x^3 + (u + 1)*y^3 + z^3

We compute a gcd over a number field:

sage: F.<u> = GF(31^2)
sage: R.<x,y,z> = F[]
sage: p = x^3 + (1+u)*y^3 + z^3
sage: q = p^3 * (x - y + z*u)
sage: gcd(p,q)
x^3 + (u + 1)*y^3 + z^3
```
sage: x = polygen(QQ)
sage: F.<u> = NumberField(x^3 - 2)
sage: R.<x,y,z> = F[]
sage: p = x^3 + (1+u)*y^3 + z^3
sage: q = p^3 * (x - y + z*u)
sage: gcd(p,q)
x^3 + (u + 1)*y^3 + z^3
```

### gradient()
Return a list of partial derivatives of this polynomial, ordered by the variables of the parent.

**EXAMPLES:**
```
sage: P.<x,y,z> = PolynomialRing(QQ,3)
sage: f= x*y + 1
sage: f.gradient()
[y, x, 0]
```

### hamming_weight()
Return the number of non-zero coefficients of this polynomial.

This is also called weight, `hamming_weight()` or sparsity.

**EXAMPLES:**
```
sage: R.<x, y> = ZZ[]
sage: f = x^3 - y
sage: f.number_of_terms()
2
sage: R(0).number_of_terms()
0
sage: f = (x+y)^100
sage: f.number_of_terms()
101
```

The method `hamming_weight()` is an alias:
```
sage: f.hamming_weight()
101
```

### integral(var)
Integrates this polynomial with respect to the provided variable.

One requires that \( Q \) is contained in the ring.

**INPUT:**
- `variable` - the integral is taken with respect to variable

**EXAMPLES:**
```
sage: R.<x, y> = PolynomialRing(QQ, 2)
sage: f = 3*x^3*y^2 + 5*y^2 + 3*x + 2
sage: f.integral(x)
3/4*x^4*y^2 + 5/2*y^2 + 3/2*x^2 + 2*x
sage: f.integral(y)
x^3*y^3 + 5/3*y^3 + 3*x*y^2 + 2*x*y
```

---

404 Chapter 3. Multivariate Polynomials
Check that trac ticket #15896 is solved:

```python
sage: s = x+y
sage: s.integral(x)+x
1/2*x^2 + x*y + x
sage: s.integral(x)*s
1/2*x^3 + 3/2*x^2*y + x*y^2
```

**inverse_of_unit()**

Return the inverse of this polynomial if it is a unit.

**EXAMPLES:**

```python
sage: R.<x,y> = QQ[]
sage: x.inverse_of_unit()
Traceback (most recent call last):
  ... ArithmeticError: Element is not a unit.
sage: R(1/2).inverse_of_unit()
2
```

**is_constant()**

Return True if this polynomial is constant.

**EXAMPLES:**

```python
sage: P.<x,y,z> = PolynomialRing(GF(127))
sage: x.is_constant()
False
sage: P(1).is_constant()
True
```

**is_homogeneous()**

Return True if this polynomial is homogeneous.

**EXAMPLES:**

```python
sage: P.<x,y> = PolynomialRing(RationalField(), 2)
sage: (x+y).is_homogeneous()
True
sage: (x.parent()(0)).is_homogeneous()
True
sage: (x+y^2).is_homogeneous()
False
sage: (x^2 + y^2).is_homogeneous()
True
sage: (x^2 + y^2*x).is_homogeneous()
False
sage: (x^2*y + y^2*x).is_homogeneous()
True
```

**is_monomial()**

Return True if this polynomial is a monomial. A monomial is defined to be a product of generators with coefficient 1.

**EXAMPLES:**
406 Chapter 3. Multivariate Polynomials

```python
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: x.is_monomial()
True
sage: (2*x).is_monomial()
False
sage: (x*y).is_monomial()
True
sage: (x*y + x).is_monomial()
False
sage: P(2).is_monomial()
False
sage: P.zero().is_monomial()
False
```

### is_squarefree()

Return True if this polynomial is square free.

**EXAMPLES:**

```python
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: f= x^2 + 2*x*y + 1/2*z
sage: f.is_squarefree()
True
sage: h = f^2
sage: h.is_squarefree()
False
```

### is_term()

Return True if self is a term, which we define to be a product of generators times some coefficient, which need not be 1.

Use `is_monomial()` to require that the coefficient be 1.

**EXAMPLES:**

```python
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: x.is_term()
True
sage: (2*x).is_term()
True
sage: (x*y).is_term()
True
sage: (x*y + x).is_term()
False
sage: P(2).is_term()
True
sage: P.zero().is_term()
False
```

### is_univariate()

Return True if self is a univariate polynomial, that is if self contains only one variable.

**EXAMPLES:**
```python
sage: P.<x,y,z> = GF(2)[
sage: f = x^2 + 1
sage: f.is_univariate()
    True
sage: f = y*x^2 + 1
sage: f.is_univariate()
    False
sage: f = P(0)
sage: f.is_univariate()
    True
```

**is_zero()**

Return True if this polynomial is zero.

**EXAMPLES:**

```python
sage: P.<x,y> = PolynomialRing(QQ)
sage: x.is_zero()
    False
sage: (x-x).is_zero()
    True
```

**iterator_exp_coeff(as_ETuples=True)**

Iterate over self as pairs of ((E)Tuple, coefficient).

**INPUT:**

- as_ETuples – (default: True) if True iterate over pairs whose first element is an ETuple, otherwise as a tuples

**EXAMPLES:**

```python
sage: R.<a,b,c> = QQ[
    f = a*c^3 + a^2*b + 2*b^4
    list(f.iterator_exp_coeff())
        [((0, 4, 0), 2), ((1, 0, 3), 1), ((2, 1, 0), 1)]
    list(f.iterator_exp_coeff(as_ETuples=False))
        [((0, 4, 0), 2), ((1, 0, 3), 1), ((2, 1, 0), 1)]
```
\texttt{sage: } f.lc()
\texttt{5}

\texttt{lcm(g)}

Return the least common multiple of \texttt{self} and \texttt{g}.

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage: } P.<x,y,z> = QQ[]
\texttt{sage: } p = (x+y)*(y+z)
\texttt{sage: } q = (z^4+2)*(y+z)
\texttt{sage: } lcm(p,q)
\texttt{x*y*z^4 + y^2*z^4 + x^4*z^5 + y*z^5 + 2*x*y^2 + 2*x^2*z + 2*y^2*z}
\texttt{sage: } P.<x,y,z> = ZZ[]
\texttt{sage: } p = 2*(x+y)*(y+z)
\texttt{sage: } q = 3*(z^4+2)*(y+z)
\texttt{sage: } lcm(p,q)
\texttt{6*x*y*z^4 + 6*y^2*z^4 + 6*x^4*z^5 + 6*y*z^5 + 12*x*y^2 + 12*x^2*z + 12*y^2*z}
\texttt{sage: } r.<x,y> = PolynomialRing(GF(2^{8}, 'a'), 2)
\texttt{sage: } a = r.base_ring().0
\texttt{sage: } f = (a^2+a)*x^2*y + (a^4+a^3+a)*y + a^5
\texttt{sage: } f.lcm(x^4)
\texttt{(a^2 + a)*x^6*y + (a^4 + a^3 + a)*x^4*y + (a^5)*x^4}
\texttt{sage: } w = var('w')
\texttt{sage: } r.<x,y> = PolynomialRing(NumberField(w^4 + 1, 'a'), 2)
\texttt{sage: } a = r.base_ring().0
\texttt{sage: } f = (a^2+a)*x^2*y + (a^4+a^3+a)*y + a^5
\texttt{sage: } f.lcm(x^4)
\texttt{(a^2 + a)*x^6*y + (a^3 + a - 1)*x^4*y + (-a)*x^4}
\end{verbatim}

\texttt{lift(I)}

\begin{verbatim}
given an ideal \(I = (f_1,\ldots,f_r)\) and some \(g (== \texttt{self})\) in \(I\), find \(s_1,\ldots,s_r\) such that \(g = s_1 f_1 + \ldots + s_r f_r\).

A ValueError exception is raised if \(g (== \texttt{self})\) does not belong to \(I\).

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage: } A.<x,y> = PolynomialRing(QQ,2,order='degrevlex')
\texttt{sage: } I = A.ideal([x^10 + x^9*y^2, y^8 - x^2*y^7 ])
\texttt{sage: } f = x*y^13 + y^12
\texttt{sage: } M = f.lift(I)
\texttt{sage: } M
\texttt{[y^7, x^7*y^2 + x^8 + x^5*y^3 + x^6*y + x^3*y^4 + x^4*y^2 + x^5*y^5 + x^2*y^3 + y^4]}
\texttt{sage: } sum( map( mul , zip( M, I.gens() ) ) ) == f
\texttt{True}
\end{verbatim}

Check that \texttt{trac ticket #13671} is fixed:
```
sage: R.<x1,x2> = QQ[]
sage: I = R.ideal(x2^2 + x1 - 2, x1**2 - 1)
sage: f = I.gen(0) + x2*I.gen(1)
sage: f.lift(I)
[1, x2]
sage: (f+1).lift(I)
Traceback (most recent call last):
  ...  
ValueError: polynomial is not in the ideal
sage: f.lift(I)
[1, x2]
```

**lm()**

Returns the lead monomial of self with respect to the term order of `self.parent()`. In Sage a monomial is a product of variables in some power without a coefficient.

**EXAMPLES:**

```
sage: R.<x,y,z>=PolynomialRing(GF(7),3,order='lex')
sage: f = x^1*y^2 + y^3*z^4
sage: f.lm()
x*y^2
sage: f.lm()
x^3*y^2*z^4
sage: f.lm()
x^3*y^2*z^4
sage: f.lm()
x*y^2*z^3
sage: f.lm()
x*y^2*z^4
```

**lt()**

Leading term of this polynomial. In Sage a term is a product of variables in some power and a coefficient.

**EXAMPLES:**

```
sage: R.<x,y,z>=PolynomialRing(GF(7),3,order='lex')
sage: f = 3*x^1*y^2 + 2*y^3*z^4
sage: f.lt()
3*x*y^2
```

(continues on next page)
monomial_coefficient(mon)

Return the coefficient in the base ring of the monomial mon in self, where mon must have the same parent as self.

This function contrasts with the function coefficient which returns the coefficient of a monomial viewing this polynomial in a polynomial ring over a base ring having fewer variables.

INPUT:

• mon - a monomial

OUTPUT:

coefficient in base ring

See also:

For coefficients in a base ring of fewer variables, look at coefficient.

EXAMPLES:

sage: P.<x,y> = QQ[]

The parent of the return is a member of the base ring.

sage: f = 2 * x * y
sage: c = f.monomial_coefficient(x*y); c
2
sage: c.parent()
Rational Field

sage: f = y^2 + y^2*x - x^9 - 7*x + 5*x*y
sage: f.monomial_coefficient(y^2)
1
sage: f.monomial_coefficient(x*y)
5
sage: f.monomial_coefficient(x^9)
-1
sage: f.monomial_coefficient(x^10)
0

monomials()

Return the list of monomials in self. The returned list is decreasingly ordered by the term ordering of self.parent().

EXAMPLES:

sage: P.<x,y,z> = QQ[]

sage: f = x + 3/2*y^2*z^2 + 2/3
sage: f.monomials()
[y^2z^2, x, 1]

number_of_terms()

Return the number of non-zero coefficients of this polynomial.
This is also called weight, `hamming_weight()` or sparsity.

**EXAMPLES:**

```
sage: R.<x, y> = ZZ[]
sage: f = x^3 - y
sage: f.number_of_terms()
2
sage: R(0).number_of_terms()
0
sage: f = (x+y)^100
sage: f.number_of_terms()
101
```

The method `hamming_weight()` is an alias:

```
sage: f.hamming_weight()
101
```

**numerator()**

Return a numerator of self computed as self * self.denominator()

If the base_field of self is the Rational Field then the numerator is a polynomial whose base_ring is the Integer Ring, this is done for compatibility to the univariate case.

**Warning:** This is not the numerator of the rational function defined by self, which would always be self since self is a polynomial.

**EXAMPLES:**

First we compute the numerator of a polynomial with integer coefficients, which is of course self.

```
sage: R.<x, y> = ZZ[]
sage: f = x^3 + 17*y + 1
sage: f.numerator()
x^3 + 17*y + 1
sage: f == f.numerator()
True
```

Next we compute the numerator of a polynomial with rational coefficients.

```
sage: R.<x,y> = PolynomialRing(QQ)
sage: f = (1/17)*x^19 - (2/3)*y + 1/3; f
1/17*x^19 - 2/3*y + 1/3
sage: f.numerator()
x^19 + 17/3*y + 1
sage: f == f.numerator()
False
sage: f.numerator().base_ring()
Integer Ring
```

We check that the computation of numerator and denominator is valid.
sage: K=QQ['x,y']
sage: f=K.random_element()
sage: f.numerator() / f.denominator() == f
True

The following tests against a bug fixed in trac ticket #11780:

sage: P.<foo,bar> = ZZ[]
sage: Q.<foo,bar> = QQ[]
sage: f = Q.random_element()
sage: f.numerator().parent() is P
True

nvariables()

Return the number variables in this polynomial.

EXAMPLES:

sage: P.<x,y,z> = PolynomialRing(GF(127))
sage: f = x*y + z
sage: f.nvariables()
3
sage: f = x + y
sage: f.nvariables()
2

quo_rem(right)

Returns quotient and remainder of self and right.

EXAMPLES:

sage: R.<x,y> = QQ[]
sage: f = y*x^2 + x + 1
sage: f.quo_rem(x)
(x*y + 1, 1)
sage: f.quo_rem(y)
(x^2, x + 1)

sage: R.<x,y> = ZZ[]
sage: f = 2*y*x^2 + x + 1
sage: f.quo_rem(x)
(2*x*y + 1, 1)
sage: f.quo_rem(y)
(2*x^2, x + 1)
sage: f.quo_rem(3*x)
(0, 2*x^2*y + x + 1)

reduce(I)

Return a remainder of this polynomial modulo the polynomials in I.

INPUT:

* I - an ideal or a list/set/iterable of polynomials.

OUTPUT:

A polynomial \( r \) such that:
- $\text{self} - r$ is in the ideal generated by $I$.
- No term in $r$ is divisible by any of the leading monomials of $I$.

The result $r$ is canonical if:

- $I$ is an ideal, and Sage can compute a Groebner basis of it.
- $I$ is a list/set/iterable that is a (strong) Groebner basis for the term order of $\text{self}$. (A strong Groebner basis is such that for every leading term $t$ of the ideal generated by $I$, there exists an element $g$ of $I$ such that the leading term of $g$ divides $t$.)

The result $r$ is implementation-dependent (and possibly order-dependent) otherwise. If $I$ is an ideal and no Groebner basis can be computed, its list of generators $I.gens()$ is used for the reduction.

**EXAMPLES:**

```python
sage: P.<x,y,z> = QQ[]
sage: f1 = -2 * x^2 + x^3
sage: f2 = -2 * y + x* y
sage: f3 = -x^2 + y^2
sage: F = Ideal([f1,f2,f3])
sage: g = x*y - 3*x*y^2
sage: g.reduce(F)
-6*y^2 + 2*y
sage: g.reduce(F.gens())
-6*y^2 + 2*y
```

$\mathbb{Z}$ is also supported.

```python
sage: P.<x,y,z> = ZZ[]
sage: f1 = -2 * x^2 + x^3
sage: f2 = -2 * y + x* y
sage: f3 = -x^2 + y^2
sage: F = Ideal([f1,f2,f3])
sage: g = x*y - 3*x*y^2
sage: g.reduce(F)
-6*y^2 + 2*y
sage: g.reduce(F.gens())
-6*y^2 + 2*y
```

The reduction is not canonical when $I$ is not a Groebner basis:

```python
sage: A.<x,y> = QQ[]
sage: (x+y).reduce([x+y, x-y])
2*y
sage: (x+y).reduce([x-y, x+y])
0
```

**resultant$(other, variable=None)$**

Compute the resultant of this polynomial and the first argument with respect to the variable given as the second argument.

If a second argument is not provide the first variable of the parent is chosen.

3.1. Multivariate Polynomials and Polynomial Rings 413
**INPUT:**
- `other` - polynomial
- `variable` - optional variable (default: None)

**EXAMPLES:**

```python
sage: P.<x,y> = PolynomialRing(QQ,2)
sage: a = x+y
sage: b = x^3-y^3
sage: c = a.resultant(b); c
-2*y^3
sage: d = a.resultant(b,y); d
2*x^3
```

The SINGULAR example:

```python
sage: R.<x,y,z> = PolynomialRing(GF(32003),3)
sage: f = 3 * (x+2)^3 + y
sage: g = x+y+z
sage: f.resultant(g,x)
3*y^3 + 9*y^2*z + 9*y*z^2 + 3*z^3 - 18*y^2 - 36*y*z - 18*z^2 + 35*y + 36*z - 24
```

Results are also supported over the Integers:

```python
sage: R.<x,y,a,b,u>=PolynomialRing(ZZ, 5, order='lex')
sage: r = (x^4*y^2+x^4*y-x*a-x*b+a*b+u,x)
sage: r
y^6*a^4 - 4*y^5*a^4*b - 4*y^5*a^3*u + y^5*a^2 - y^5 + 6*y^4*a^4*b^2 + 12*y^4*a^3*u
-3*b^2*a - 3*y^4*a^2*b + 6*y^4*a^2*u^2 - 2*y^4*a^2*b + 4*y^4*b - 4*y^3*a^4*b^3 -w
-12*y^3*a^3*b^2*u + 6*y^3*a^2*b^2 - 12*y^3*a^2*b*u^2 + 6*y^3*a*b*u - 4*y^3*a*b^2
-3*a - 6*y^3*b^2 + y^3*u^2 + y^2*b*a^4*b^4 + 4*y^2*a^3*b^3 + 3*y^2*a^3*b^3 -u
-2*y^2*b^2 + 2*b^2 + 2*b^2 + 2*y^2*b^2 + 2*y^2*b + 2*y^2*b + 2*y^2*b + 2*y^2*b + 2*y^2*b + 2
```

**sub_m_mul_q(m, q)**

Return self - m*q, where m must be a monomial and q a polynomial.

**INPUT:**
- `m` - a monomial
- `q` - a polynomial

**EXAMPLES:**

```python
sage: P.<x,y,z>=PolynomialRing(QQ,3)
sage: x.sub_m_mul_q(y,z)
-y*z + x
```

**subs(fixed=None, **kw)**

Fixes some given variables in a given multivariate polynomial and returns the changed multivariate polynomials. The polynomial itself is not affected. The variable,value pairs for fixing are to be provided as dictionary of the form `{variable:value}`.

This is a special case of evaluating the polynomial with some of the variables constants and the others the original variables, but should be much faster if only few variables are to be fixed.
INPUT:

- **fixed** - (optional) dict with variable:value pairs
- ****kw** - names parameters

OUTPUT: a new multivariate polynomial

EXAMPLES:

```python
sage: R.<x,y> = QQ[]
sage: f = x^2 + y + x^2*y^2 + 5
sage: f(5,y)
25*y^2 + y + 30
sage: f.subs({x:5})
25*y^2 + y + 30
sage: f.subs(x=5)
25*y^2 + y + 30
sage: P.<x,y,z> = PolynomialRing(GF(2),3)
sage: f = x + y + 1
sage: f.subs({x:y+1})
0
sage: f.subs(x=y)
1
sage: f.subs(x=x)
x + y + 1
sage: f.subs({x:z})
y + z + 1
sage: f.subs({x:z+1})
y + z
sage: f.subs({x:1/y})
(y^2 + y + 1)/y
sage: f.subs({x:1/y})
(y^2 + y + 1)/y
```

The parameters are substituted in order and without side effects:

```python
sage: R.<x,y>=QQ[]
sage: g=x+y
g.subs({x:x+1,y:x*y})
x*y + x + 1
sage: g.subs({x:x+1}).subs({y:x*y})
x*y + x + 1
sage: g.subs({y:x*y}).subs({x:x+1})
x*y + x + y + 1
```

```python	sage: R.<x,y> = QQ[]
sage: f = x + 2*y
sage: f.subs(x=y,y=x)
2*x + y
```

**total_degree**(std_grading=False)

Return the total degree of *self*, which is the maximum degree of all monomials in *self*.
EXAMPLES:

```
sage: R.<x,y,z> = QQ[

sage: f = 2*x*y^3*z^2

sage: f.total_degree()

6
sage: f = 4*x^2*y^2*z^3

sage: f.total_degree()

7
sage: f = 99*x^6*y^3*z^9

sage: f.total_degree()

18
sage: f = x^y^3*z^6+3*x^2

sage: f.total_degree()

10
sage: f = z^3+8*x^4*y^5*z

sage: f.total_degree()

10
sage: f = z^9+10*x^4+y^8*x^2

sage: f.total_degree()

10
```

A matrix term ordering changes the grading. To get the total degree using the standard grading, use `std_grading=True`:

```
sage: tord = TermOrder(matrix(3, [[3,2,1],[1,1,0],[1,0,0]]))

sage: tord
Matrix term order with matrix
[[3 2 1]
 [1 1 0]
 [1 0 0]]

sage: R.<x,y,z> = PolynomialRing(QQ, order=tord)

sage: f = x^2*y

sage: f.total_degree()

8
sage: f.total_degree(std_grading=True)

3
```

`univariate_polynomial(R=None)`

Returns a univariate polynomial associated to this multivariate polynomial.

**INPUT:**

- `R` (default: `None`) `PolynomialRing`

If this polynomial is not in at most one variable, then a `ValueError` exception is raised. This is checked using the `is_univariate()` method. The new Polynomial is over the same base ring as the given `MPolynomial` and in the variable `x` if no ring `R` is provided.

**EXAMPLES:**

```
sage: R.<x, y> = QQ[

sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5

sage: f.univariate_polynomial()

Traceback (most recent call last):
...
```

(continues on next page)
sage: g = f.subs({x:10}); g
700*y^2 - 2*y + 305

debug: trying to get one variable out
sage: g.univariate_polynomial()
700*y^2 - 2*y + 305
sage: g.univariate_polynomial(PolynomialRing(QQ, 'z'))
700*z^2 - 2*z + 305

Here’s an example with a constant multivariate polynomial:

sage: g = R(1)
sage: h = g.univariate_polynomial(); h
1
sage: h.parent()
Univariate Polynomial Ring in x over Rational Field

**variable** (i=0)

Return the i-th variable occurring in self. The index i is the index in self.variables().

**EXAMPLES:**

sage: P.<x,y,z> = GF(2)[]
sage: f = x*y^2 + z + 1
sage: f.variables()
(x, z)
sage: f.variable(1)
z

**variables()**

Return a tuple of all variables occurring in self.

**EXAMPLES:**

sage: P.<x,y,z> = GF(2)[]
sage: f = x*y^2 + z + 1
sage: f.variables()
(x, z)

sage.rings.polynomial.multi_polynomial_libsingular.unpickle_MPolynomialRing_libsingular(base_ring, names, term_order)

inverse function for MPolynomialRing_libsingular.__reduce__

**EXAMPLES:**

sage: P.<x,y> = PolynomialRing(QQ)
sage: loads(dumps(P)) is P # indirect doctest
True

sage.rings.polynomial.multi_polynomial_libsingular.unpickle_MPolynomial_libsingular(R, d)

Deserialize an MPolynomial_libsingular object

**INPUT:**

- R - the base ring
- d - a Python dictionary as returned by MPolynomial_libsingular.dict()
EXAMPLES:

```python
sage: P.<x,y> = PolynomialRing(QQ)
sage: loads(dumps(x)) == x # indirect doctest
True
```

### 3.1.9 Direct low-level access to SINGULAR’s Groebner basis engine via libSINGULAR

**AUTHOR:**
- Martin Albrecht (2007-08-08): initial version

**EXAMPLES:**

```python
sage: x,y,z = QQ['x,y,z'].gens()
sage: I = ideal(x^5 + y^4 + z^3 - 1, x^3 + y^3 + z^2 - 1)
sage: I.groebner_basis('libsingular:std')
[y^6 + x*y^4 + 2*y^3*z^2 + x*z^3 + z^4 - 2*y^3 - 2*z^2 - x + 1,
  x^2*y^3 - y^4 + x^2*z^2 - z^3 - x^2 + 1, x^3 + y^3 + z^2 - 1]
```

We compute a Groebner basis for cyclic 6, which is a standard benchmark and test ideal:

```python
sage: R.<x,y,z,t,u,v> = QQ['x,y,z,t,u,v']
sage: I = sage.rings.ideal.Cyclic(R,6)
sage: B = I.groebner_basis('libsingular:std')
sage: len(B)
45
```

Two examples from the Mathematica documentation (done in Sage):
- We compute a Groebner basis:

```python
sage: R.<x,y> = PolynomialRing(QQ, order='lex')
sage: ideal(x^2 - 2*y^2, x*y - 3).groebner_basis('libsingular:slimgb')
[x - 2/3*y^3, y^4 - 9/2]
```
- We show that three polynomials have no common root:

```python
sage: R.<x,y> = QQ[]
sage: ideal(x+y, x^2 - 1, y^2 - 2*x).groebner_basis('libsingular:slimgb')
[1]
```

`sage.rings.polynomial.multi_polynomial_ideal_libsingular.interreduced_libsingular(I)`
SINGULAR’s `interred()` command.

**INPUT:**
- `I` – a Sage ideal

**EXAMPLES:**

```python
sage: P.<x,y,z> = PolynomialRing(ZZ)
sage: I = ideal( x^2 - 3*y, y^3 - x*y, z^3 - x, x^4 - y^z + 1 )
sage: I.interreduced_basis()
[y^z^2 - 81*x*y - 9*y - z, z^3 - x, x^2 - 3*y, 9*y^2 - y*z + 1]
```
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: I = ideal( x^2 - 3*y, y^3 - x*y, z^3 - x, x^4 - y*z + 1 )
sage: I.interreduced_basis()
[y*z^2 - 81*x*y - 9*y - z, z^3 - x, x^2 - 3*y, y^2 - 1/9*y*z + 1/9]

sage.rings.polynomial.multi_polynomial_ideal_libsingular.kbase_libsingular(I, degree=None)
SINGULAR’s kbase() algorithm.

INPUT:

- I – a groebner basis of an ideal
- degree – integer (default: None); if not None, return only the monomials of the given degree

OUTPUT:

Computes a vector space basis (consisting of monomials) of the quotient ring by the ideal, resp. of a free module by the module, in case it is finite dimensional and if the input is a standard basis with respect to the ring ordering. If the input is not a standard basis, the leading terms of the input are used and the result may have no meaning.

With two arguments: computes the part of a vector space basis of the respective quotient with degree of the monomials equal to the second argument. Here, the quotient does not need to be finite dimensional.

EXAMPLES:

sage: R.<x,y> = PolynomialRing(QQ, order='lex')
sage: I = R.ideal(x^2-2*y^2, x*y-3)
sage: I.normal_basis() # indirect doctest
[y^3, y^2, y, 1]
sage: J = R.ideal(x^2-2*y^2)

sage: [J.normal_basis(d) for d in (0..4)] # indirect doctest
[[1], [y, x], [y^2, x*y], [y^3, x*y^2], [y^4, x*y^3]]

sage.rings.polynomial.multi_polynomial_ideal_libsingular.slimgb_libsingular(I)
SINGULAR’s slimgb() algorithm.

INPUT:

- I – a Sage ideal

sage.rings.polynomial.multi_polynomial_ideal_libsingular.std_libsingular(I)
SINGULAR’s std() algorithm.

INPUT:

- I – a Sage ideal

3.1.10 PolyDict engine for generic multivariate polynomial rings

This module provides an implementation of the underlying arithmetic for multi-variate polynomial rings using Python dicts.

This class is not meant for end users, but instead for implementing multivariate polynomial rings over a completely general base. It does not do strong type checking or have parents, etc. For speed, it has been implemented in Cython.

The functions in this file use the ‘dictionary representation’ of multivariate polynomials

\{(e_1, ..., e_r):c_1, ...\} \leftrightarrow c_1 x^{e_1} \cdots x^r \cdot e_r + \cdots
which we call a polydict. The exponent tuple \((e_1, \ldots, e_r)\) in this representation is an instance of the class \(ETuple\). This class behaves like a normal Python tuple but also offers advanced access methods for sparse monomials like positions of non-zero exponents etc.

AUTHORS:

- William Stein
- David Joyner
- Martin Albrecht (ETuple)
- Joel B. Mohler (2008-03-17) – ETuple rewrite as sparse C array

class sage.rings.polynomial.polydict.ETuple

Bases: object

Representation of the exponents of a polydict monomial. If \((0,0,3,0,5)\) is the exponent tuple of \(x_2^3 \cdot x_4^5\) then this class only stores \(\{2:3, 4:5\}\) instead of the full tuple. This sparse information may be obtained by provided methods.

The index/value data is all stored in the \_data\ C int array member variable. For the example above, the C array would contain 2,3,4,5. The indices are interlaced with the values.

This data structure is very nice to work with for some functions implemented in this class, but tricky for others. One reason that I really like the format is that it requires a single memory allocation for all of the values. A hash table would require more allocations and presumably be slower. I didn’t benchmark this question (although, there is no question that this is much faster than the prior use of python dicts).

combine_to_positives\((other)\)

Given a pair of ETuples \((self, other)\), returns a triple of ETuples \((a, b, c)\) so that \(self = a + b, other = a + c\) and \(b\) and \(c\) have all positive entries.

EXAMPLES:

```python
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([-2, 1, -5, 3, 1, 0])
print(e.combine_to_positives(f))
((-2, -3, -5, 3, 0, 0), (0, 4, 0, 0, 1, 0), (3, 0, 2, 1, 0, 2))
```

common_nonzero_positions\((other, sort=False)\)

Returns an optionally sorted list of non-zero positions either in self or other, i.e. the only positions that need to be considered for any vector operation.

EXAMPLES:

```python
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([1, 0, 2])
sage: f = ETuple([0, 0, 1])
sage: e.common_nonzero_positions(f)
{0, 2}
sage: e.common_nonzero_positions(f, sort=True)
[0, 2]
```

dotprod\((other)\)

Return the dot product of this tuple by other.

EXAMPLES:
**eadd**(other)
Vector addition of self with other.

EXAMPLES:

```
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([1,0,2])
sage: f = ETuple([0,1,1])
sage: e.dotprod(f)
2
sage: e = ETuple([1,1,-1])
sage: f = ETuple([0,-2,1])
sage: e.dotprod(f)
-3
```

Verify that trac ticket #6428 has been addressed:

```
sage: R.<y, z> = Frac(QQ['x'])[]
sage: type(y)<class 'sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict'>
sage: y^(2^32)
Traceback (most recent call last):
  ... OverflowError: exponent overflow (2147483648)
```

**eadd_p**(other, pos)
Add other to self at position pos.

EXAMPLES:

```
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([1,0,2])
(1, 1, 3)
```

```
sage: e = ETuple([0] * 7)
(0, 0, 0, 0, 5, 0, 0)
```

```
sage: ETuple([0,1]).eadd_p(1, 0) == ETuple([1,1])
True
```

**eadd_scaled**(other, scalar)
Vector addition of self with scalar * other.

EXAMPLES:

```
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([1,0,2])
(1, 5, 2)
```

```
sage: e = ETuple([0]*7)
(0, 0, 0, 0, 5, 0, 0)
```

```
sage: ETuple([0,1]).eadd_p(1, 0) == ETuple([1,1])
True
(continues on next page)```
\begin{verbatim}
sage: e.eadd_scaled(f, 3)
(1, 3, 5)
\end{verbatim}

**emax(other)**

Vector of maximum of components of self and other.

EXAMPLES:

\begin{verbatim}
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([1,0,2])
sage: f = ETuple([0,1,1])
sage: e.emax(f)
(1, 1, 2)
sage: e = ETuple([1,2,3,4])
sage: f = ETuple([4,0,2,1])
sage: f.emax(e)
(4, 2, 3, 4)
sage: e = ETuple([1,-2,-2,4])
sage: f = ETuple([4,0,0,0])
sage: f.emax(e)
(4, 0, 0, 4)
sage: f.emax(e).nonzero_positions()
[0, 3]
\end{verbatim}

**emin(other)**

Vector of minimum of components of self and other.

EXAMPLES:

\begin{verbatim}
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([1,0,2])
sage: f = ETuple([0,1,1])
sage: e.emin(f)
(0, 0, 1)
sage: e = ETuple([1,0,-1])
sage: f = ETuple([0,-2,1])
sage: e.emin(f)
(0, -2, -1)
\end{verbatim}

**emul(factor)**

Scalar Vector multiplication of self.

EXAMPLES:

\begin{verbatim}
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([1,0,2])
sage: e.emul(2)
(2, 0, 4)
\end{verbatim}

**escalar_div(n)**

Divide each exponent by n.

EXAMPLES:
**etuple**

Vector subtraction of self with other.

**Examples:**

```python
sage: from sage.rings.polynomial.polydict import ETuple
tsage: e = ETuple([1,0,2])
tsage: f = ETuple([0,1,1])
tsage: e.esub(f)
(1, -1, 1)
```

**is_constant()**

Return if all exponents are zero in the tuple.

**Examples:**

```python
sage: from sage.rings.polynomial.polydict import ETuple
tsage: e = ETuple([1,0,2])
tsage: e.is_constant()
False
```

**is_multiple_of(n)**

Test whether each entry is a multiple of n.

**Examples:**

```python
sage: from sage.rings.polynomial.polydict import ETuple
tsage: ETuple([0,0]).is_multiple_of(3)
True
```

**nonzero_positions(sort=False)**

Return the positions of non-zero exponents in the tuple.

**Input:**

- `sort` – (default: False) if True a sorted list is returned; if False an unsorted list is returned

**Examples:**

```python
```
```python
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([1,0,2])
sage: e.nonzero_positions()
[0, 2]
```

### nonzero_values(sort=True)

Return the non-zero values of the tuple.

**INPUT:**

- `sort` – (default: True) if True the values are sorted by their indices; otherwise the values are returned unsorted

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([2,0,1])
sage: e.nonzero_values()
[2, 1]
sage: f = ETuple([0,-1,1])
sage: f.nonzero_values(sort=True)
[-1, 1]
```

### reversed()

Return the reversed ETuple of `self`.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([1,2,3])
sage: e.reversed()
(3, 2, 1)
```

### sparse_iter()

Iterator over the elements of `self` where the elements are returned as (i, e) where i is the position of e in the tuple.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([1,0,2,0,3])
sage: list(e.sparse_iter())
[(0, 1), (2, 2), (4, 3)]
```

### unweighted_degree()

Return the sum of entries.

**ASSUMPTION:**

All entries are non-negative.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([1,0,2,0])
sage: e.unweighted_degree()
4
```
class sage.rings.polynomial.polydict.PolyDict
    Bases: object

INPUT:

• pdict – dict or list, which represents a multi-variable polynomial with the distribute representation (a copy
  is not made)
• zero – (optional) zero in the base ring
• force_int_exponents – bool (optional) arithmetic with int exponents is much faster than some of the
  alternatives, so this is True by default
• force_etuples – bool (optional) enforce that the exponent tuples are instances of ETuple class

EXAMPLES:

```python
sage: from sage.rings.polynomial.polydict import PolyDict
sage: PolyDict({(2,3):2, (1,2):3, (2,1):4})
PolyDict with representation {(1, 2): 3, (2, 1): 4, (2, 3): 2}

# I've removed fractional exponent support in ETuple when moving to a sparse C integer array
#PolyDict with representation {(2, 1): 4, (1, 2, 1): 3, (2/3, 3, 5): 2}

sage: PolyDict({(2,3):0, (1,2):3, (2,1):4}, remove_zero=True)
PolyDict with representation {(1, 2): 3, (2, 1): 4}

sage: PolyDict({(0,0):RIF(-1,1)}, remove_zero=True)
PolyDict with representation {(0, 0): 0.?}
```

coefficient(mon)

Return a polydict that defines a polynomial in 1 less number of variables that gives the coefficient of mon
in this polynomial.

The coefficient is defined as follows. If f is this polynomial, then the coefficient is the sum T/mon where
the sum is over terms T in f that are exactly divisible by mon.

coefficients()

Return the coefficients of self.

EXAMPLES:

```python
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict({(2,3):2, (1,2):3, (2,1):4})
sage: sorted(f.coefficients())
[2, 3, 4]
```

degree(x=None)

dict()

Return a copy of the dict that defines self. It is safe to change this. For a reference, use dictref.

EXAMPLES:

```python
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict({(2,3):2, (1,2):3, (2,1):4})
sage: f.dict()
{(1, 2): 3, (2, 1): 4, (2, 3): 2}
```
exponents()
Return the exponents of self.

EXAMPLES:
```
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict(((2,3):2, (1,2):3, (2,1):4))
sage: sorted(f.exponents())
[(1, 2), (2, 1), (2, 3)]
```

homogenize(var)

is_constant()
Return True if self is a constant and False otherwise.

EXAMPLES:
```
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict(((2,3):2, (1,2):3, (2,1):4))
sage: f.is_constant()
False
sage: g = PolyDict(((0,0):2))
sage: g.is_constant()
True
sage: h = PolyDict({})
sage: h.is_constant()
True
```

is_homogeneous()

latex(vars, atomic_exponents=True, atomic_coefficients=True, sortkey=None)
Return a nice polynomial latex representation of this PolyDict, where the vars are substituted in.

INPUT:
• vars – list
• atomic_exponents – bool (default: True)
• atomic_coefficients – bool (default: True)

EXAMPLES:
```
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict(((2,3):2, (1,2):3, (2,1):4))
sage: latex(['a', 'WW'], atomic_exponents=True)
'2 a^{2} WW^{3} + 4 a^{2} WW + 3 a WW^{2}'
```

When atomic_exponents is False, the exponents are surrounded in parenthesis, since ^ has such high precedence:
```
# I've removed fractional exponent support in ETuple when moving to a sparse C→
integer array
→exponents=False)
#sage: f.latex(['a', 'b', 'c'], atomic_exponents=False)
#'4 a^{2}bc + 3 ab^{2}c + 2 a^{2/3}b^{3}c^{5}'
```
\texttt{lcm}\texttt{(greater\_etuple)}

Provides functionality of \texttt{lc}, \texttt{lm}, and \texttt{lt} by calling the tuple compare function on the provided term order \(T\).

\textbf{INPUT:}

\begin{itemize}
  \item \texttt{greater\_etuple} – a term order
\end{itemize}

\texttt{list()}

Return a list that defines \texttt{self}. It is safe to change this.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict({(2,3):2, (1,2):3, (2,1):4})
sage: sorted(f.list())
[[2, [2, 3]], [3, [1, 2]], [4, [2, 1]]]
\end{verbatim}

\texttt{max\_exp()}

Returns an ETuple containing the maximum exponents appearing. If there are no terms at all in the PolyDict, it returns None.

The \texttt{nvars} parameter is necessary because a PolyDict doesn’t know it from the data it has (and an empty PolyDict offers no clues).

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict({(2,3):2, (1,2):3, (2,1):4})
sage: f.max_exp()
(2, 3)
sage: PolyDict({}).max_exp() # returns None
\end{verbatim}

\texttt{min\_exp()}

Returns an ETuple containing the minimum exponents appearing. If there are no terms at all in the PolyDict, it returns None.

The \texttt{nvars} parameter is necessary because a PolyDict doesn’t know it from the data it has (and an empty PolyDict offers no clues).

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict({(2,3):2, (1,2):3, (2,1):4})
sage: f.min_exp()
(1, 1)
sage: PolyDict({}).min_exp() # returns None
\end{verbatim}

\texttt{monomial\_coefficient}(\texttt{mon})

\textbf{INPUT:}

a PolyDict with a single key

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict({(2,3):2, (1,2):3, (2,1):4})
sage: f.monomial_coefficient(PolyDict({(2,1):1}).dict())
\end{verbatim}
**poly_repr**

Return a nice polynomial string representation of this PolyDict, where the vars are substituted in.

**INPUT:**
- vars – list
- atomic_exponents – bool (default: True)
- atomic_coefficients – bool (default: True)

**EXAMPLES:**

```sage
def poly_repr_ex(f, vars, atomic_exponents=True, atomic_coefficients=True, sortkey=None):
    return f.poly_repr(vars, atomic_exponents=atomic_exponents, atomic_coefficients=atomic_coefficients, sortkey=sortkey)
```

```
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict({(2,3):2, (1,2):3, (2,1):4})
sage: f.poly_repr(['a', 'WW'])
'2*a^2*WW^3 + 4*a^2*WW + 3*a*WW^2'
```

When atomic_exponents is False, the exponents are surrounded in parenthesis, since ^ has such high precedence.

```sage
# I've removed fractional exponent support in ETuple when moving to a sparse C
˓→integer array
#sage: f.poly_repr(['a', 'b', 'c'], atomic_exponents=False)
#'4*a^(2)*b*c + 3*a*b^(2)*c + 2*a^(2/3)*b^(3)*c^(5)'
```

We check to make sure that when we are in characteristic two, we don’t put negative signs on the generators.

```sage
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict({(2,3):RIF(1/2,3/2), (1,2):RIF(-1,1)})
sage: f.poly_repr(['x', 'y'])
'1.?*x^2*y^3 + 0.?*x*y^2'
```

**polynomial_coefficient**

Return a polydict that defines the coefficient in the current polynomial viewed as a tower of polynomial extensions.

**INPUT:**
- degrees – a list of degree restrictions; list elements are None if the variable in that position should be unrestricted

**EXAMPLES:**

```sage
def polynomial_coefficient(f, degrees):
    return f.polynomial_coefficient(degrees)
```

```
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict({(2,3):2, (1,2):3, (2,1):4})
sage: f.polynomial_coefficient([None, None])
PolyDict with representation {{0, 1}: 4, (0, 3): 2}
sage: f = PolyDict({(0,3):2, (0,2):3, (2,1):4})
sage: f.polynomial_coefficient([None, None])
PolyDict with representation {{0, 2}: 3, (0, 3): 2}
```
rich_compare(other, op, sortkey=None)

Compare two PolyDict argument is given it should be a sort key used to specify a term order.

If not sort key is provided than only comparison by equality (== or !=) is supported.

EXAMPLES:

```python
from sage.rings.polynomial.polydict import PolyDict
from sage.structure.richcmp import op_EQ, op_NE, op_LT

p1 = PolyDict({(0,): 1})
p2 = PolyDict({(0,): 2})
p1.rich_compare(PolyDict({(0,): 1}), op_EQ)
True
p1.rich_compare(p2, op_EQ)
False
p1.rich_compare(p2, op_NE)
True
p1.rich_compare(p2, op_LT)
Traceback (most recent call last):
  ... TypeError: ordering of PolyDicts requires a sortkey

O = TermOrder()
p1.rich_compare(p2, op_LT, O.sortkey)
True

p3 = PolyDict({(3, 2, 4): 1, (3, 2, 5): 2})
p4 = PolyDict({(3, 2, 4): 1, (3, 2, 3): 2})
p3.rich_compare(p4, op_LT, O.sortkey)
False
```

scalar_lmult(s)

Left Scalar Multiplication

EXAMPLES:

```python
from sage.rings.polynomial.polydict import PolyDict
x, y = FreeMonoid(2, 'x, y').gens()  # a strange object to live in a polydict, but non-commutative!
f = PolyDict({(2, 3): x})
f.scalar_lmult(y)
PolyDict with representation {(2, 3): y*x}
f = PolyDict({(2, 3): 2, (1, 2): 3, (2, 1): 4})
f.scalar_lmult(-2)
PolyDict with representation {(1, 2): -6, (2, 1): -8, (2, 3): -4}
f.scalar_lmult(RIF(-1,1))
PolyDict with representation {(1, 2): 0.?e1, (2, 1): 0.?e1, (2, 3): 0.?e1}
```

scalar_rmult(s)

Right Scalar Multiplication

EXAMPLES:

```python
from sage.rings.polynomial.polydict import PolyDict
x, y = FreeMonoid(2, 'x, y').gens()  # a strange object to live in a polydict, but non-commutative!
f = PolyDict({(2, 3): x})
f.scalar_rmult(y)
PolyDict with representation {x*(2, 3)}
f = PolyDict({(2, 3): 2, (1, 2): 3, (2, 1): 4})
f.scalar_rmult(-2)
PolyDict with representation {6*(1, 2) - 8*(2, 1) - 4*(2, 3)}
f.scalar_rmult(RIF(-1,1))
PolyDict with representation {(1, 2): 0.?e1, (2, 1): 0.?e1, (2, 3): 0.?e1}
```
term_lmult(exponent, s)
Return this element multiplied by s on the left and with exponents shifted by exponent.

INPUT:

• exponent – a ETuple
• s – a scalar

EXAMPLES:

sage: from sage.rings.polynomial.polydict import ETuple, PolyDict
sage: x, y = FreeMonoid(2, 'x, y').gens() # a strange object to live in a...
˓→polydict, but non-commutative!
sage: f = PolyDict({(2, 3): x})
sage: f.term_lmult(ETuple((1, 2)), y)
PolyDict with representation {(3, 5): y*x}

sage: f = PolyDict({(2, 3): 2, (1,2): 3, (2,1): 4})
sage: f.term_lmult(ETuple((1, 2)), -2)
PolyDict with representation {(2, 4): -6, (3, 3): -8, (3, 5): -4}

term_rmult(exponent, s)
Return this element multiplied by s on the right and with exponents shifted by exponent.

INPUT:

• exponent – a ETuple
• s – a scalar

EXAMPLES:

sage: from sage.rings.polynomial.polydict import ETuple, PolyDict
sage: x, y = FreeMonoid(2, 'x, y').gens() # a strange object to live in a...
˓→polydict, but non-commutative!
sage: f = PolyDict({(2, 3): x})
sage: f.term_rmult(ETuple((1, 2)), y)
PolyDict with representation {(3, 5): y*x}

sage: f = PolyDict({(2, 3): 2, (1,2): 3, (2,1): 4})
sage: f.term_rmult(ETuple((1, 2)), -2)
PolyDict with representation {(2, 4): -6, (3, 3): -8, (3, 5): -4}

total_degree()
valuation(x=None)
sage.rings.polynomial.polydict.make_ETuple(data, length)
sage.rings.polynomial.polydict.make_PolyDict(data)

### 3.1.11 Compute Hilbert series of monomial ideals

This implementation was provided at trac ticket #26243 and is supposed to be a way out when Singular fails with an int overflow, which will regularly be the case in any example with more than 34 variables.

```python
class sage.rings.polynomial.hilbert.Node
    Bases: object

    A node of a binary tree

    It has slots for data that allow to recursively compute the first Hilbert series of a monomial ideal.

sage.rings.polynomial.hilbert.first_hilbert_series(I, grading=None, return_grading=False)

Return the first Hilbert series of the given monomial ideal.

**INPUT:**

- `I` – a monomial ideal (possibly defined in singular)
- `grading` – (optional) a list or tuple of integers used as degree weights
- `return_grading` – (default: False) whether to return the grading

**OUTPUT:**

A univariate polynomial, namely the first Hilbert function of `I`, and if `return_grading=True` also the grading used to compute the series.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.hilbert import first_hilbert_series
sage: R = singular.ring(0, '(x,y,z)', 'dp')
sage: I = singular.ideal(['x^2','y^2','z^2'])
sage: first_hilbert_series(I)
-t^6 + 3*t^4 - 3*t^2 + 1
sage: first_hilbert_series(I, return_grading=True)
(-t^6 + 3*t^4 - 3*t^2 + 1, (1, 1, 1))
sage: first_hilbert_series(I, grading=(1,2,3))
-t^12 + t^10 + t^8 - t^4 - t^2 + 1
```

```
sage: first_hilbert_series((R*R.gens())^2, grading=range(1,10))
t^9 + t^8 + t^7 + t^6 + t^5 + t^4 + t^3 + t^2 + t + 1
```

```
sage: first_hilbert_series((matrix(R,3,R.gens())^2).list()*R).groebner_basis()]
```

**sage.rings.polynomial.hilbert.hilbert_poincare_series(I, grading=None)**

Return the Hilbert Poincaré series of the given monomial ideal.

**INPUT:**

- `I` – a monomial ideal (possibly defined in Singular)
- `grading` – (optional) a tuple of degree weights

**EXAMPLES:**

```python
sage: R = PolynomialRing(QQ,'x',9)
sage: I = [m.lm() for m in (matrix(R,3,R.gens())^2).list().R().groebner_basis()]*R
sage: hilbert_poincare_series(I)
t^7 - 3*t^6 + 2*t^5 + 2*t^4 - 2*t^3 + 6*t^2 + 5*t + 1)/(t^4 - 4*t^3 + 6*t^2 - 4*t + 1)
sage: hilbert_poincare_series((R*R.gens())^2, grading=range(1,10))
t^9 + t^8 + t^7 + t^6 + t^5 + t^4 + t^3 + t^2 + t + 1
```

3.1. Multivariate Polynomials and Polynomial Rings 431
The following example is taken from trac ticket #20145:

```
sage: n=4;m=11;P = PolynomialRing(QQ,n*m,"x"); x = P.gens(); M = Matrix(n,x)
sage: from sage.rings.polynomial.hilbert import first_hilbert_series
sage: I = P.ideal(M.minors(2))
sage: J = P*[m.lm() for m in I.groebner_basis()]
sage: hilbert_poincare_series(J).numerator()
120*t^3 + 135*t^2 + 30*t + 1
sage: hilbert_poincare_series(J).denominator().factor()
(t - 1)^14
```

This example exceeds the current capabilities of Singular:

```
sage: J.hilbert_numerator(algorithm='singular')
Traceback (most recent call last):
  ...:
RuntimeError: error in Singular function call 'hilb':
int overflow in hilb 1
```

### 3.1.12 Class to flatten polynomial rings over polynomial ring

For example $\mathbb{Q}[a',b'][x',y']$ flattens to $\mathbb{Q}[a',b',x',y']$.

**EXAMPLES:**

```
sage: R = QQ['x']['y']['s','t']['X']
sage: from sage.rings.polynomial.flatten import FlatteningMorphism
code
```

```
sage: R = QQ['a','b']['x','y','z']['t1','t2']
sage: from sage.rings.polynomial.flatten import FlatteningMorphism
```

```
Authors:
Vincent Delecroix, Ben Hutz (July 2016): initial implementation
```

**class** `sage.rings.polynomial.flatten.FlatteningMorphism`(domain)

**Bases:** sage.categories.morphism.Morphism

**EXAMPLES:**

```
sage: R = QQ['a','b']['x','y','z']['t1','t2']
sage: from sage.rings.polynomial.flatten import FlatteningMorphism
sage: f = FlatteningMorphism(R)
sage: f.codomain()
```

```
Multivariate Polynomial Ring in x, y, z, t1, t2 over Rational Field
```

```
sage: p = R(('a+b)*x + (a^2-b)*t2*(z+y)')
sage: p
```

```
(a^2 - b)*y + (a^2 - b)*z*t2 + (a + b)*x
```

```
sage: f(p)
a^2*y*t2 + a^2*z*t2 - b*y*t2 - b*z*t2 + a*x + b*x
```

(continues on next page)
 sage: f(p).parent()
Multivariate Polynomial Ring in a, b, x, y, z, t1, t2 over Rational Field

Also works when univariate polynomial ring are involved:

 sage: R = QQ['x']['y']['s','t']['X']
sage: from sage.rings.polynomial.flatten import FlatteningMorphism
sage: f = FlatteningMorphism(R)
sage: f.codomain()
Multivariate Polynomial Ring in x, y, s, t, X over Rational Field
 sage: p = R('((x^2 + 1) + (x+2)*y + x*y^3)*(s+t) + x*y*X')
sage: p
x*y*X + (x*y^3 + (x + 2)*y + x^2 + 1)*s + (x*y^3 + (x + 2)*y + x^2 + 1)*t
 sage: f(p)
x*y^3*s + x*y^3*t + x^2*s + x*y*s + x^2*t + x*y*t + x*y*X + 2*y*s + 2*y*t + s + t
 sage: f(p).parent()
Multivariate Polynomial Ring in x, y, s, t, X over Rational Field

inverse()
Return the inverse of this flattening morphism.

This is the same as calling section().

EXAMPLES:

 sage: f = QQ['x,y']['u,v'].flattening_morphism()
sage: f.inverse()
Unflattening morphism:
  From: Multivariate Polynomial Ring in x, y, u, v over Rational Field
  To: Multivariate Polynomial Ring in u, v over Multivariate Polynomial...
  in x, y over Rational Field

section()
Inverse of this flattening morphism.

EXAMPLES:

 sage: R = QQ['a','b','c']['x','y','z']
sage: from sage.rings.polynomial.flatten import FlatteningMorphism
sage: h = FlatteningMorphism(R)
sage: h.section()
Unflattening morphism:
  From: Multivariate Polynomial Ring in a, b, c, x, y, z over Rational Field
  To: Multivariate Polynomial Ring in x, y, z over Multivariate Polynomial...
  in a, b, c over Rational Field

 sage: R = ZZ['a']['b']['c']
sage: from sage.rings.polynomial.flatten import FlatteningMorphism
sage: FlatteningMorphism(R).section()
Unflattening morphism:
  From: Multivariate Polynomial Ring in a, b, c over Integer Ring
  To: Univariate Polynomial Ring in c over Univariate Polynomial Ring in b...
  over Univariate Polynomial Ring in a over Integer Ring
class sage.rings.polynomial.flatten.FractionSpecializationMorphism(domain, D)
Bases: sage.categories.morphism.Morphism

A specialization morphism for fraction fields over (stacked) polynomial rings

class sage.rings.polynomial.flatten.SpecializationMorphism(domain, D)
Bases: sage.categories.morphism.Morphism

Morphisms to specialize parameters in (stacked) polynomial rings

EXAMPLES:

```
sage: R.<c> = PolynomialRing(QQ)
sage: S.<x,y,z> = PolynomialRing(R)
sage: D = dict({c:1})
sage: from sage.rings.polynomial.flatten import SpecializationMorphism
sage: f = SpecializationMorphism(S, D)
sage: g = f(x^2 + c*y^2 - z^2); g
x^2 + y^2 - z^2
sage: g.parent()
Multivariate Polynomial Ring in x, y, z over Rational Field
```

```
sage: R.<c> = PolynomialRing(QQ)
sage: S.<z> = PolynomialRing(R)
sage: from sage.rings.polynomial.flatten import SpecializationMorphism
sage: xi = SpecializationMorphism(S, {c:0}); xi
Specialization morphism:
  From: Univariate Polynomial Ring in z over Univariate Polynomial Ring in c → over Rational Field
  To:  Univariate Polynomial Ring in z over Rational Field
sage: xi(z^2+c)
z^2
```

```
sage: R1.<u,v> = PolynomialRing(QQ)
sage: R2.<a,b,c> = PolynomialRing(R1)
sage: S.<x,y,z> = PolynomialRing(R2)
sage: D = dict({a:1, b:2, x:0, u:1})
sage: from sage.rings.polynomial.flatten import SpecializationMorphism
sage: xi = SpecializationMorphism(S, D); xi
Specialization morphism:
  From: Multivariate Polynomial Ring in x, y, z over Multivariate Polynomial Ring in a, b, c over Multivariate Polynomial Ring in u, v over Rational Field
  To:  Multivariate Polynomial Ring in y, z over Univariate Polynomial Ring in c → over Rational Field
sage: xi(a*(x*z+y^2)*u+b*v*u*(x*z+y^2)*y^2*c+c*y^2*z^2)
2*v*c*y^4 + c*y^2*z^2 + y^2
```

class sage.rings.polynomial.flatten.UnflatteningMorphism(domain, codomain)
Bases: sage.categories.morphism.Morphism

Inverses for FlatteningMorphism

EXAMPLES:

```
sage: R = QQ['c','x','y','z']
sage: S = QQ['c']['x','y','z']
```
```
sage: from sage.rings.polynomial.flatten import UnflatteningMorphism
sage: f = UnflatteningMorphism(R, S)
```
```
sage: g = f(R('x^2 + c*y^2 - z^2')); g
x^2 + c*y^2 - z^2
sage: g.parent()
Multivariate Polynomial Ring in x, y, z over Univariate Polynomial Ring in c over Rational Field
```
```
sage: R = QQ['a','b', 'x','y']
sage: S = QQ['a','b'][['x','y']]
sage: from sage.rings.polynomial.flatten import UnflatteningMorphism
sage: UnflatteningMorphism(R, S)
Unflattening morphism:
  From: Multivariate Polynomial Ring in a, b, x, y over Rational Field
  To:  Multivariate Polynomial Ring in x, y over Multivariate Polynomial Ring in a, b over Rational Field
```

### 3.1.13 Monomials

`sage.rings.monomials.monomials(v, n)`

Given two lists `v` and `n`, of exactly the same length, return all monomials in the elements of `v`, where variable `i` (i.e., `v[i]`) in the monomial appears to degree strictly less than `n[i]`.

**INPUT:**

- `v` – list of ring elements
- `n` – list of integers

**EXAMPLES:**

```
sage: monomials([x], [3])
[1, x, x^2]
sage: R.<x,y,z> = QQ[]
sage: monomials([x,y], [5,5])
[1, y, y^2, y^3, y^4, x, x*y, x*y^2, x*y^3, x*y^4, x^2, x^2*y, x^2*y^2, x^2*y^3, x^2*y^4, x^3, x^3*y, x^3*y^2, x^3*y^3, x^3*y^4, x^4, x^4*y, x^4*y^2, x^4*y^3, x^4*y^4]
sage: monomials([x,y,z], [2,3,2])
[1, z, y, y*z, y^2, y^2*z, x, x*z, x*y, x*y*z, x*y^2, x*y^2*z]
```

### 3.2 Classical Invariant Theory

#### 3.2.1 Classical Invariant Theory

This module lists classical invariants and covariants of homogeneous polynomials (also called algebraic forms) under the action of the special linear group. That is, we are dealing with polynomials of degree $d$ in $n$ variables. The special linear group $SL(n, C)$ acts on the variables $(x_1, \ldots, x_n)$ linearly.

. MATH:
The linear action on the variables transforms a polynomial $p$ generally into a different polynomial $gp$. We can think of it as an action on the space of coefficients in $p$. An invariant is a polynomial in the coefficients that is invariant under this action. A covariant is a polynomial in the coefficients and the variables $(x_1, \ldots, x_n)$ that is invariant under the combined action.

For example, the binary quadratic $p(x, y) = ax^2 + bxy + cy^2$ has as its invariant the discriminant $\text{disc}(p) = b^2 - 4ac$. This means that for any $SL(2, \mathbb{C})$ coordinate change

\[
\begin{pmatrix} x' \\
y' \end{pmatrix} = \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix} \begin{pmatrix} x \\
y \end{pmatrix}, \quad \alpha \delta - \beta \gamma = 1
\]

the discriminant is invariant, $\text{disc}(p(x', y')) = \text{disc}(p(x, y))$.

To use this module, you should use the factory object `invariant_theory`. For example, take the quartic:

```
sage: R.<x,y> = QQ[]
sage: q = x^4 + y^4
sage: quartic = invariant_theory.binary_quartic(q); quartic
Binary quartic with coefficients (1, 0, 0, 0, 1)
```

One invariant of a quartic is known as the Eisenstein D-invariant. Since it is an invariant, it is a polynomial in the coefficients (which are integers in this example):

```
sage: quartic.EisensteinD()
1
```

One example of a covariant of a quartic is the so-called $g$-covariant (actually, the Hessian). As with all covariants, it is a polynomial in $x, y$ and the coefficients:

```
sage: quartic.g_covariant()
-x^2*y^2
```

As usual, use tab completion and the online help to discover the implemented invariants and covariants.

In general, the variables of the defining polynomial cannot be guessed. For example, the zero polynomial can be thought of as a homogeneous polynomial of any degree. Also, since we also want to allow polynomial coefficients we cannot just take all variables of the polynomial ring as the variables of the form. This is why you will have to specify the variables explicitly if there is any potential ambiguity. For example:

```
sage: invariant_theory.binary_quartic(R.zero(), [x,y])
Binary quartic with coefficients (0, 0, 0, 0, 0)
```

```
sage: invariant_theory.binary_quartic(x^4, [x,y])
Binary quartic with coefficients (0, 0, 0, 0, 1)
```

```
sage: R.<x,y,t> = QQ[]
sage: invariant_theory.binary_quartic(x^4 + y^4 + t*x^2*y^2, [x,y])
Binary quartic with coefficients (1, 0, t, 0, 1)
```

Finally, it is often convenient to use inhomogeneous polynomials where it is understood that one wants to homogenize them. This is also supported, just define the form with an inhomogeneous polynomial and specify one less variable:
```python
sage: R.<x,t> = QQ[]
sage: invariant_theory.binary_quartic(x^4 + 1 + t*x^2, [x])
Binary quartic with coefficients (1, 0, t, 0, 1)
```

REFERENCES:

- Wikipedia article Glossary_of_invariant_theory

AUTHORS:

- Volker Braun (2013-01-24): initial version
- Jesper Noordsij (2018-05-18): support for binary quintics added

```python
class sage.rings.invariants.invariant_theory.AlgebraicForm(n, d, polynomial, *args, **kwds)
Bases: sage.rings.invariants.invariant_theory.FormsBase
```

The base class of algebraic forms (i.e. homogeneous polynomials).
You should only instantiate the derived classes of this base class.
Derived classes must implement `coeffs()` and `scaled_coeffs()`

INPUT:

- `n` – The number of variables.
- `d` – The degree of the polynomial.
- `polynomial` – The polynomial.
- `*args` – The variables, as a single list/tuple, multiple arguments, or `None` to use all variables of the polynomial.

Derived classes must implement the same arguments for the constructor.

EXAMPLES:

```python
sage: from sage.rings.invariants.invariant_theory import AlgebraicForm
sage: R.<x,y> = QQ[]
sage: p = x^2 + y^2
sage: AlgebraicForm(2, 2, p).variables()
(x, y)
sage: AlgebraicForm(2, 2, p, None).variables()
(x, y)
sage: AlgebraicForm(3, 2, p).variables()
(x, y, None)
sage: AlgebraicForm(3, 2, p, None).variables()
(x, y, None)
```

```python
sage: from sage.rings.invariants.invariant_theory import AlgebraicForm
sage: R.<x,y,s,t> = QQ[]
sage: p = s*x^2 + t*y^2
sage: AlgebraicForm(2, 2, p, [x,y]).variables()
(x, y)
sage: AlgebraicForm(2, 2, p, x,y).variables()
(x, y)
sage: AlgebraicForm(3, 2, p, [x,y,None]).variables()
(x, y, None)
```

(continues on next page)
sage: AlgebraicForm(3, 2, p, x,y,None).variables()
(x, y, None)

sage: AlgebraicForm(2, 1, p, [x,y]).variables()
Traceback (most recent call last):
...  
ValueError: polynomial is of the wrong degree

sage: AlgebraicForm(2, 2, x^2+y, [x,y]).variables()
Traceback (most recent call last):
...  
ValueError: polynomial is not homogeneous

coefficients()
Alias for coeffs().
See the documentation for coeffs() for details.

EXAMPLES:

sage: R.<a,b,c,d,e,f,g, x,y,z> = QQ[]
sage: p = a*x^2 + b*y^2 + c*z^2 + d*x*y + e*x*z + f*y*z
sage: q = invariant_theory.quadratic_form(p, x,y,z)
sage: q.coefficients()
(a, b, c, d, e, f)
sage: q.coeffs()
(a, b, c, d, e, f)

form()
Return the defining polynomial.

OUTPUT:
The polynomial used to define the algebraic form.

EXAMPLES:

sage: R.<x,y> = QQ[]
sage: quartic = invariant_theory.binary_quartic(x^4+y^4)
sage: quartic.form()
x^4 + y^4

homogenized(var='h')
Return form as defined by a homogeneous polynomial.

INPUT:

• var – either a variable name, variable index or a variable (default: 'h').

OUTPUT:
The same algebraic form, but defined by a homogeneous polynomial.

EXAMPLES:
sage: T.<t> = QQ[]
sage: quadratic = invariant_theory.binary_quadratic(t^2 + 2*t + 3)
sage: quadratic
Binary quadratic with coefficients (1, 3, 2)
sage: quadratic.homogenized()
Binary quadratic with coefficients (1, 3, 2)
sage: quadratic == quadratic.homogenized()
True
sage: quadratic.form()
t^2 + 2*t + 3
sage: quadratic.homogenized().form()
t^2 + 2*t*h + 3*h^2
sage: R.<x,y,z> = QQ[]
sage: quadratic = invariant_theory.ternary_quadratic(x^2 + 1, [x,y])
sage: quadratic.homogenized().form()
x^2 + h^2
sage: R.<x> = QQ[]
sage: quintic = invariant_theory.binary_quintic(x^4 + 1, x)
sage: quintic.homogenized().form()
x^4*h + h^5

polynomial()

Return the defining polynomial.

OUTPUT:

The polynomial used to define the algebraic form.

EXAMPLES:

sage: R.<x,y> = QQ[]
sage: quartic = invariant_theory.binary_quartic(x^4+y^4)
sage: quartic.form()
x^4 + y^4
sage: quartic.polynomial()
x^4 + y^4

transformed(g)

Return the image under a linear transformation of the variables.

INPUT:

- g – a $GL(n, \mathbb{C})$ matrix or a dictionary with the variables as keys. A matrix is used to define the linear transformation of homogeneous variables, a dictionary acts by substitution of the variables.

OUTPUT:

A new instance of a subclass of AlgebraicForm obtained by replacing the variables of the homogeneous polynomial by their image under g.

EXAMPLES:

sage: R.<x,y,z> = QQ[]
sage: cubic = invariant_theory.ternary_cubic(x^3 + 2*y^3 + 3*z^3 + 4*x*y*z)
sage: cubic.transformed({x:y, y:z, z:x}).form()
\[3x^3 + y^3 + 4xyz + 2z^3\]

```python
cyc = matrix([[0, 1, 0],[0, 0, 1],[1, 0, 0]])
cubic.transformed(cyc) == cubic.transformed({x:y, y:z, z:x})
```

```python
g = matrix(QQ, [[1, 0, 0], [-1, 1, -3], [-5, -5, 16]])
cubic.transformed(g).transformed(g.inverse()) == cubic
```

```python
class sage.rings.invariants.invariant_theory.BinaryQuartic(n, d, polynomial, *args)

Bases: sage.rings.invariants.invariant_theory.AlgebraicForm

Invariant theory of a binary quartic.

You should use the invariant_theory factory object to construct instances of this class. See binary_quartic() for details.

**EisensteinD()**

One of the Eisenstein invariants of a binary quartic.

**OUTPUT:**

The Eisenstein D-invariant of the quartic.

\[f(x) = a_0x_1^4 + 4a_1x_0x_1^3 + 6a_2x_0^2x_1^2 + 4a_3x_0^3x_1 + a_4x_0^4\]

\[\Rightarrow D(f) = a_0a_4 + 3a_2^2 - 4a_1a_3\]

**EXAMPLES:**

```python
sage: R.<a0, a1, a2, a3, a4, x0, x1> = QQ[]
sage: f = a0*x1^4+4*a1*x0*x1^3+6*a2*x0^2*x1^2+4*a3*x0^3*x1+a4*x0^4
sage: inv = invariant_theory.binary_quartic(f, x0, x1)
sage: inv.EisensteinD()
3*a2^2 - 4*a1*a3 + a0*a4
```

**EisensteinE()**

One of the Eisenstein invariants of a binary quartic.

**OUTPUT:**

The Eisenstein E-invariant of the quartic.

\[f(x) = a_0x_1^4 + 4a_1x_0x_1^3 + 6a_2x_0^2x_1^2 + 4a_3x_0^3x_1 + a_4x_0^4\]

\[\Rightarrow E(f) = a_0a_4^2 + a_1^2a_4 - a_0a_2a_4 - 2a_1a_2a_3 + a_3^2\]

**EXAMPLES:**

```python
sage: R.<a0, a1, a2, a3, a4, x0, x1> = QQ[]
sage: f = a0*x1^4+4*a1*x0*x1^3+6*a2*x0^2*x1^2+4*a3*x0^3*x1+a4*x0^4
sage: inv = invariant_theory.binary_quartic(f, x0, x1)
sage: inv.EisensteinE()
a2^3 - 2*a1*a2*a3 + a0*a3^2 + a1^2*a4 - a0*a2*a4
```

**coeffs()**

The coefficients of a binary quartic.
Given

\[ f(x) = a_0 x_1^4 + a_1 x_0 x_1^3 + a_2 x_0^2 x_1^2 + a_3 x_0^3 x_1 + a_4 x_0^4 \]

this function returns \( a = (a_0, a_1, a_2, a_3, a_4) \)

**EXAMPLES:**

```sage
sage: R.<a0, a1, a2, a3, a4, x0, x1> = QQ[]
sage: p = a0*x1^4 + a1*x1^3*x0 + a2*x1^2*x0^2 + a3*x1*x0^3 + a4*x0^4
sage: quartic = invariant_theory.binary_quartic(p, x0, x1)
sage: quartic.coeffs()
(a0, a1, a2, a3, a4)
```

$$\Rightarrow D(f) = \frac{1}{144} \left( \frac{\partial^2 f}{\partial x \partial x} \right)$$

**EXAMPLES:**

```sage
sage: R.<a0, a1, a2, a3, a4, x> = QQ[]
sage: p = a0 + a1*x + a2*x^2 + a3*x^3 + a4*x^4
sage: quartic = invariant_theory.binary_quartic(p, x)
sage: quartic.coeffs()
(a0, a1, a2, a3, a4)
```

**g_covariant()**

The g-covariant of a binary quartic.

**OUTPUT:**

The g-covariant of the quartic.

\[ f(x) = a_0 x_1^4 + 4a_1 x_0 x_1^3 + 6a_2 x_0^2 x_1^2 + 4a_3 x_0^3 x_1 + a_4 x_0^4 \]

\[ \Rightarrow D(f) = \frac{1}{144} \left( \frac{\partial^2 f}{\partial x \partial x} \right) \]

**EXAMPLES:**

```sage
sage: R.<a0, a1, a2, a3, a4, x, y> = QQ[]
sage: p = a0*x^4+4*a1*x^3*y+6*a2*x^2*y^2+4*a3*x*y^3+a4*y^4
sage: inv = invariant_theory.binary_quartic(p, x, y)
sage: g = inv.g_covariant(); g
a1^2*x^4 - a0*a2*x^4 + 2*a1*a2*x^3*y - 2*a0*a3*x^3*y + 3*a2^2*x^2*y^2
- 2*a1*a3*x^2*y^2 - a0*a4*x^2*y^2 + 2*a2*a3*x*y^3
- 2*a1*a4*x*y^3 + a3^2*y^4 - a2*a4*y^4
```

**h_covariant()**

The h-covariant of a binary quartic.

**OUTPUT:**

The h-covariant of the quartic.

\[ f(x) = a_0 x_1^4 + 4a_1 x_0 x_1^3 + 6a_2 x_0^2 x_1^2 + 4a_3 x_0^3 x_1 + a_4 x_0^4 \]

\[ \Rightarrow D(f) = \frac{1}{144} \left( \frac{\partial^2 f}{\partial x \partial x} \right) \]
EXAMPLES:

```
sage: R.<a0, a1, a2, a3, a4, x, y> = QQ[]
sage: p = a0*x^4+4*a1*x^3*y+6*a2*x^2*y^2+4*a3*x*y^3+a4*y^4
sage: inv = invariant_theory.binary_quartic(p, x, y)
sage: h = inv.h_covariant(); h
-2*a1^3*x^6 + 3*a0*a1*a2*x^6 - a0^2*a3*x^6 - 6*a1^2*a2*x^5*y + 9*a0*a2^2*x^5*y
- 2*a0*a1*a3*x^5*y - a0^2*a4*x^5*y - 10*a1^2*a3*x^4*y^2 + 15*a0*a2*a3*x^4*y^2
- 5*a0*a1*a4*x^4*y^2 + 10*a0*a3^2*x^3*y^3 - 10*a1^2*a4*x^3*y^3
- 10*a1*a3^2*x^2*y^4 + 15*a0*a3*a4*x^2*y^4 + 5*a0*a3^2*x^2*y^4
+ 6*a2*a3^2*x*y^5 - 9*a2^2*a4*x*y^5 + 2*a1*a3*a4*x*y^5 + a0*a4^2*x*y^5
+ 2*a3^3*y^6 - 3*a2^2*a3*a4*y^6 + a1*a4^2*y^6
sage: inv_inhomogeneous = invariant_theory.binary_quartic(p.subs(y=1), x)
sage: inv_inhomogeneous.h_covariant()
-2*a1^3*x^6 + 3*a0*a1*a2*x^6 - a0^2*a3*x^6 - 6*a1^2*a2*x^5 + 9*a0*a2^2*x^5
- 2*a0*a1*a3*x^5 - a0^2*a4*x^5 - 10*a1^2*a3*x^4 + 15*a0*a2*a3*x^4
- 5*a0*a1*a4*x^4 + 10*a0*a3^2*x^3 - 10*a1^2*a4*x^3 + 10*a1*a3^2*x^2
- 15*a1*a2^2*a4*x^2 + 5*a0*a3^2*a4*x^2 + 6*a2*a3^2*x - 9*a2^2*a4*x
+ 2*a1*a3*a4*x + a0*a4^2*x + 2*a3^3 - 3*a2^2*a3*a4 + a1*a4^2
sage: g = inv.g_covariant()
sage: h == 1/8 * (p.derivative(x)*g.derivative(y)-p.derivative(y)*g.derivative(x))
True
```

**monomials()**

List the basis monomials in the form.

**OUTPUT:**

A tuple of monomials. They are in the same order as `coeffs()`.

**EXAMPLES:**

```
sage: R.<x,y> = QQ[]
sage: quartic = invariant_theory.binary_quartic(x^4+y^4)
sage: quartic.monomials()
(y^4, x*y^3, x^2*y^2, x^3*y, x^4)
```

**scaled_coeffs()**

The coefficients of a binary quartic.

Given

\[ f(x) = a_0 x_1^4 + 4a_1 x_0 x_1^3 + 6a_2 x_0^2 x_1^2 + 4a_3 x_0^3 x_1 + a_4 x_0^4 \]

this function returns \(a = (a_0, a_1, a_2, a_3, a_4)\)

**EXAMPLES:**

```
sage: R.<a0, a1, a2, a3, a4, x0, x1> = QQ[]
sage: quartic = a0*x1^4 + 4*a1*x1^3*x0 + 6*a2*x1^2*x0^2 + 4*a3*x1^3*x0^3 + a4*x0^4
sage: inv = invariant_theory.binary_quartic(quartic, x0, x1)
sage: inv.scaled_coeffs()
(a0, a1, a2, a3, a4)
```
(continues on next page)
\begin{verbatim}
sage: R.<a0, a1, a2, a3, a4, x> = QQ[]
sage: quartic = a0 + 4*a1*x + 6*a2*x^2 + 4*a3*x^3 + a4*x^4
sage: inv = invariant_theory.binary_quartic(quartic, x)
sage: inv.scaled_coeffs()
(a0, a1, a2, a3, a4)
\end{verbatim}

class \texttt{sage.rings.invariants.invariant\_theory.BinaryQuintic}(n, d, \textit{polynomial}, *\textit{args})

Bases: \texttt{sage.rings.invariants.invariant\_theory.AlgebraicForm}

Invariant theory of a binary quintic form.

You should use the \texttt{invariant\_theory} factory object to construct instances of this class. See \texttt{binary\_quintic()} for details.

REFERENCES:

For a description of all invariants and covariants of a binary quintic, see section 73 of \cite{Cle1872}.

\textbf{A\_invariant()}

Return the invariant $A$ of a binary quintic.

OUTPUT:

The $A$-invariant of the binary quintic.

EXAMPLES:

\begin{verbatim}
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5^x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.A_invariant()
4/625*a2^2*a3^2 - 12/625*a1*a3^3 - 12/625*a2^3*a4 + 38/625*a1*a2*a3*a4 + 6/125*a0*a3^2*a4 - 18/625*a1^2*a4^2 - 16/125*a0*a2*a4^2 + 6/125*a1*a2^2*a5 - 16/125*a1^2*a3*a5 - 2/25*a0*a2*a3*a5 + 4/5*a0*a4*a5 - 2*a0^2*a5^2
\end{verbatim}

\textbf{B\_invariant()}

Return the invariant $B$ of a binary quintic.

OUTPUT:

The $B$-invariant of the binary quintic.

EXAMPLES:

\begin{verbatim}
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5^x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.B_invariant()
1/1562500*a2^4*a3^4 - 3/781250*a1*a2^2*a3^5 + 9/1562500*a1^2*a3^6 - 3/781250*a2^5*a3^2*a4 + 37/1562500*a1*a2^3*a3^3*a4 - 57/1562500*a1^2*a2*a3^4*a4 + 3/312500*a0*a2^2*a3^4*a4 + 8/625*a0^2*a1^2*a4^2*a5^2 - 4/125*a0^3*a2*a4^2*a5^2 + 16/3125*a0^4*a5^3 + 4/125*a0*a1^3*a2*a5^3 - 6/125*a0^2*a1*a2^2*a5^3 - 4/125*a0*a1^3*a2*a5^3 + 2/25*a0^3*a2*a5^3
\end{verbatim}
C_invariant()

Return the invariant $C$ of a binary quintic.

OUTPUT:

The $C$-invariant of the binary quintic.

EXAMPLES:

```python
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.C_invariant()
-3/1953125000*a2^6*a3^6 + 27/1953125000*a1*a2^4*a3^7
- 249/781250000*a1^2*a2^2*a3^8 - 3/78125000*a0*a2^3*a3^8
+ 3/976562500*a1^3*a3^9 + 27/156250000*a0*a1*a2*a3^9
... + 192/15625*a0^2*a1^3*a2^2*a3^2*a5^4 - 36/3125*a0^3*a1*a2*a3^3*a5^4
+ 24/15625*a0^2*a1^4*a3^2*a5^4 - 24/3125*a0^3*a1^2*a2*a3^2*a5^4
+ 6/625*a0^4*a2^2*a3^2*a5^4
```

H_covariant(as_form=False)

Return the covariant $H$ of a binary quintic.

INPUT:

- `as_form` – if `as_form` is False, the result will be returned as polynomial (default). If it is True the result is returned as an object of the class `AlgebraicForm`.

OUTPUT:

The $H$-covariant of the binary quintic as polynomial or as binary form.

EXAMPLES:

```python
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.H_covariant()
-2/25*a4^2*x0^6 + 1/5*a3*a5*x0^6 - 3/25*a3*a4*x0^5*x1
+ 3/5*a2*a5*x0^5*x1 - 3/25*a3^2*x0^4*x1^2 + 3/25*a2*a4*x0^4*x1^2
+ 6/5*a1*a5*x0^4*x1^2 + 4/25*a2*a3*x0^3*x1^3 + 14/25*a1*a4*x0^3*x1^3
... + 2*a0*a5*x0^3*x1^3 - 3/25*a2^2*x0^2*x1^4 + 3/25*a1*a3*x0^2*x1^4
+ 6/5*a0*a4*x0^2*x1^4 - 3/25*a1^2*a2*x0^1*x1^5 + 3/5*a0*a3*x0^1*x1^5
+ 1/5*a0*a2*a1^2*x1^6
```

R_invariant()

Return the invariant $R$ of a binary quintic.

OUTPUT:

The $R$-invariant of the binary quintic.

EXAMPLES:
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]

sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 +
   → a5*x0^5

sage: quintic = invariant_theory.binary_quintic(p, x0, x1)

sage: quintic.R_invariant()
3/3906250000000*a1^2*a2^5*a3^11 - 3/976562500000*a0*a2^6*a3^11
- 51/781250000000*a1^3*a2^3*a3^12 + 27/976562500000*a0*a1*a2^4*a3^12
+ 27/1953125000000*a1^4*a2*a3^13 - 81/156250000000*a0*a1^2*a2^2*a3^13
...+
+ 384/9765625*a0*a1^10*a5^7 - 192/390625*a0^2*a1^8*a2*a5^7
+ 192/78125*a0^3*a1^6*a2^2*a5^7 - 96/15625*a0^4*a1^4*a2^3*a5^7
+ 24/3125*a0^5*a1^2*a2^4*a5^7 - 12/3125*a0^6*a2^5*a5^7

**T_covariant** *(as_form=False)*

Return the covariant \( T \) of a binary quintic.

**INPUT:**

- **as_form** – if `as_form` is `False`, the result will be returned as polynomial (default). If it is `True` the result is returned as an object of the class `AlgebraicForm`.

**OUTPUT:**

The \( T \)-covariant of the binary quintic as polynomial or as binary form.

**EXAMPLES:**

sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]

sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 +
   → a5*x0^5

sage: quintic = invariant_theory.binary_quintic(p, x0, x1)

sage: quintic.T_covariant()
2/125*a4^3*x0^9 - 3/50*a3*a4*a5*x0^9 + 1/10*a2*a5^2*x0^9
+ 9/250*a3^2*a4*a5*x0^8*x1 - 3/25*a3^2*a5^2*x0^8*x1 + 1/50*a2^2*a4*a5*x0^8*x1
+ 2/5*a1*a5^2*x0^8*x1 + 3/250*a3^2*a4*x0^7*x1^2 + 8/125*a2^2*a4^2*x0^7*x1^2
...+
+ 11/25*a0*a1*a4*x0^2*x1^7 - a0*a1^2*a5*x0^2*x1^7 - 9/250*a1^3*a5*x0^2*x1^7
+ 3/25*a0^2*a2^2*x0^2*x1^8 - 1/50*a0*a1*a3*x0^2*x1^8 - 2/5*a0^2*a4*x0^2*x1^8
- 2/125*a1^3*a3*x1^9 + 3/50*a0*a1*a2*x1^9 - 1/10*a0^2*a3*x1^9

**alpha_covariant** *(as_form=True)*

Binary nonic given by ...

**EXAMPLES:**

### 3.2. Classical Invariant Theory

445
\begin{verbatim}
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.alpha_covariant()
1/2500*a2^2*a3^3*x0 - 3/2500*a1*a3^4*x0 - 1/625*a2^3*a3*a4*x0 + 3/625*a1*a2*a3^2*a4*x0 + 3/625*a0*a3^3*a4*x0 + 2/625*a1*a2^2*a4^2*x0 - 6/625*a1^2*a3*a4^2*x0 - 12/625*a0*a2*a3*a4^2*x0 + 24/625*a0*a1*a4^3*x0 - 12/625*a1^2*a2*a3*a5*x1 - 1/125*a0*a2^2*a3*a5*x1 + 8/125*a0*a1*a3^2*a5*x1 + 24/625*a1^3*a4*a5*x1 - 8/125*a0*a1*a2*a4*a5*x1 - 4/25*a0^2*a3*a4*a5*x1 - 4/25*a0*a1^2*a5^2*x1 + 2/5*a0^2*a2*a5^2*x1

sage: quintic.alpha_covariant(as_form=True)
Binary monic given by ...

arithmetic_invariants()
Return a set of generating arithmetic invariants of a binary quintic.

An arithmetic invariants is an invariant whose coefficients are integers for a general binary quintic. They are linear combinations of the Clebsch invariants, such that they still generate the ring of invariants.

OUTPUT:

The arithmetic invariants of the binary quintic. They are given by

\[
\begin{align*}
I_4 &= 2^{-1} \cdot 5^4 \cdot A \\
I_8 &= 5^5 \cdot (2^{-1} \cdot 47 \cdot A^2 - 2^2 \cdot B) \\
I_{12} &= 5^{10} \cdot (2^{-1} \cdot 3 \cdot A^3 - 2^5 \cdot 3^{-1} \cdot C) \\
I_{18} &= 2^8 \cdot 3^{-1} \cdot 5^{15} \cdot R
\end{align*}
\]

where A, B, C and R are the \texttt{BinaryQuintic.clebsch_invariants}().

EXAMPLES:

\begin{verbatim}
sage: R.<x0, x1> = QQ[]
sage: p = 2*x1^5 + 4*x1^4*x0 + 5*x1^3*x0^2 + 7*x1^2*x0^3 - 11*x1*x0^4 + x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.arithmetic_invariants()
{'I12': -1156502613073152, 'I18': -12712872348048797642752, 'I4': -138016, 'I8': 14164936192}
\end{verbatim}

We can check that the coefficients of the invariants have no common divisor for a general quintic form:

\begin{verbatim}
sage: R.<a0,a1,a2,a3,a4,a5,x0,x1> = QQ[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: invs = quintic.arithmetic_invariants()
sage: [invs[x].content() for x in invs]
[1, 1, 1, 1]
\end{verbatim}

\texttt{beta_covariant(as_form=False)}
Return the covariant \(\beta\) of a binary quintic.
\end{verbatim}
INPUT:

- `as_form` – if `as_form` is `False`, the result will be returned as polynomial (default). If it is `True` the result is returned as an object of the class `AlgebraicForm`.

OUTPUT:

The $\beta$-covariant of the binary quintic as polynomial or as binary form.

EXAMPLES:

```sage
R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5
quintic = invariant_theory.binary_quintic(p, x0, x1)
quintic.beta_covariant()
-1/62500*a2^3*a3^4*x0 + 9/125000*a1*a2*a3^5*x0 - 27/125000*a0*a3^6*x0
+ 13/125000*a2^4*a3^2*a4*x0 - 31/625000*a0*a2^2*a3^3*a4*x0
- 3/62500*a1^2*a3^4*a4*x0 + 27/15625*a0*a2*a3^4*a4*x0
... - 16/125*a0^2*a1*a3^2*a5^2*x1 - 28/625*a0*a1*a3*a4*a5^2*x1
+ 6/125*a0^2*a1*a2*a4*a5^2*x1 + 8/25*a0^3*a3*a4*a5^2*x1
+ 4/25*a0^2*a1^2*a5^3*x1 - 2/5*a0*a3*a2*a5^3*x1
sage: quintic.beta_covariant(as_form=True)
Binary monic given by ...
```

canonical_form(reduce_gcd=False)

Return a canonical representative of the quintic.

Given a binary quintic $f$ with coefficients in a field $K$, returns a canonical representative of the $GL(2, \bar{K})$-orbit of the quintic, where $\bar{K}$ is an algebraic closure of $K$. This means that two binary quintics $f$ and $g$ are $GL(2, \bar{K})$-equivalent if and only if their canonical forms are the same.

INPUT:

- `reduce_gcd` – If set to `True`, then a variant of this canonical form is computed where the coefficients are coprime integers. The obtained form is then unique up to multiplication by a unit. See also `binary_quintic_from_invariants()`.

OUTPUT:

A canonical $GL(2, \bar{K})$-equivalent binary quintic.

EXAMPLES:

```sage
R.<x0, x1> = QQ[]
p = 2*x1^5 + 4*x1^4*x0 + 5*x1^3*x0^2 + 7*x1^2*x0^3 - 11*x1*x0^4 + x0^5
f = invariant_theory.binary_quintic(p, x0, x1)
g = matrix(QQ, [[11,5],[7,2]])
gf = f.transformed(g)
f.canonical_form() == gf.canonical_form()
True
h = f.canonical_form(reduce_gcd=True)
gcd(h.coeffs())
1
```

clebsch_invariants(as_tuple=False)

Return the invariants of a binary quintic as described by Clebsch.
The following invariants are returned: $A$, $B$, $C$ and $R$.

OUTPUT:

The Clebsch invariants of the binary quintic.

EXAMPLES:

```python
sage: R.<x0, x1> = QQ[]
sage: p = 2*x1^5 + 4*x1^4*x0 + 5*x1^3*x0^2 + 7*x1^2*x0^3 + 11*x1*x0^4 + x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.clebsch_invariants()
{'A': -276032/625,
'B': 4983526016/390625,
'C': -247056495846408/244140625,
'R': -148978972828696847376/30517578125}
sage: quintic.clebsch_invariants(as_tuple=True)
(-276032/625,
4983526016/390625,
-247056495846408/244140625,
-148978972828696847376/30517578125)
```

coeffs()

The coefficients of a binary quintic.

Given

$$f(x) = a_0 x_1^5 + a_1 x_0 x_1^4 + a_2 x_0^2 x_1^3 + a_3 x_0^3 x_1^2 + a_4 x_0^4 x_1 + a_5 x_1^5$$

this function returns $a = (a_0, a_1, a_2, a_3, a_4, a_5)$

EXAMPLES:

```python
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.coeffs()
(a0, a1, a2, a3, a4, a5)
sage: R.<a0, a1, a2, a3, a4, a5, x> = QQ[]
sage: p = a0 + a1*x + a2*x^2 + a3*x^3 + a4*x^4 + a5*x^5
sage: quintic = invariant_theory.binary_quintic(p, x)
sage: quintic.coeffs()
(a0, a1, a2, a3, a4, a5)
```

delta_covariant(as_form=False)

Return the covariant $\delta$ of a binary quintic.

INPUT:

- `as_form` – if `as_form` is `False`, the result will be returned as polynomial (default). If it is `True` the result is returned as an object of the class `AlgebraicForm`.

OUTPUT:

The $\delta$-covariant of the binary quintic as polynomial or as binary form.

EXAMPLES:
```python
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.delta_covariant()
1/1562500000*a2^6*a3^7*x0 - 9/1562500000*a1*a2^4*a3^8*x0 + 9/625000000*a1^2*a2*a3^9*x0 - 9/1562500000*a0*a2^3*a3^9*x0 - 81/1562500000*a1^3*a3^10*x0 + 64/3125*a0^3*a2^2*a3^5*x1 - 12/625*a0^4*a2*a3^5*x1 + 16/3125*a0^3*a2^2*a3^5*x1 - 16/625*a0^4*a2^2*a3^5*x1 + 4/125*a0^5*a2^2*a3^5*x1
sage: quintic.delta_covariant(as_form=True)
Binary monic given by ...
```

classmethod from_invariants(invariants, x, z, *args, **kwargs)

Construct a binary quintic from its invariants.

This function constructs a binary quintic whose invariants equal the ones provided as argument up to scaling.

INPUT:

• invariants – A list or tuple of invariants that are used to reconstruct the binary quintic.

OUTPUT:

A BinaryQuintic.

EXAMPLES:

```python
sage: R.<x,y> = QQ[]
sage: from sage.rings.invariants.invariant_theory import BinaryQuintic
sage: BinaryQuintic.from_invariants([3,6,12], x, y)
Binary quintic with coefficients (0, 1, 0, 0, 1, 0)
```

gamma_covariant(as_form=False)

Return the covariant \( \gamma \) of a binary quintic.

INPUT:

• as_form – if as_form is False, the result will be returned as polynomial (default). If it is True the result is returned as an object of the class AlgebraicForm.

OUTPUT:

The \( \gamma \)-covariant of the binary quintic as polynomial or as binary form.

EXAMPLES:

```python
sage: R.<x,y> = QQ[]
sage: from sage.rings.invariants.invariant_theory import BinaryQuintic
sage: BinaryQuintic.gamma_covariant(as_form=False)
1/1562500000*a2^5*a3^6*x0 - 9/625000000*a1*a2^3*a3^7*x0 + 27/312500000*a1^2*a2^2*a3^8*x0 + 27/312500000*a0*a2^2*a3^8*x0
```

(continues on next page)
- $81/312500000*a0*a1*a3^9*x0 - 19/312500000*a2^6*a3^4*a4*x0$
- $32/3125*a0^2*a1^3*a2^2*a5^4*x1 + 6/625*a0^3*a1*a2^3*a5^4*x1$
- $8/3125*a0^2*a1*a3^5*a4^4*x1 + 8/625*a0^3*a1^2*a2*a3*a5^4*x1$
- $2/125*a0^4*a2*a3^5*a5^4*x1$

```
sage: quintic.gamma_covariant(as_form=True)
Binary monic given by ...
```

### `i_covariant(as_form=False)`

Return the covariant $i$ of a binary quintic.

**INPUT:**

- `as_form` - if `as_form` is False, the result will be returned as polynomial (default). If it is True the result is returned as an object of the class `AlgebraicForm`.

**OUTPUT:**

The $i$-covariant of the binary quintic as polynomial or as binary form.

**EXAMPLES:**

```
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: p = a0^5*x1^5 + a1*a2^4*x0 + a2^3*x0^3 + a3^2*x0^2 + a4*x0^4 + ...
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.i_covariant()  # as_form=False
3/50*a3^2*x0^2 - 4/25*a2*a4*x0^2 + 2/5*a1*a5*x0^2 + 1/25*a2*a3*x0^2 + 2*a0*a5*x0 + 3/50*a2^2*x1^2 - 4/25*a1*a3*x1^2 + 2/5*a0*a4*x1^2

sage: quintic.i_covariant(as_form=True)  # as_form=True
Binary quadratic given by ...
```

### `invariants(type='clebsch')`

Return a tuple of invariants of a binary quintic.

**INPUT:**

- `type` – The type of invariants to return. The default choice is to return the Clebsch invariants.

**OUTPUT:**

The invariants of the binary quintic.

**EXAMPLES:**

```
sage: R.<x0, x1> = QQ[]
sage: p = 2*x1^5 + 4*x1^4*x0 + 5*x1^3*x0^2 + 7*x1^2*x0^3 - 11*x1*x0^4 + x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.invariants()  # type=Clebsch
(-276032/625, 4983526016/390625, -24705649584625, -1489789728496847376/30517578125)
sage: quintic.invariants('unknown')  # type=unknown
(continues on next page)
```
Traceback (most recent call last):
...
ValueError: unknown type of invariants unknown for a binary quintic

\textbf{j\_covariant}(\texttt{as\_form=False})

Return the covariant \(j\) of a binary quintic.

\textbf{INPUT:}

\begin{itemize}
  \item \texttt{as\_form} – if \texttt{as\_form} is \texttt{False}, the result will be returned as polynomial (default). If it is \texttt{True} the result is returned as an object of the class \texttt{AlgebraicForm}.
\end{itemize}

\textbf{OUTPUT:}

The \(j\)-covariant of the binary quintic as polynomial or as binary form.

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage:} \texttt{R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]}
\texttt{sage:} \texttt{p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5}
\texttt{sage:} \texttt{quintic = invariant\_theory.binary\_quintic(p, x0, x1)}
\texttt{sage:} \texttt{quintic.j\_covariant()}  
-3/500*a3^3*x0^3 + 3/125*a2*a3*a4*x0^3 - 6/125*a1*a4^2*x0^3 - 3/50*a2^2*a5*x0^3 + 3/25*a1*a3*a5*x0^3 - 3/500*a2*a3^2*x0^2*x1 + 3/250*a2^2*a4*x0^2*x1 + 3/125*a1*a3*a4*x0^2*x1 - 6/25*a0*a4^2*x0^2*x1 - 3/25*a1*a2*a5*x0^2*x1 + 3/5*a0*a3*a5*x0^2*x1 - 3/500*a2^2*a3*x0*x1^2 + 3/250*a1*a3^2*x0*x1^2 + 3/125*a1*a2*a4*x0*x1^2 - 6/25*a0*a3*a4*x0*x1^2 - 3/25*a1^2*a5*x0*x1^2 + 3/5*a0*a2*a5*x0*x1^2 - 3/500*a2^3*x1^3 + 3/125*a1*a2*a3*x1^3 - 6/125*a1^2*a4*x1^3 + 3/25*a0*a2*a4*x1^3
\texttt{sage:} \texttt{quintic.j\_covariant(as\_form=True)}
Binary cubic given by ...
\end{verbatim}

\textbf{monomials()}

List the basis monomials of the form.

This function lists a basis of monomials of the space of binary quintics of which this form is an element.

\textbf{OUTPUT:}

A tuple of monomials. They are in the same order as \texttt{coeffs()}. 

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage:} \texttt{R.<x,y> = QQ[]}
\texttt{sage:} \texttt{quintic = invariant\_theory.binary\_quintic(x^5+y^5)}
\texttt{sage:} \texttt{quintic.monomials()}  
\texttt{((y^5, x*y^4, x^2*y^3, x^3*y^2, x^4*y, x^5)}
\end{verbatim}

\textbf{scaled\_coeffs()}

The coefficients of a binary quintic.

Given

\[
f(x) = a_0 x_1^5 + 5 a_1 x_0 x_1^4 + 10 a_2 x_0^2 x_1^3 + 10 a_3 x_0^3 x_1^2 + 5 a_4 x_0^4 x_1 + a_5 x_0^5
\]

this function returns \(a = (a_0, a_1, a_2, a_3, a_4, a_5)\)

\section{3.2. Classical Invariant Theory}
EXAMPLES:

```
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: p = a0*x1^5 + 5*a1*x1^4*x0 + 10*a2*x1^3*x0^2 + 10*a3*x1^2*x0^3 + 5*a4*x1*x0^4 + a5*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.scaled_coeffs()
(a0, a1, a2, a3, a4, a5)

sage: R.<a0, a1, a2, a3, a4, a5, x> = QQ[]
sage: p = a0 + 5*a1*x + 10*a2*x^2 + 10*a3*x^3 + 5*a4*x^4 + a5*x^5
sage: quintic = invariant_theory.binary_quintic(p, x)
sage: quintic.scaled_coeffs()
(a0, a1, a2, a3, a4, a5)
```

**tau_covariant** *(as_form=False)*

Return the covariant $\tau$ of a binary quintic.

**INPUT:**

* as_form – if as_form is False, the result will be returned as polynomial (default). If it is True the result is returned as an object of the class *AlgebraicForm*.

**OUTPUT:**

The $\tau$-covariant of the binary quintic as polynomial or as binary form.

**EXAMPLES:**

```
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.tau_covariant()
1/62500*a2^2*a3^4*x0^2 - 3/62500*a1*a3^5*x0^2
- 1/15625*a2^3*a3^2*a4*x0^3 + 1/6250*a1*a2*a3^3*a4*x0^2
+ 3/6250*a0*a3^4*a4*x0^2 - 3/31250*a2^4*a4^2*x0^2
...  
- 2/125*a0*a1*a2^2*a4*a5*x1^2 - 4/125*a0*a1^2*a3*a4*a5*x1^2
+ 2/25*a0*a2^2*a3*a4*a5*x1^2 - 8/625*a1*a4^2*a5*x1^2
+ 8/125*a0*a1^2*a2^2*a5*x1^2 - 2/25*a0^2*a2^2*a5^2*x1^2
sage: quintic.tau_covariant(as_form=True)
Binary quadratic given by ...
```

**theta_covariant** *(as_form=False)*

Return the covariant $\theta$ of a binary quintic.

**INPUT:**

* as_form – if as_form is False, the result will be returned as polynomial (default). If it is True the result is returned as an object of the class *AlgebraicForm*.

**OUTPUT:**

The $\theta$-covariant of the binary quintic as polynomial or as binary form.

**EXAMPLES:**

```
```
sage: R.<a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.theta_covariant()
-1/625000*a2^3*a3^5*x0^2 + 9/1250000*a1*a2*a3^6*x0^2
- 27/1250000*a0*a3^7*x0^2 + 3/250000*a2^4*a3^3*a4*x0^2
- 7/125000*a1*a2^2*a3^4*a4*x0^2 - 3/312500*a1^2*a3^5*a4*x0^2
...
+ 6/625*a0^2*a1*a2^2*a4*a5^2*x1^2 + 24/625*a0^2*a1^2*a3*a4*a5^2*x1^2
- 12/125*a0*a1^3*a2*a3^2*a5^2*x1^2 + 8/625*a0^2*a1^4*a5^3*x1^2
- 8/125*a0*a2*a1^2*a2*a5^3*x1^2 + 2/25*a0^3*a2^2*a5^3*x1^2
```

```python
class sage.rings.invariants.invariant_theory.FormsBase(n, homogeneous, ring, variables)
Bases: sage.structure.sage_object.SageObject

The common base class of `AlgebraicForm` and `SeveralAlgebraicForms`.

This is an abstract base class to provide common methods. It does not make much sense to instantiate it.

**is_homogeneous()**

Return whether the forms were defined by homogeneous polynomials.

**OUTPUT:**

Boolean. Whether the user originally defined the form via homogeneous variables.

**EXAMPLES:**

```
sage: R.<x,y,t> = QQ[]
sage: quartic = invariant_theory.binary_quartic(x^4+y^4+t*x^2*y^2, [x,y])
sage: quartic.is_homogeneous()
True
sage: quartic.form()
x^2*y^2*t + x^4 + y^4
```

```
sage: R.<x,y,t> = QQ[]
sage: quartic = invariant_theory.binary_quartic(x^4+1+t*x^2, [x])
sage: quartic.is_homogeneous()
False
sage: quartic.form()
x^4 + x^2*t + 1
```

**ring()**

Return the polynomial ring.

**OUTPUT:**

A polynomial ring. This is where the defining polynomial(s) live. Note that the polynomials may be homogeneous or inhomogeneous, depending on how the user constructed the object.

**EXAMPLES:**

3.2. Classical Invariant Theory
```python
sage: R.<x,y,t> = QQ[]
sage: quartic = invariant_theory.binary_quartic(x^4+y^4+t*x^2*y^2, [x,y])
sage: quartic.ring()
Multivariate Polynomial Ring in x, y, t over Rational Field
sage: R.<x,y,t> = QQ[]
sage: quartic = invariant_theory.binary_quartic(x^4+1+t*x^2, [x])
sage: quartic.ring()
Multivariate Polynomial Ring in x, y, t over Rational Field
```

variables()

Return the variables of the form.

OUTPUT:

A tuple of variables. If inhomogeneous notation is used for the defining polynomial then the last entry will be None.

EXAMPLES:

```python
sage: R.<x,y,t> = QQ[]
```

```python
sage: quartic = invariant_theory.binary_quartic(x^4+y^4+t*x^2*y^2, [x,y])
sage: quartic.variables()
(x, y)
```

```python
sage: R.<x,y,t> = QQ[]
```

```python
sage: quartic = invariant_theory.binary_quartic(x^4+1+t*x^2, [x])
sage: quartic.variables()
(x, None)
```

class sage.rings.invariants.invariant_theory.InvariantTheoryFactory

Bases: object

Factory object for invariants of multilinear forms.

Use the invariant_theory object to construct algebraic forms. These can then be queried for invariant and covariants.

EXAMPLES:

```python
sage: R.<x,y,z> = QQ[]
```

```python
sage: invariant_theory.ternary_cubic(x^3+y^3+z^3)
Ternary cubic with coefficients (1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0)
```

```python
sage: invariant_theory.ternary_cubic(x^3+y^3+z^3).J_covariant()
x^6*y^3 - x^3*y^6 - x^6*z^3 + y^6*z^3 + x^3*z^6 - y^3*z^6
```

binary_form_from_invariants(*degree, invariants, variables=None, as_form=True, **args, **kwargs)

Reconstruct a binary form from the values of its invariants.

INPUT:

- **degree** – The degree of the binary form.
- **invariants** – A list or tuple of values of the invariants of the binary form.
- **variables** – A list or tuple of two variables that are used for the resulting form (only if as_form is True). If no variables are provided, two abstract variables x and z will be used.
• as_form – boolean. If False, the function will return a tuple of coefficients of a binary form.

**OUTPUT:**

A binary form or a tuple of its coefficients, whose invariants are equal to the given invariants up to a scaling.

**EXAMPLES:**

In the case of binary quadratics and cubics, the form is reconstructed based on the value of the discriminant. See also `binary_quadratic_coefficients_from_invariants()` and `binary_cubic_coefficients_from_invariants()`. These methods will always return the same result if the discriminant is non-zero:

```sage
discriminant = 1
sage: invariant_theory.binary_form_from_invariants(2, [discriminant])
Binary quadratic with coefficients (1, -1/4, 0)
sage: invariant_theory.binary_form_from_invariants(3, [discriminant], as_form=False)
(0, 1, -1, 0)
```

For binary cubics, there is no class implemented yet, so `as_form=True` will yield an `NotImplementedError`:

```sage:
Traceback (most recent call last):
  ...:
NotImplementedError: no class for binary cubics implemented
```

For binary quintics, the three Clebsch invariants of the form should be provided to reconstruct the form. For more details about these invariants, see `clebsch_invariants()`:

```sage:
invariants = [1, 0, 0]
invariant_theory.binary_form_from_invariants(5, invariants)
Binary quintic with coefficients (1, 0, 0, 0, 0, 1)
```

An optional scaling argument may be provided in order to scale the resulting quintic. For more details, see `binary_quintic_coefficients_from_invariants()`:

```sage:
invariants = [3, 4, 7]
invariant_theory.binary_form_from_invariants(5, invariants, scaling='normalized')
Binary quintic with coefficients (24389/892616806656, 4205/11019966576, 0, 1015/209952, -145/1296, -3/16)
invariant_theory.binary_form_from_invariants(5, invariants, scaling='coprime')
Binary quintic with coefficients (-2048, 3840, 0, 876960, 2724840, -613089)
```

The invariants can also be computed using the invariants of a given binary quintic. The resulting form has the same invariants up to scaling, is GL(2, Q)-equivalent to the provided form and hence has the same canonical form (see `canonical_form()`):
sage: R.<x0, x1> = QQ[]
sage: p = 3*x1^5 + 6*x1^4*x0 + 3*x1^3*x0^2 + 4*x1^2*x0^3 - 5*x1*x0^4 + 4*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: invariants = quintic.clebsch_invariants(as_tuple=True)
sage: newquintic = invariant_theory.binary_form_from_invariants(5, invariants, variables=quintic.variables())
sage: newquintic
Binary quintic with coefficients (9592267437341790539005557/244140625000000000, 2149296282076255632004064707/61035156250000000000000000000, 11149651890347700974453304786783/76293945312500000000000, 122650775751894638395648891202734239/476837158203125000000000000000, 32399663094570565284728633453218447/1192092895507812500000000000000, 150450650364460836408395841632538558481466127/1490116119384765625000000000000)
sage: quintic.canonical_form() == newquintic.canonical_form()
True
For binary forms of other degrees, no reconstruction has been implemented yet. For forms of degree 6, see trac ticket #26462:

```
sage: invariant_theory.binary_form_from_invariants(6, invariants)
Traceback (most recent call last):
  ...
NotImplementedError: no reconstruction for binary forms of degree 6 implemented
```

### binary_quadratic(quadratic, *args)

Invariant theory of a quadratic in two variables.

**INPUT:**

- `quadratic` – a quadratic form.
- `x`, `y` – the homogeneous variables. If `y` is `None`, the quadratic is assumed to be inhomogeneous.

**REFERENCES:**


**EXAMPLES:**

```
sage: R.<x,y> = QQ[]
sage: invariant_theory.binary_quadratic(x^2+y^2)
Binary quadratic with coefficients (1, 1, 0)
sage: T.<t> = QQ[]
sage: invariant_theory.binary_quadratic(t^2 + 2*t + 1, [t])
Binary quadratic with coefficients (1, 1, 2)
```

### binary_quartic(quartic, *args, **kwds)

Invariant theory of a quartic in two variables.

The algebra of invariants of a quartic form is generated by invariants $i, j$ of degrees 2, 3. This ring is naturally isomorphic to the ring of modular forms of level 1, with the two generators corresponding to the Eisenstein series $E_4$ (see `EisensteinD()`) and $E_6$ (see `EisensteinE()`). The algebra of covariants is generated by these two invariants together with the form $f$ of degree 1 and order 4, the Hessian $g$ (see `g_covariant()`) of degree 2 and order 4, and a covariant $h$ (see `h_covariant()`) of degree 3 and order 6. They are related by a syzygy

$$jf^3 - gf^2i + 4g^3 + h^2 = 0$$
of degree 6 and order 12.

INPUT:

• **quartic** – a quartic.
• **x, y** – the homogeneous variables. If y is `None`, the quartic is assumed to be inhomogeneous.

REFERENCES:


EXAMPLES:

```python
sage: R.<x,y> = QQ[]
sage: quartic = invariant_theory.binary_quartic(x^4+y^4)
sage: quartic
Binary quartic with coefficients (1, 0, 0, 0, 1)
sage: type(quartic)
<class 'sage.rings.invariants.invariant_theory.BinaryQuartic'>
```

binary_quintic(quintic, *args, **kwds)

Create a binary quintic for computing invariants.

A binary quintic is a homogeneous polynomial of degree 5 in two variables. The algebra of invariants of a binary quintic is generated by the invariants \( A, B \) and \( C \) of respective degrees 4, 8 and 12 (see `A_invariant()`, `B_invariant()` and `C_invariant()`).

INPUT:

• **quintic** – a homogeneous polynomial of degree five in two variables or a (possibly inhomogeneous) polynomial of degree at most five in one variable.
• ***args** – the two homogeneous variables. If only one variable is given, the polynomial quintic is assumed to be univariate. If no variables are given, they are guessed.

REFERENCES:

• [Cle1872]

EXAMPLES:

If no variables are provided, they will be guessed:

```python
sage: R.<x,y> = QQ[]
sage: quintic = invariant_theory.binary_quintic(x^5+y^5)
sage: quintic
Binary quintic with coefficients (1, 0, 0, 0, 0, 1)
```

If only one variable is given, the quintic is the homogenisation of the provided polynomial:

```python
sage: quintic = invariant_theory.binary_quintic(x^5+y^5, x)
sage: quintic
Binary quintic with coefficients (y^5, 0, 0, 0, 0, 1)
sage: quintic.is_homogeneous()
False
```

If the polynomial has three or more variables, the variables should be specified:
\begin{verbatim}
sage: R.<x,y,z> = QQ[]
sage: quintic = invariant_theory.binary_quintic(x^5+z*y^5)
Traceback (most recent call last):
...
ValueError: need 2 or 1 variables, got (x, y, z)
sage: quintic = invariant_theory.binary_quintic(x^5+z*y^5, x, y)
sage: quintic
Binary quintic with coefficients (z, 0, 0, 0, 0, 1)
sage: type(quintic)
<class 'sage.rings.invariants.invariant_theory.BinaryQuintic'>
\end{verbatim}

**inhomogeneous_quadratic_form**(*polynomial*, *args*)

Invariants of an inhomogeneous quadratic form.

**INPUT:**

- *polynomial* – an inhomogeneous quadratic form.
- *args* – the variables as multiple arguments, or as a single list/tuple.

**EXAMPLES:**

\begin{verbatim}
sage: R.<x,y,z> = QQ[]
sage: quadratic = x^2+2*y^2+3*x*y+4*x+5*y+6
sage: inv3 = invariant_theory.inhomogeneous_quadratic_form(quadratic)
sage: type(inv3)
<class 'sage.rings.invariants.invariant_theory.TernaryQuadratic'>
sage: inv4 = invariant_theory.inhomogeneous_quadratic_form(x^2+y^2+z^2)
sage: type(inv4)
<class 'sage.rings.invariants.invariant_theory.QuadraticForm'>
\end{verbatim}

**quadratic_form**(*polynomial*, *args*)

Invariants of a homogeneous quadratic form.

**INPUT:**

- *polynomial* – a homogeneous or inhomogeneous quadratic form.
- *args* – the variables as multiple arguments, or as a single list/tuple. If the last argument is None, the cubic is assumed to be inhomogeneous.

**EXAMPLES:**

\begin{verbatim}
sage: R.<x,y,z> = QQ[]
sage: quadratic = x^2+y^2+z^2
sage: inv = invariant_theory.quadratic_form(quadratic)
sage: type(inv)
<class 'sage.rings.invariants.invariant_theory.TernaryQuadratic'>
\end{verbatim}

If some of the ring variables are to be treated as coefficients you need to specify the polynomial variables:

\begin{verbatim}
sage: R.<x,y,z, a,b> = QQ[]
sage: quadratic = a*x^2+b*y^2+z^2+2*y*z
sage: invariant_theory.quadratic_form(quadratic, x,y,z)
Ternary quadratic with coefficients (a, b, 1, 0, 0, 2)
sage: invariant_theory.quadratic_form(quadratic, [x,y,z])  # alternate syntax
Ternary quadratic with coefficients (a, b, 1, 0, 0, 2)
\end{verbatim}
Inhomogeneous quadratic forms (see also \texttt{inhomogeneous_quadratic_form()}) can be specified by passing \texttt{None} as the last variable:

\begin{verbatim}
sage: inhom = quadratic.subs(z=1)
sage: invariant_theory.quadratic_form(inhom, x,y,None)
Ternary quadratic with coefficients (a, b, 1, \theta, \theta, 2)
\end{verbatim}

\textbf{quaternary_biquadratic}(\texttt{quadratic1, quadratic2, *args, **kwds})

Invariants of two quadratics in four variables.

**INPUT:**

- \texttt{quadratic1, quadratic2} – two polynomials. Either homogeneous quadratic in 4 homogeneous variables, or inhomogeneous quadratic in 3 variables.
- \texttt{w, x, y, z} – the variables. If \texttt{z} is \texttt{None}, the quadratics are assumed to be inhomogeneous.

**EXAMPLES:**

\begin{verbatim}
sage: R.<w,x,y,z> = QQ[]
sage: q1 = w^2+x^2+y^2+z^2
sage: q2 = w*x + y*z
sage: inv = invariant_theory.quaternary_biquadratic(q1, q2)
sage: type(inv)
<class 'sage.rings.invariants.invariant_theory.TwoQuaternaryQuadratics'>
\end{verbatim}

Distance between two spheres [Sal1958], [Sal1965]

\begin{verbatim}
sage: R.<x,y,z, a,b,c, r1,r2> = QQ[]
sage: S1 = -r1^2 + x^2 + y^2 + z^2
sage: S2 = -r2^2 + (x-a)^2 + (y-b)^2 + (z-c)^2
sage: inv = invariant_theory.quaternary_biquadratic(S1, S2, [x, y, z])
sage: inv.Delta_invariant()  
-r1^2
sage: inv.Delta_prime_invariant()  
-r2^2
sage: inv.Theta_invariant()  
a^2 + b^2 + c^2 - 3*r1^2 - r2^2
sage: inv.Theta_prime_invariant()  
a^2 + b^2 + c^2 - r1^2 - 3*r2^2
sage: inv.Phi_invariant()  
2*a^2 + 2*b^2 + 2*c^2 - 3*r1^2 - 3*r2^2
sage: inv.J_covariant()  
0
\end{verbatim}

\textbf{quaternary_quadratic}(\texttt{quadratic, *args})

Invariant theory of a quadratic in four variables.

**INPUT:**

- \texttt{quadratic} – a quadratic form.
- \texttt{w, x, y, z} – the homogeneous variables. If \texttt{z} is \texttt{None}, the quadratic is assumed to be inhomogeneous.

**REFERENCES:**

- Wikipedia article Invariant_of_a_binary_form

**EXAMPLES:**

3.2. Classical Invariant Theory

459
ternary_biquadratic(quadratic1, quadratic2, *args, **kwds)

Invariants of two quadratics in three variables.

INPUT:

- quadratic1, quadratic2 – two polynomials. Either homogeneous quadratic in 3 homogeneous variables, or inhomogeneous quadratic in 2 variables.
- x, y, z – the variables. If z is None, the quadratics are assumed to be inhomogeneous.

EXAMPLES:

```python
sage: R.<x,y,z> = QQ[]
```

```python
sage: q1 = x^2+y^2+z^2
sage: q2 = x*y + y*z + x*z
```

```python
sage: inv = invariant_theory.ternary_biquadratic(q1, q2)
```

```python
sage: type(inv)
<class 'sage.rings.invariants.invariant_theory.TwoTernaryQuadratics'>
```

Distance between two circles:

```python
sage: R.<x,y, a,b, r1,r2> = QQ[]
```

```python
sage: S1 = -r1^2 + x^2 + y^2
sage: S2 = -r2^2 + (x-a)^2 + (y-b)^2
```

```python
sage: inv = invariant_theory.ternary_biquadratic(S1, S2, [x, y])
```

```python
sage: inv.Delta_invariant()
-r1^2
```

```python
sage: inv.Delta_prime_invariant()
-r2^2
```

```python
sage: inv.Theta_invariant()
a^2 + b^2 - 2*r1^2 - r2^2
```

```python
sage: inv.Theta_prime_invariant()
a^2 + b^2 - r1^2 - 2*r2^2
```

```python
sage: inv.F_covariant()
2*x^2*a^2 + y^2*a^2 - 2*x^2*a^3 + a^4 + 2*x*y*a*b - 2*y*a^2*b + x^2*b^2 + 2*y^2*b^2 - 2*y*a*b^2 - 2*x*y*a*b + 2*x*y*b^3 + 3*b^4 - 2*x^2*a*r1^2 - 2*y^2*a*r1^2 - 2*a^2*b*r1^2 + r1^4 - 2*x^2*r2^2 - 2*y^2*r2^2 + 2*x*a^2*r2^2 - 2*y*b^2*r2^2 + 2*y*b*r2^2 - 2*b^2*r2^2 + 2*r1^2*r2^2 + r2^4
```

```python
sage: inv.J_covariant()
-8*x^2*y*a*b^3 + 8*x*y^2*a^2 + 8*x^3*a^2*b + 8*x^4*a^3 - 16*x*y^2*a*b^2 - 8*x^2*a^2*b - 8*x^3*a^2 + 16*x^4*a^3 - 8*x^4*y^2*a*b - 8*x^4*a^2*b + 8*x^4*y^2*a*b - 8*x^4*a^2*b + 8*x^4*y^2*a*b - 8*x^4*a^2*b + 8*x^4*y^2*a*b - 8*x^4*a^2*b + 8*x^4*y^2*a*b - 8*x^4*a^2*b + 8*x^4*y^2*a*b - 8*x^4*a^2*b + 8*x^4*y^2*a*b - 8*x^4*a^2*b
```

```python
sage: R.<w,x,y,z> = QQ[]
```

```python
sage: invariant_theory.quaternary_quadratic(w^2+x^2+y^2+z^2)
Quaternary quadratic with coefficients (1, 1, 1, 0, 0, 0, 0, 0, 0, 0)
```

```python
sage: R.<x,y,z> = QQ[]
```

```python
sage: invariant_theory.quaternary_quadratic(1+x^2+y^2+z^2)
Quaternary quadratic with coefficients (1, 1, 1, 0, 0, 0, 0, 0, 0, 0)
```
ternary_cubic(cubic, *args, **kwds)

Invariants of a cubic in three variables.

The algebra of invariants of a ternary cubic under $SL_3(C)$ is a polynomial algebra generated by two invariants $S$ (see S_invariant()) and $T$ (see T_invariant()) of degrees 4 and 6, called Aronhold invariants.

The ring of covariants is given as follows. The identity covariant $U$ of a ternary cubic has degree 1 and order 3. The Hessian $H$ (see Hessian()) is a covariant of ternary cubics of degree 3 and order 3. There is a covariant $\Theta$ (see Theta_covariant()) of ternary cubics of degree 8 and order 6 that vanishes on points $x$ lying on the Salmon conic of the polar of $x$ with respect to the curve and its Hessian curve. The Brioschi covariant $J$ (see J_covariant()) is the Jacobian of $U$, $\Theta$, and $H$ of degree 12, order 9. The algebra of covariants of a ternary cubic is generated over the ring of invariants by $U$, $\Theta$, $H$, and $J$, with a relation

$$J^2 = 4\Theta^3 + TU^2\Theta^2 + \Theta(-4S^3U^4 + 2STU^3H - 72S^2U^2H^2$$

$$- 18TU^3H^3 + 108SH^4) - 16S^4U^5H - 11S^2TU^4H^2$$

$$- 4T^2U^3H^3 + 54STU^2H^4 - 432S^2UH^5 - 27TH^6$$

REFERENCES:

• Wikipedia article Ternary_cubic

INPUT:

• cubic – a homogeneous cubic in 3 homogeneous variables, or an inhomogeneous cubic in 2 variables.

• x, y, z – the variables. If z is None, the cubic is assumed to be inhomogeneous.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: cubic = invariant_theory.ternary_cubic(x^3+y^3+z^3)
sage: type(cubic)
<class 'sage.rings.invariants.invariant_theory.TernaryCubic'>
```

ternary_quadratic(quadratic, *args, **kwds)

Invariants of a quadratic in three variables.

INPUT:

• quadratic – a homogeneous quadratic in 3 homogeneous variables, or an inhomogeneous quadratic in 2 variables.

• x, y, z – the variables. If z is None, the quadratic is assumed to be inhomogeneous.

REFERENCES:

• Wikipedia article Invariant_of_a_binary_form

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: quadratic = x^2+y^2+z^2
sage: inv = invariant_theory.ternary_quadratic(quadratic)
```

3.2. Classical Invariant Theory 461
class sage.rings.invariants.invariant_theory.QuadraticForm(n, d, polynomial, *args)
Bases: sage.rings.invariants.invariant_theory.AlgebraicForm
Invariant theory of a multivariate quadratic form.
You should use the invariant_theory factory object to construct instances of this class. See quadratic_form() for details.

as_QuadraticForm()
Convert into a QuadraticForm.
OUTPUT:
Sage has a special quadratic forms subsystem. This method converts self into this QuadraticForm representation.
EXAMPLES:

coeffs()
The coefficients of a quadratic form.
Given
\[ f(x) = \sum_{0 \leq i < n} a_i x_i^2 + \sum_{0 \leq j < k < n} a_{jk} x_j x_k \]
this function returns \( a = (a_0, \ldots, a_n, a_00, a_{01}, \ldots, a_{n-1,n}) \)
EXAMPLES:
**discriminant()**

Return the discriminant of the quadratic form.

Up to an overall constant factor, this is just the determinant of the defining matrix, see `matrix()`.

For a quadratic form in \( n \) variables, the overall constant is \( 2^{n-1} \) if \( n \) is odd and \( (-1)^{n/2} 2^n \) if \( n \) is even.

**EXAMPLES:**

```python
sage: R.<a,b,c, x,y> = QQ[]
sage: p = a*x^2+b*x*y+c*y^2
sage: quadratic = invariant_theory.quadratic_form(p, x,y)
sage: quadratic.discriminant()
b^2 - 4*a*c
sage: R.<a,b,c,d,e,f,g, x,y,z> = QQ[]
sage: p = a*x^2 + b*y^2 + c*z^2 + d*x*y + e*x*z + f*y*z
sage: quadratic = invariant_theory.quadratic_form(p, x,y,z)
sage: quadratic.discriminant()
4*a*b*c - c*d^2 - b*e^2 + d*e*f - a*f^2
```

**dual()**

Return the dual quadratic form.

**OUTPUT:**

A new quadratic form (with the same number of variables) defined by the adjoint matrix.

**EXAMPLES:**

```python
sage: R.<a,b,c,x,y,z> = QQ[]
sage: cubic = x^2+y^2+z^2
sage: quadratic = invariant_theory.ternary_quadratic(a*x^2+b*y^2+c*z^2, [x,y,z])
sage: quadratic.form()
a*x^2 + b*y^2 + c*z^2
sage: quadratic.dual().form()
b*c*x^2 + a*c*y^2 + a*b*z^2
sage: R.<x,y,z, t> = QQ[]
sage: cubic = x^2+y^2+1 + t*x*y
sage: quadratic = invariant_theory.ternary_quadratic(x^2+y^2+1 + t*x*y, [x,y])
sage: quadratic.dual()
Ternary quadratic with coefficients (1, 1, -1/4*t^2 + 1, -t, 0, 0)
sage: R.<x,y, t> = QQ[]
sage: quadratic = invariant_theory.ternary_quadratic(x^2+y^2+1 + t*x*y, [x,y])
sage: quadratic.dual()
Ternary quadratic with coefficients (1, 1, -1/4*t^2 + 1, -t, 0, 0)
```

**classmethod from_invariants**

```
Construct a binary quadratic from its discriminant.

**INPUT:**

- discriminant – Value of the discriminant used to reconstruct the binary quadratic.

3.2. Classical Invariant Theory 463
```
OUTPUT:
A QuadraticForm with 2 variables.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: from sage.rings.invariants.invariant_theory import QuadraticForm
sage: QuadraticForm.from_invariants(1, x, y)
Binary quadratic with coefficients (1, -1/4, 0)
```

**invariants**(type='discriminant')

Return a tuple of invariants of a binary quadratic.

**INPUT:**

- type – The type of invariants to return. The default choice is to return the discriminant.

**OUTPUT:**

The invariants of the binary quadratic.

**EXAMPLES:**

```
sage: R.<x0, x1> = QQ[]
sage: p = 2*x1^2 + 5*x0*x1 + 3*x0^2
sage: quadratic = invariant_theory.binary_quadratic(p, x0, x1)
sage: quadratic.invariants()
(1,)
sage: quadratic.invariants('unknown')
Traceback (most recent call last):
...  
ValueError: unknown type of invariants unknown for a binary quadratic
```

**matrix()**

Return the quadratic form as a symmetric matrix

**OUTPUT:**

This method returns a symmetric matrix $A$ such that the quadratic $Q$ equals

$$Q(x, y, z, \ldots) = (x, y, \ldots)A(x, y, \ldots)^t$$

**EXAMPLES:**

```
sage: R.<x,y,z> = QQ[]
sage: quadratic = invariant_theory.ternary_quadratic(x^2+y^2+z^2+x*y)
sage: matrix(quadratic)
[ 1 1/2 0]
[1/2 1 0]
[ 0 0 1]
sage: quadratic._matrix_() == matrix(quadratic)
True
```

**monomials()**

List the basis monomials in the form.

**OUTPUT:**

A tuple of monomials. They are in the same order as **coeffs**().
EXAMPLES:

```python
sage: R.<x,y> = QQ[]
sage: quadratic = invariant_theory.quadratic_form(x^2+y^2)
sage: quadratic.monomials()
(x^2, y^2, x*y)
```

```python
sage: quadratic = invariant_theory.inhomogeneous_quadratic_form(x^2+y^2)
sage: quadratic.monomials()
(x^2, y^2, 1, x*y, x, y)
```

```
scaled_coeffs()

The scaled coefficients of a quadratic form.

Given

\[ f(x) = \sum_{0 \leq i < n} a_i x_i^2 + \sum_{0 \leq j < k < n} 2a_{jk} x_j x_k \]

this function returns \( a = (a_0, \ldots, a_n, a_{00}, a_{01}, \ldots, a_{n-1,n}) \)
```

EXAMPLES:

```python
sage: R.<a,b,c,d,e,f,g, x,y,z> = QQ[
```

```python
sage: p = a*x^2 + b*y^2 + c*z^2 + d*x*y + e*x*z + f*y*z
sage: inv = invariant_theory.quadratic_form(p, x,y,z)
```

```
sage: inv = invariant_theory.quadratic_form(p, x,y,z); inv
Ternary quadratic with coefficients (a, b, c, d, e, f)
sage: inv.coeffs()
(a, b, c, d, e, f)
sage: inv.scaled_coeffs()
(a, b, c, 1/2*d, 1/2*e, 1/2*f)
```

```
class sage.rings.invariants.invariant_theory.SeveralAlgebraicForms(forms)

Bases: sage.rings.invariants.invariant_theory.FormsBase

The base class of multiple algebraic forms (i.e. homogeneous polynomials).

You should only instantiate the derived classes of this base class.

See AlgebraicForm for the base class of a single algebraic form.

INPUT:

- `forms` – a list/tuple/iterable of at least one AlgebraicForm object, all with the same number of variables.

Interpreted as multiple homogeneous polynomials in a common polynomial ring.

EXAMPLES:

```python
sage: from sage.rings.invariants.invariant_theory import AlgebraicForm,
    SeveralAlgebraicForms
```

```python
sage: R.<x,y> = QQ[]
sage: p = AlgebraicForm(2, 2, x^2, (x,y))
sage: q = AlgebraicForm(2, 2, y^2, (x,y))
sage: pq = SeveralAlgebraicForms([p, q])
```

```
get_form(i)

Return the i-th form.

EXAMPLES:
```
```
sage: R.<x,y> = QQ[]
sage: q1 = invariant_theory.quadratic_form(x^2 + y^2)
sage: q2 = invariant_theory.quadratic_form(x*y)
sage: from sage.rings.invariants.invariant_theory import SeveralAlgebraicForms
sage: q12 = SeveralAlgebraicForms([q1, q2])
sage: q12.get_form(0) is q1
True
sage: q12.get_form(1) is q2
True
sage: q12[0] is q12.get_form(0) # syntactic sugar
True
sage: q12[1] is q12.get_form(1) # syntactic sugar
True
```

**homogenized**(var='h')

Return form as defined by a homogeneous polynomial.

**INPUT:**

- `var` – either a variable name, variable index or a variable (default: 'h').

**OUTPUT:**

The same algebraic form, but defined by a homogeneous polynomial.

**EXAMPLES:**

```
sage: R.<x,y,z> = QQ[]
sage: q = invariant_theory.quaternary_biquadratic(x^2+1, y^2+1, [x,y,z])
sage: q
Joint quaternary quadratic with coefficients (1, 0, 0, 1, 0, 0, 0, 0, 0, 0)
and quaternary quadratic with coefficients (0, 1, 0, 1, 0, 0, 0, 0, 0, 0)
sage: q.homogenized()
Joint quaternary quadratic with coefficients (1, 0, 0, 1, 0, 0, 0, 0, 0, 0)
and quaternary quadratic with coefficients (0, 1, 0, 1, 0, 0, 0, 0, 0, 0)
sage: type(q) is type(q.homogenized())
True
```

**n_forms()**

Return the number of forms.

**EXAMPLES:**

```
sage: R.<x,y> = QQ[]
sage: q1 = invariant_theory.quadratic_form(x^2 + y^2)
sage: q2 = invariant_theory.quadratic_form(x*y)
sage: from sage.rings.invariants.invariant_theory import SeveralAlgebraicForms
sage: q12 = SeveralAlgebraicForms([q1, q2])
sage: q12.n_forms()
2
sage: len(q12) == q12.n_forms()  # syntactic sugar
True
```

**class sage.rings.invariants.invariant_theory.TernaryCubic**(n, d, polynomial, *args)

Bases: `sage.rings.invariants.invariant_theory.AlgebraicForm`

Invariant theory of a ternary cubic.
You should use the `invariant_theory` factory object to construct instances of this class. See `ternary_cubic()` for details.

**Hessian()**

Return the Hessian covariant.

**OUTPUT:**

The Hessian matrix multiplied with the conventional normalization factor $1/216$.

**EXAMPLES:**

```python
sage: R.<x,y,z> = QQ[]
sage: cubic = invariant_theory.ternary_cubic(x^3+y^3+z^3)
sage: cubic.Hessian()
x*y*z
```

```python
sage: R.<x,y> = QQ[]
sage: cubic = invariant_theory.ternary_cubic(x^3+y^3+1)
sage: cubic.Hessian()
x*y
```

**J_covariant()**

Return the J-covariant of the ternary cubic.

**EXAMPLES:**

```python
sage: R.<x,y,z> = QQ[]
sage: cubic = invariant_theory.ternary_cubic(x^3+y^3+z^3)
sage: cubic.J_covariant()
x^6*y^3 - x^3*y^6 - x^6*z^3 + y^6*z^3 + x^3*z^6 - y^3*z^6
```

```python
sage: R.<x,y> = QQ[]
sage: cubic = invariant_theory.ternary_cubic(x^3+y^3+1)
sage: cubic.J_covariant()
x^6*y^3 - x^3*y^6 - x^6 + y^6 + x^3 - y^3
```

**S_invariant()**

Return the S-invariant.

**EXAMPLES:**

```python
sage: R.<x,y,z> = QQ[]
sage: cubic = invariant_theory.ternary_cubic(x^2*y+y^3+z^3+x*y*z)
sage: cubic.S_invariant()
-1/1296
```

**T_invariant()**

Return the T-invariant.

**EXAMPLES:**

```python
sage: R.<x,y,z> = QQ[]
sage: cubic = invariant_theory.ternary_cubic(x^3+y^3+z^3)
sage: cubic.T_invariant()
1
```
Theta_covariant()
Return the Θ covariant.

EXAMPLES:

```python
sage: R.<x,y,z> = QQ[]
sage: cubic = invariant_theory.ternary_cubic(x^3+y^3+z^3)
sage: cubic.Theta_covariant()
-x^3*y^3 - x^3*z^3 - y^3*z^3
```

coeffs()
Return the coefficients of a cubic.

Given

\[ p(x, y) = a_{30}x^3 + a_{21}x^2y + a_{12}xy^2 + a_{03}y^3 + a_{20}x^2 + \\
\quad a_{11}xy + a_{02}y^2 + a_{10}x + a_{01}y + a_{00} \]

this function returns \( a = (a_{30}, a_{03}, a_{00}, a_{21}, a_{20}, a_{12}, a_{02}, a_{10}, a_{01}, a_{11}) \)

EXAMPLES:

```python
sage: R.<x,y,z,a30,a21,a12,a03,a20,a10,a02,a11,a00> = QQ[]
sage: p = ( a30*x^3 + a21*x^2*y + a12*x*y^2 + a03*y^3 + a20*x^2 + \\
\quad a11*x*y + a02*y^2 + a10*x + a01*y + a00 )
sage: invariant_theory.ternary_cubic(p, x, y).coeffs()
(a30, a03, a00, a21, a20, a12, a02, a10, a01, a11)
```

monomials()
List the basis monomials of the form.

OUTPUT:
A tuple of monomials. They are in the same order as `coeffs()`.

EXAMPLES:
sage: R.<x,y,z> = QQ[]
sage: cubic = invariant_theory.ternary_cubic(x^3+y*z^2)
sage: cubic.monomials()
(x^3, y^3, z^3, x^2*y, x^2*z, x*y^2, y^2*z, x*z^2, y*z^2, x*y*z)

polar_conic()

Return the polar conic of the cubic.

OUTPUT:

Given the ternary cubic \( f(X, Y, Z) \), this method returns the symmetric matrix \( A(x, y, z) \) defined by

\[
xf_X + yf_Y + zf_Z = (X, Y, Z) \cdot A(x, y, z) \cdot (X, Y, Z)^t
\]

EXAMPLES:

sage: R.<x,y,z,X,Y,Z,a30,a21,a12,a03,a20,a10,a02,a10,a01,a00> = QQ[]
sage: p = ( a30*x^3 + a21*x^2*y + a12*x*y^2 + a03*y^3 + a20*x^2*z +
....:
   a11*x*y*z + a02*y^2*z + a10*x*z^2 + a01*y*z^2 + a00*z^3 )
sage: cubic = invariant_theory.ternary_cubic(p, x,y,z)
sage: cubic.polar_conic()
\[
\begin{bmatrix}
3*x*a30 + y*a21 + z*a20 & x*a21 + y*a12 + 1/2*z*a11 & x*a20 + 1/2*y*a11 + z*a10 \\
x*a21 + y*a12 + 1/2*z*a11 & x*a12 + 3*y*a03 + z*a02 + 1/2*x*a11 + y*a02 + z*a01 \\
x*a20 + 1/2*y*a11 + z*a10 & 1/2*x*a11 + y*a02 + z*a01 & x*a10 + y*a01 + 3*z*a00
\end{bmatrix}
\]
sage: polar_eqn = X*p.derivative(x) + Y*p.derivative(y) + Z*p.derivative(z)
sage: polar = invariant_theory.ternary_quadratic(polar_eqn, [x,y,z])
sage: polar.matrix().subs(X=x,Y=y,Z=z) == cubic.polar_conic()
True

scaled_coeffs()

Return the coefficients of a cubic.

Compared to \( \text{coeffs()} \), this method returns rescaled coefficients that are often used in invariant theory.

Given

\[
p(x, y) = a_{30}x^3 + a_{21}x^2y + a_{12}xy^2 + a_{03}y^3 + a_{20}x^2z +
   a_{11}xy + a_{02}y^2 + a_{10}x + a_{01}y + a_{00}
\]

this function returns \( a = (a_{30}, a_{03}, a_{00}, a_{21}/3, a_{20}/3, a_{12}/3, a_{02}/3, a_{10}/3, a_{01}/3, a_{11}/6) \)

EXAMPLES:

sage: R.<x,y,z,a30,a21,a12,a03,a20,a10,a02,a10,a01,a00> = QQ[]
sage: p = ( a30*x^3 + a21*x^2*y + a12*x*y^2 + a03*y^3 + a20*x^2*z +
....:
   a11*x*y*z + a02*y^2*z + a10*x*z^2 + a01*y*z^2 + a00*z^3 )
sage: invariant_theory.ternary_cubic(p, x,y,z).scaled_coeffs()
(a30, a03, a00, 1/3*a21, 1/3*a20, 1/3*a12, 1/3*a03, a20/3, 1/3*a02, 1/3*a10, 1/3*a01, 1/6*a11)

syzygy \((U, S, T, H, Theta, J)\)

Return the syzygy of the cubic evaluated on the invariants and covariants.

INPUT:

- \( U, S, T, H, Theta, J \) – polynomials from the same polynomial ring.

3.2. Classical Invariant Theory

469
OUTPUT:

0 if evaluated for the form, the S invariant, the T invariant, the Hessian, the \( \Theta \) covariant and the J-covariant of a ternary cubic.

EXAMPLES:

```python
sage: R.<x,y,z> = QQ[]
sage: monomials = (x^3, y^3, z^3, x^2*y, x^2*z, x*y^2,
....: y^2*z, x*z^2, y*z^2, x*y*z)
sage: random_poly = sum([ randint(0,10000) * m for m in monomials ])
sage: cubic = invariant_theory.ternary_cubic(random_poly)
sage: U = cubic.form()
sage: S = cubic.S_invariant()
sage: T = cubic.T_invariant()
sage: H = cubic.Hessian()
sage: Theta = cubic.Theta_covariant()
sage: J = cubic.J_covariant()
sage: cubic.syzygy(U, S, T, H, Theta, J)
```

```python
class sage.rings.invariants.invariant_theory.TernaryQuadratic(n, d, polynomial, *args)
Bases: sage.rings.invariants.invariant_theory.QuadraticForm

Invariant theory of a ternary quadratic.

You should use the invariant_theory factory object to construct instances of this class. See ternary_quadratic() for details.

coeffs()

Return the coefficients of a quadratic.

Given

\[
p(x, y) = a_{20}x^2 + a_{11}xy + a_{02}y^2 + a_{10}x + a_{01}y + a_{00}
\]

this function returns \( a = (a_{20}, a_{02}, a_{00}, a_{11}, a_{10}, a_{01}) \)

EXAMPLES:

```python
sage: R.<x,y,z,a20,a11,a02,a10,a01,a00> = QQ[]
sage: p = ( a20*x^2 + a11*x*y + a02*y^2 +
....: a10*x*z + a01*y*z + a00*z^2 )
sage: invariant_theory.ternary_quadratic(p, x,y,z).coeffs()
(a20, a02, a00, a11, a10, a01)
sage: invariant_theory.ternary_quadratic(p.subs(z=1), x, y).coeffs()
(a20, a02, a00, a11, a10, a01)
```

covariant_conic(other)

Return the ternary quadratic covariant to self and other.

INPUT:

- other – Another ternary quadratic.

OUTPUT:

The so-called covariant conic, a ternary quadratic. It is symmetric under exchange of self and other.

EXAMPLES:
sage: ring.<x,y,z> = QQ[]
sage: Q = invariant_theory.ternary_quadratic(x^2+y^2+z^2)
sage: R = invariant_theory.ternary_quadratic(x*y+x*z+y*z)
sage: Q.covariant_conic(R)
-x*y - x*z - y*z
sage: R.covariant_conic(Q)
-x*y - x*z - y*z

monomials()  
List the basis monomials of the form.

OUTPUT:  
A tuple of monomials. They are in the same order as coeffs().

EXAMPLES:

sage: R.<x,y,z> = QQ[]
sage: quadratic = invariant_theory.ternary_quadratic(x^2+y*z)
sage: quadratic.monomials()
(x^2, y^2, z^2, x*y, x*z, y*z)

scaled_coeffs()  
Return the scaled coefficients of a quadratic.

Given  
\[ p(x, y) = a_{20}x^2 + a_{11}xy + a_{02}y^2 + a_{10}x + a_{01}y + a_{00} \]

this function returns  
\[ a = (a_{20}, a_{02}, a_{00}, 1/2a_{11}, 1/2a_{10}, 1/2a_{01}) \]

EXAMPLES:

sage: R.<x,y,z,a20,a11,a02,a10,a01,a00> = QQ[]
sage: p = ( a20*x^2 + a11*x*y + a02*y^2 + 
.....: a10*x*z + a01*y*z + a00*z^2 )
sage: invariant_theory.ternary_quadratic(p, x,y,z).scaled_coeffs()
(a20, a02, a00, 1/2*a11, 1/2*a10, 1/2*a01)
sage: invariant_theory.ternary_quadratic(p.subs(z=1), x, y).scaled_coeffs()
(a20, a02, a00, 1/2*a11, 1/2*a10, 1/2*a01)

class sage.rings.invariants.invariant_theory.TwoAlgebraicForms(forms)  
Bases: sage.rings.invariants.invariant_theory.SeveralAlgebraicForms

first()  
Return the first of the two forms.

OUTPUT:  
The first algebraic form used in the definition.

EXAMPLES:

sage: R.<x,y> = QQ[]
sage: q0 = invariant_theory.quadratic_form(x^2 + y^2)
sage: q1 = invariant_theory.quadratic_form(x*y)
sage: from sage.rings.invariants.invariant_theory import TwoAlgebraicForms
sage: q = TwoAlgebraicForms([q0, q1])

(continues on next page)
sage: q.first() is q0
True
sage: q.get_form(0) is q0
True
sage: q.first().polynomial()
x^2 + y^2

second()
Return the second of the two forms.

OUTPUT:
The second form used in the definition.

EXAMPLES:

sage: R.<x,y> = QQ[]
sage: q0 = invariant_theory.quadratic_form(x^2 + y^2)
sage: q1 = invariant_theory.quadratic_form(x*y)
sage: from sage.rings.invariants.invariant_theory import TwoAlgebraicForms
sage: q = TwoAlgebraicForms([q0, q1])
sage: q.second() is q1
True
sage: q.get_form(1) is q1
True
sage: q.second().polynomial()
x*y

class sage.rings.invariants.invariant_theory.TwoQuaternaryQuadratics(forms)
Bases: sage.rings.invariants.invariant_theory.TwoAlgebraicForms

Invariant theory of two quaternary quadratics.
You should use the invariant_theory factory object to construct instances of this class. See quaternary_biquadratics() for details.

REFERENCES:
• section on “Invariants and Covariants of Systems of Quadrics” in [Sal1958], [Sal1965]

Delta_invariant()
Return the $\Delta$ invariant.

EXAMPLES:

sage: R.<x,y,z,t,a0,a1,a2,a3,b0,b1,b2,b3,b4,b5,A0,A1,A2,A3,B0,B1,B2,B3,B4,B5> = QQ[]
sage: p1 = a0*x^2 + a1*y^2 + a2*z^2 + a3
sage: p1 += b0*x*y + b1*x*z + b2*x + b3*y*z + b4*y + b5*z
sage: p2 = A0*x^2 + A1*y^2 + A2*z^2 + A3
sage: p2 += B0*x*y + B1*x*z + B2*x + B3*y*z + B4*y + B5*z
sage: q = invariant_theory.quaternary_biquadratic(p1, p2, [x, y, z])
sage: coeffs = det(t * q[0].matrix() + q[1].matrix()).polynomial(t).
    coefficients(sparse=False)
True
Delta_prime_invariant()  
Return the $\Delta'$ invariant.

EXAMPLES:

```python
sage: R.<x,y,z,t,a0,a1,a2,a3,b0,b1,b2,b3,b4,b5,A0,A1,A2,A3,B0,B1,B2,B3,B4,B5> = QQ[]
sage: p1 = a0*x^2 + a1*y^2 + a2*z^2 + a3
sage: p1 += b0*x*y + b1*x*z + b2*x + b3*y*z + b4*y + b5*z
sage: p2 = A0*x^2 + A1*y^2 + A2*z^2 + A3
sage: p2 += B0*x*y + B1*x*z + B2*x + B3*y*z + B4*y + B5*z
sage: q = invariant_theory.quaternary_biquadratic(p1, p2, [x, y, z])
sage: coeffs = det(t * q[0].matrix() + q[1].matrix()).polynomial(t).
    --coefficients(sparse=False)
sage: q.Delta_prime_invariant() == coeffs[0]
True
```

J_covariant()  
The $J$-covariant.

This is the Jacobian determinant of the two biquadratics, the $T$-covariant, and the $T'$-covariant with respect to the four homogeneous variables.

EXAMPLES:

```python
sage: R.<w,x,y,z,a0,a1,a2,a3,A0,A1,A2,A3> = QQ[]
sage: p1 = a0*x^2 + a1*y^2 + a2*z^2 + a3*w^2
sage: p2 = A0*x^2 + A1*y^2 + A2*z^2 + A3*w^2
sage: q = invariant_theory.quaternary_biquadratic(p1, p2, [w, x, y, z])
sage: q.J_covariant().factor()
z * y * x * w * (a3*A2 - a2*A3) * (a3*A1 - a1*A3) * (-a2*A1 + a1*A2)
    * (a3*A0 - a0*A3) * (-a2*A0 + a0*A2) * (-a1*A0 + a0*A1)
```

Phi_invariant()  
Return the $\Phi'$ invariant.

EXAMPLES:

```python
sage: R.<x,y,z,t,a0,a1,a2,a3,b0,b1,b2,b3,b4,b5,A0,A1,A2,A3,B0,B1,B2,B3,B4,B5> = QQ[]
sage: p1 = a0*x^2 + a1*y^2 + a2*z^2 + a3
sage: p1 += b0*x*y + b1*x*z + b2*x + b3*y*z + b4*y + b5*z
sage: p2 = A0*x^2 + A1*y^2 + A2*z^2 + A3
sage: p2 += B0*x*y + B1*x*z + B2*x + B3*y*z + B4*y + B5*z
sage: q = invariant_theory.quaternary_biquadratic(p1, p2, [x, y, z])
sage: coeffs = det(t * q[0].matrix() + q[1].matrix()).polynomial(t).
    --coefficients(sparse=False)
sage: q.Phi_invariant() == coeffs[2]
True
```

T_covariant()  
The $T$-covariant.

EXAMPLES:
sage: R.<x,y,z,t,a0,a1,a2,a3,b0,b1,b2,b3,b4,b5,A0,A1,A2,A3,B0,B1,B2,B3,B4,B5> = QQ[

sage: p1 = a0*x^2 + a1*y^2 + a2*z^2 + a3
sage: p1 += b0*x*y + b1*x*z + b2*x + b3*y*z + b4*y + b5*z
sage: p2 = A0*x^2 + A1*y^2 + A2*z^2 + A3
sage: p2 += B0*x*y + B1*x*z + B2*x + B3*y*z + B4*y + B5*z
sage: q = invariant_theory.quaternary_biquadratic(p1, p2, [x, y, z])

sage: T = invariant_theory.quaternary_quadratic(q.T_covariant(), [x,y,z]).matrix()

sage: M = q[0].matrix().adjugate() + t*q[1].matrix().adjugate()

sage: M = M.adjugate().apply_map(# long time (4s on my thinkpad, W530)

....: lambda m: m.coefficient(t))

sage: M == q.Delta_invariant()*T
# long time True

T_prime_covariant()
The $T^\prime$-covariant.

EXAMPLES:

sage: R.<x,y,z,t,a0,a1,a2,a3,b0,b1,b2,b3,b4,b5,A0,A1,A2,A3,B0,B1,B2,B3,B4,B5> = QQ[

sage: p1 = a0*x^2 + a1*y^2 + a2*z^2 + a3
sage: p1 += b0*x*y + b1*x*z + b2*x + b3*y*z + b4*y + b5*z
sage: p2 = A0*x^2 + A1*y^2 + A2*z^2 + A3
sage: p2 += B0*x*y + B1*x*z + B2*x + B3*y*z + B4*y + B5*z
sage: q = invariant_theory.quaternary_biquadratic(p1, p2, [x, y, z])

sage: Tprime = invariant_theory.quaternary_quadratic(q.T_prime_covariant(), [x,y,z]).matrix()

sage: M = q[0].matrix().adjugate() + t*q[1].matrix().adjugate()

sage: M = M.adjugate().apply_map(# long time (4s on my thinkpad, W530)

....: lambda m: m.coefficient(t^2))

sage: M == q.Delta_prime_invariant() * Tprime
# long time True

Theta_invariant()
Return the $\Theta$ invariant.

EXAMPLES:

sage: R.<x,y,z,t,a0,a1,a2,a3,b0,b1,b2,b3,b4,b5,A0,A1,A2,A3,B0,B1,B2,B3,B4,B5> = QQ[

sage: p1 = a0*x^2 + a1*y^2 + a2*z^2 + a3
sage: p1 += b0*x*y + b1*x*z + b2*x + b3*y*z + b4*y + b5*z
sage: p2 = A0*x^2 + A1*y^2 + A2*z^2 + A3
sage: p2 += B0*x*y + B1*x*z + B2*x + B3*y*z + B4*y + B5*z
sage: q = invariant_theory.quaternary_biquadratic(p1, p2, [x, y, z])

sage: coeffs = det(t * q[0].matrix() + q[1].matrix()).polynomial(t).

coefficients(sparse=False)

sage: q.Theta_invariant() == coeffs[3]
True
** Theta_prime_invariant()**

Return the $\Theta'$ invariant.

EXAMPLES:

```
sage: R.<x,y,z,t,a0,a1,a2,a3,b0,b1,b2,b3,b4,b5,A0,A1,A2,A3,B0,B1,B2,B3,B4,B5> = QQ[]
sage: p1 = a0*x^2 + a1*y^2 + a2*z^2 + a3
sage: p1 += b0*x*y + b1*x*z + b2*x + b3*y*z + b4*y + b5*z
sage: p2 = A0*x^2 + A1*y^2 + A2*z^2 + A3
sage: p2 += B0*x*y + B1*x*z + B2*x + B3*y*z + B4*y + B5*z
sage: q = invariant_theory.quaternary_biquadratic(p1, p2, [x, y, z])
sage: coeffs = det(t * q[0].matrix() + q[1].matrix()).polynomial(t).
sage: coefficents(sparse=False)
sage: q.Theta_prime_invariant() == coeffs[1]
True
```

** syzygy(Delta, Theta, Phi, Theta_prime, Delta_prime, U, V, T, T_prime, J)**

Return the syzygy evaluated on the invariants and covariants.

INPUT:

- Delta, Theta, Phi, Theta_prime, Delta_prime, U, V, T, T_prime, J – polynomials from the same polynomial ring.

OUTPUT:

Zero if the U is the first polynomial, V the second polynomial, and the remaining input are the invariants and covariants of a quaternary biquadratic.

EXAMPLES:

```
sage: R.<w,x,y,z> = QQ[]
sage: monomials = [x^2, x*y, y^2, x*z, y*z, z^2, x*w, y*w, z*w, w^2]
sage: def q_rnd(): return sum(randint(-1000,1000)*m for m in monomials)
sage: biquadratic = invariant_theory.quaternary_biquadratic(q_rnd(), q_rnd())
sage: Delta = biquadratic.Delta_invariant()
sage: Theta = biquadratic.Theta_invariant()
sage: Phi = biquadratic.Phi_invariant()
sage: Theta_prime = biquadratic.Theta_prime_invariant()
sage: Delta_prime = biquadratic.Delta_prime_invariant()
sage: U = biquadratic.first().polynomial()
sage: V = biquadratic.second().polynomial()
sage: T = biquadratic.T_covariant()
sage: T_prime = biquadratic.T_prime_covariant()
sage: J = biquadratic.J_covariant()
sage: biquadratic.syzygy(Delta, Theta, Phi, Theta_prime, Delta_prime, U, V, T, T_prime, J)
0
```

If the arguments are not the invariants and covariants then the output is some (generically non-zero) polynomial:

```
sage: biquadratic.syzygy(1, 1, 1, 1, 1, 1, 1, 1, 1, x)
-x^2 + 1
```

**class** `sage.rings.invariants.invariant_theory.TwoTernaryQuadratics` *(forms)*

Bases: `sage.rings.invariants.invariant_theory.TwoAlgebraicForms`
Invariant theory of two ternary quadratics.

You should use the `invariant_theory` factory object to construct instances of this class. See `ternary_biquadratics()` for details.

REFERENCES:

- Section on “Invariants and Covariants of Systems of Conics”, Art. 388 (a) in [Sal1954]

**Delta_invariant()**

Return the $\Delta$ invariant.

EXAMPLES:

```
sage: R.<a00, a01, a11, a02, a12, a22, b00, b01, b11, b02, b12, b22, y0, y1, y2, t> = QQ[]
sage: p1 = a00*y0^2 + 2*a01*y0*y1 + a11*y1^2 + 2*a02*y0*y2 + 2*a12*y1*y2 + a22*y2^2
sage: p2 = b00*y0^2 + 2*b01*y0*y1 + b11*y1^2 + 2*b02*y0*y2 + 2*b12*y1*y2 + b22*y2^2
sage: q = invariant_theory.ternary_biquadratic(p1, p2, [y0, y1, y2])
sage: coeffs = det(t * q[0].matrix() + q[1].matrix()).polynomial(t).coefficients(sparse=False)
sage: q.Delta_invariant() == coeffs[3]
True
```

**Delta_prime_invariant()**

Return the $\Delta'$ invariant.

EXAMPLES:

```
sage: R.<a00, a01, a11, a02, a12, a22, b00, b01, b11, b02, b12, b22, y0, y1, y2, t> = QQ[]
sage: p1 = a00*y0^2 + 2*a01*y0*y1 + a11*y1^2 + 2*a02*y0*y2 + 2*a12*y1*y2 + a22*y2^2
sage: p2 = b00*y0^2 + 2*b01*y0*y1 + b11*y1^2 + 2*b02*y0*y2 + 2*b12*y1*y2 + b22*y2^2
sage: q = invariant_theory.ternary_biquadratic(p1, p2, [y0, y1, y2])
sage: coeffs = det(t * q[0].matrix() + q[1].matrix()).polynomial(t).coefficients(sparse=False)
sage: q.Delta_prime_invariant() == coeffs[0]
True
```

**F_covariant()**

Return the $F$ covariant.

EXAMPLES:

```
sage: R.<a00, a01, a11, a02, a12, a22, b00, b01, b11, b02, b12, b22, x, y> = QQ[]
sage: p1 = 73*x^2 + 96*x*y - 11*y^2 + 4*x + 63*y + 57
sage: p2 = 61*x^2 - 100*x*y - 72*y^2 - 81*x + 39*y - 7
sage: q = invariant_theory.ternary_biquadratic(p1, p2, [x, y])
sage: q.F_covariant()
-32566577*x^2 + 29060637/2*x*y + 20153633/4*y^2 - 30250497/2*x - 241241273/4*y - 323820473/16
```
J_covariant()
Return the \( J \) covariant.

EXAMPLES:

sage: R.<a00, a01, a11, a02, a12, a22, b00, b01, b11, b02, b12, b22, x, y> = QQ[]
sage: p1 = 73*x^2 + 96*x*y - 11*y^2 + 4*x + 63*y + 57
sage: p2 = 61*x^2 - 100*x*y - 72*y^2 - 81*x + 39*y - 7
sage: q = invariant_theory.ternary_biquadratic(p1, p2, [x, y])
sage: q.J_covariant()
1057324024445*x^3 + 1209531088209*x^2*y + 942116599708*x*y^2 +
984553030871*y^3 + 543715345505/2*x^2 - 3065093506021/2*x*y +
755269485570/2*y^2 - 1118430692650*x - 509948695327/4*y + 3369951531745/8

Theta_invariant()
Return the \( \Theta \) invariant.

EXAMPLES:

sage: R.<a00, a01, a11, a02, a12, a22, b00, b01, b11, b02, b12, b22, y0, y1, y2, t> = QQ[]
sage: p1 = a00*y0^2 + 2*a01*y0*y1 + a11*y1^2 + 2*a02*y0*y2 + 2*a12*y1*y2 + a22*y2^2
sage: p2 = b00*y0^2 + 2*b01*y0*y1 + b11*y1^2 + 2*b02*y0*y2 + 2*b12*y1*y2 + b22*y2^2
sage: q = invariant_theory.ternary_biquadratic(p1, p2, [y0, y1, y2])
sage: coeffs = det(t * q[0].matrix() + q[1].matrix()).polynomial(t).
|coefficients(sparse=False)
sage: q.Theta_invariant() == coeffs[2]
True

Theta_prime_invariant()
Return the \( \Theta' \) invariant.

EXAMPLES:

sage: R.<a00, a01, a11, a02, a12, a22, b00, b01, b11, b02, b12, b22, y0, y1, y2, t> = QQ[]
sage: p1 = a00*y0^2 + 2*a01*y0*y1 + a11*y1^2 + 2*a02*y0*y2 + 2*a12*y1*y2 + a22*y2^2
sage: p2 = b00*y0^2 + 2*b01*y0*y1 + b11*y1^2 + 2*b02*y0*y2 + 2*b12*y1*y2 + b22*y2^2
sage: q = invariant_theory.ternary_biquadratic(p1, p2, [y0, y1, y2])
sage: coeffs = det(t * q[0].matrix() + q[1].matrix()).polynomial(t).
|coefficients(sparse=False)
sage: q.Theta_prime_invariant() == coeffs[1]
True

syzygy(Delta, Theta, Theta_prime, Delta_prime, S, S_prime, F, J)
Return the syzygy evaluated on the invariants and covariants.

INPUT:

- Delta, Theta, Theta_prime, Delta_prime, S, S_prime, F, J – polynomials from the same polynomial ring.

OUTPUT:

Zero if $S$ is the first polynomial, $S_{\text{prime}}$ the second polynomial, and the remaining input are the invariants and covariants of a ternary biquadratic.

EXAMPLES:

```python
sage: R.<x,y,z> = QQ[]
sage: monomials = [x^2, x*y, y^2, x*z, y*z, z^2]
sage: def q_rnd():
    return sum(randint(-1000, 1000)*m for m in monomials)
sage: biquadratic = invariant_theory.ternary_biquadratic(q_rnd(), q_rnd(), [x,y,z])
sage: Delta = biquadratic.Delta_invariant()
sage: Theta = biquadratic.Theta_invariant()
sage: Delta_prime = biquadratic.Delta_prime_invariant()
sage: S = biquadratic.first().polynomial()
sage: S_prime = biquadratic.second().polynomial()
sage: F = biquadratic.F_covariant()
sage: J = biquadratic.J_covariant()
sage: biquadratic.syzygy(Delta, Theta, Theta_prime, Delta_prime, S, S_prime, F, J)
0
```

If the arguments are not the invariants and covariants then the output is some (generically non-zero) polynomial:

```python
sage: biquadratic.syzygy(1, 1, 1, 1, 1, 1, 1, x)
1/64*x^2 + 1
```

`sage.rings.invariants.invariant_theory.transvectant(f, g, h=1, scale='default')`

Return the $h$-th transvectant of $f$ and $g$.

**INPUT:**

- $f, g$ – two homogeneous binary forms in the same polynomial ring.
- $h$ – the order of the transvectant. If it is not specified, the first transvectant is returned.
- `scale` – the scaling factor applied to the result. Possible values are 'default' and 'none'. The 'default' scaling factor is the one that appears in the output statement below, if the scaling factor is 'none' the quotient of factorials is left out.

**OUTPUT:**

The $h$-th transvectant of the listed forms $f$ and $g$:

$$(f, g)_h = \frac{(d_f - h)! \cdot (d_g - h)!}{d_f! \cdot d_g!} \left( \frac{\partial}{\partial x} \frac{\partial}{\partial z'} - \frac{\partial}{\partial x'} \frac{\partial}{\partial z} \right)^h (f(x, z) \cdot g(x', z'))_{(x', z')=(x, z)}$$

**EXAMPLES:**

```python
sage: from sage.rings.invariants.invariant_theory import AlgebraicForm, transvectant
sage: R.<x,y> = QQ[]
sage: f = AlgebraicForm(2, 5, x^5 + 5*x^4*y + 5*x^2*y^4 + y^5)
sage: transvectant(f, f, 4)
Binary quadratic given by 2*x^2 - 4*x*y + 2*y^2
sage: transvectant(f, f, 8)
Binary form of degree -6 given by 0
```

478 Chapter 3. Multivariate Polynomials
The default scaling will yield an error for fields of positive characteristic below $d_f$ or $d_g$ as the denominator of the scaling factor will not be invertible in that case. The scale argument `none` can be used to compute the transvectant in this case:

```python
sage: R.<a0,a1,a2,a3,a4,a5,x0,x1> = GF(5)[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5
sage: f = AlgebraicForm(2, 5, p, x0, x1)
sage: transvectant(f, f, 4)
Traceback (most recent call last):
...
ZeroDivisionError
```

The additional factors that appear when scale='none' is used can be seen if we consider the same transvectant over the rationals and compare it to the scaled version:

```python
sage: R.<a0,a1,a2,a3,a4,a5,x0,x1> = QQ[]
sage: p = a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5
sage: f = AlgebraicForm(2, 5, p, x0, x1)
sage: transvectant(f, f, 4, scale='none')
Binary quadratic given by -a3^2*x0^2 + a2*a4*x0^2 + a2*a3*x0*x1
- a1*a4*x0*x1 - a2^2*x1^2 + a1*a3*x1^2
```

If the forms are given as inhomogeneous polynomials, the homogenisation might fail if the polynomial ring has multiple variables. You can circumvent this by making sure the base ring of the polynomial has only one variable:

```python
sage: R.<x,y> = QQ[]
sage: quintic = invariant_theory.binary_quintic(x^5+x^3+2*x^2+y^5, x)
sage: transvectant(quintic, quintic, 2)
Traceback (most recent call last):
...
ValueError: polynomial is not homogeneous
```

```python
sage: R.<y> = QQ[]
sage: S.<x> = R[]
sage: quintic = invariant_theory.binary_quintic(x^5+x^3+2*x^2+y^5, x)
sage: transvectant(quintic, quintic, 2)
Binary sextic given by 1/5*x^6 + 6/5*x^5*h + (-3/25)*x^4*h^2
+ (2*y^5 - 8/25)*x^3*h^3 + (-12/25)*x^2*h^4 + 3/5*y^5*x*h^5
+ 2/5*y^5*h^6
```
3.2.2 Reconstruction of Algebraic Forms

This module reconstructs algebraic forms from the values of their invariants. Given a set of (classical) invariants, it returns a form that attains this values as invariants (up to scaling).

AUTHORS:

• Jesper Noordsij (2018-06): initial version

```
sage.rings.invariants.reconstruction.binary_cubic_coefficients_from_invariants(discriminant, invariant_choice='default')
```

Reconstruct a binary cubic from the value of its discriminant.

INPUT:

• discriminant – The value of the discriminant of the binary cubic.
• invariant_choice – The type of invariants provided. The accepted options are 'discriminant' and 'default', which are the same. No other options are implemented.

OUTPUT:

A set of coefficients of a binary cubic, whose discriminant is equal to the given discriminant up to a scaling.

EXAMPLES:

```
sage: from sage.rings.invariants.reconstruction import binary_cubic_coefficients_from_invariants
sage: coeffs = binary_cubic_coefficients_from_invariants(1)
sage: coeffs
(0, 1, -1, 0)
sage: R.<x> = QQ[]
sage: R(coeffs).discriminant()
1
```

The two non-equivalent cubics $x^3$ and $x^2 * z$ with discriminant 0 can’t be distinguished based on their discriminant, hence an error is raised:

```
sage: binary_cubic_coefficients_from_invariants(0)
Traceback (most recent call last):
  ... ValueError: no unique reconstruction possible for binary cubics with a double root
```

```
sage.rings.invariants.reconstruction.binary_quadratic_coefficients_from_invariants(discriminant, invariant_choice='default')
```

Reconstruct a binary quadratic from the value of its discriminant.

INPUT:

• discriminant – The value of the discriminant of the binary quadratic.
• invariant_choice – The type of invariants provided. The accepted options are 'discriminant' and 'default', which are the same. No other options are implemented.

OUTPUT:

A set of coefficients of a binary quadratic, whose discriminant is equal to the given discriminant up to a scaling.

EXAMPLES:
Reconstruct a binary quintic from the values of its (Clebsch) invariants.

**INPUT:**

- `invariants` – A list or tuple of values of the three or four invariants. The default option requires the Clebsch invariants $A, B, C$ and $R$ of the binary quintic.
- `K` – The field over which the quintic is defined.
- `invariant_choice` – The type of invariants provided. The accepted options are 'clebsch' and 'default', which are the same. No other options are implemented.
- `scaling` – How the coefficients should be scaled. The accepted values are 'none' for no scaling, 'normalized' to scale in such a way that the resulting coefficients are independent of the scaling of the input invariants and 'coprime' which scales the input invariants by dividing them by their gcd.

**OUTPUT:**

A set of coefficients of a binary quintic, whose invariants are equal to the given invariants up to a scaling.

**EXAMPLES:**

First we check the general case, where the invariant $M$ is non-zero:

```python
sage: R.<x0, x1> = QQ[]
sage: p = 3*x1^5 + 6*x1^4*x0 + 3*x1^3*x0^2 + 4*x1^2*x0^3 - 5*x1*x0^4 + 4*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: invs = quintic.clebsch_invariants(as_tuple=True)
sage: reconstructed = invariant_theory.binary_form_from_invariants(5, invs, variables=quintic.variables()) # indirect doctest
sage: reconstructed
Binary quintic with coefficients (9592267437341790539005557/2441406250000000000000,
214929628207625556323004064707/6103515625000000000000,
11149651890347700974453304786783/7629394531250000000000,
12265077571894638395648891202734239/476837158203125000000000,
32399663994570652847286334593218447/11920928955078125000000000,
1504506503644608395841632538558481466127/149011611193847656250000000)
```

We can see that the invariants of the reconstructed form match the ones of the original form by scaling the invariants $B$ and $C$: 

3.2. Classical Invariant Theory
If we compare the form obtained by this reconstruction to the one found by letting the covariants $\alpha$ and $\beta$ be the coordinates of the form, we find the forms are the same up to a power of the determinant of $\alpha$ and $\beta$:

```
sage: alpha = quintic.alpha_c covariant()
sage: beta = quintic.beta_c covariant()
sage: g = matrix([[alpha(x0=1, x1=0), alpha(x0=0, x1=1)], [beta(x0=1, x1=0), beta(x0=0, x1=1)])]**-1
sage: transformed = tuple([g.determinant()**-5*x for x in quintic.transformed(g).coeffs()])
```

```
sage: transformed == reconstructed.coeffs()
```

This can also be seen by computing the $\alpha$ covariant of the obtained form:

```
sage: reconstructed.alpha_c covariant().coefficient(x1)
0
```

```
sage: reconstructed.alpha_c covariant().coefficient(x0) != 0
```

If the invariant $M$ vanishes, then the coefficients are computed in a different way:

```
sage: [A, B, C] = [3, 1, 2]
sage: M = 2*A*B - 3*C
```

```
sage: from sage.rings.invariants.reconstruction import binary_quintic_coefficients_from_invariants
sage: reconstructed = binary_quintic_coefficients_from_invariants([A, B, C])
sage: reconstructed
(-66741943359375/2097152, -125141143798828125/134217728, 0, 52793920040130615234375/34359738368, 19797720015048980712890625/1099511627776, -445448700338602660400390625/17592186044416)
```

```
sage: newform = sum([reconstructed[i]*x0^i*x1^(5-i) for i in range(6)])
sage: newquintic = invariant_theory.binary_quintic(newform, x0, x1)
sage: scale = 3/newquintic.A_invariant()
```

```
[3, newquintic.B_invariant()^scale^2, newquintic.C_invariant()^scale^3]
```

Several special cases:

```
sage: quintic = invariant_theory.binary_quintic(x0^5 - x1^5, x0, x1)
sage: invs = quintic.clebsch_invariants(as_tuple=True)
sage: binary_quintic_coefficients_from_invariants(invs)
(1, 0, 0, 0, 0, 1)
```
sage: quintic = invariant_theory.binary_quintic(x0^5 + 10*x0^3*x1^2 - 15*x0*x1^4, x0, x1)
sage: invs = quintic.clebsch_invariants(as_tuple=True)
sage: binary_quintic_coefficients_from_invariants(invs)
(1, 0, 10, 0, -15, 0)
sage: quintic = invariant_theory.binary_quintic(x0^5 + x0^3 + x1^3, x0, x1)

For fields of characteristic 2, 3 or 5, there is no reconstruction implemented. This is part of trac ticket #26786.

sage: binary_quintic_coefficients_from_invariants([3,1,2], K=GF(5))
Traceback (most recent call last):
... 
NotImplementedError: no reconstruction of binary quintics implemented for fields of characteristic 2, 3 or 5

3.3 Educational Versions of Groebner Basis Related Algorithms

3.3.1 Educational versions of Groebner basis algorithms

Following [BW1993], the original Buchberger algorithm (algorithm GROEBNER in [BW1993]) and an improved version of Buchberger's algorithm (algorithm GROEBNERNEW2 in [BW1993]) are implemented.

No attempt was made to optimize either algorithm as the emphasis of these implementations is a clean and easy presentation. To compute a Groebner basis most efficiently in Sage, use the MPolynomialIdeal.groebner_basis() method on multivariate polynomial objects instead.

Note: The notion of ‘term’ and ‘monomial’ in [BW1993] is swapped from the notion of those words in Sage (or the other way around, however you prefer it). In Sage a term is a monomial multiplied by a coefficient, while in [BW1993] a monomial is a term multiplied by a coefficient. Also, what is called LM (the leading monomial) in Sage is called HT (the head term) in [BW1993].

EXAMPLES:

Consider Katsura-6 with respect to a degrevlex ordering.

sage: from sage.rings.polynomial.toy_buchberger import *
sage: P.<a,b,c,e,f,g,h,i,j,k> = PolynomialRing(GF(32003))
sage: I = sage.rings.ideal.Katsura(P, 6)

(continues on next page)
All algorithms actually compute a Groebner basis:

\begin{verbatim}
sage: Ideal(g1).basis_is_groebner()
True
sage: Ideal(g2).basis_is_groebner()
True
sage: Ideal(g3).basis_is_groebner()
True
\end{verbatim}

The results are correct:

\begin{verbatim}
sage: Ideal(g1) == Ideal(g2) == Ideal(g3)
True
\end{verbatim}

If `get_verbose()` is $\geq 1$, a protocol is provided:

\begin{verbatim}
sage: from sage.misc.verbose import set_verbose
sage: set_verbose(1)

sage: P.<a,b,c> = PolynomialRing(GF(127))

sage: I = sage.rings.ideal.Katsura(P)
// sage... ideal

sage: I
Ideal (a + 2*b + 2*c - 1, a^2 + 2*b^2 + 2*c^2 - a, 2*a*b + 2*b*c - b) of Multivariate Polynomial Ring in a, b, c over Finite Field of size 127

sage: buchberger(I) # random
(a + 2*b^2 + 2*c^2 - a, a^2 + 2*b^2 + 2*c^2 - a) => -2*b^2 - 6*b*c - 6*c^2 + b + 2*c
G: set([a + 2*b + 2*c - 1, 2*a*b + 2*b*c - b, a^2 + 2*b^2 + 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c])

(a^2 + 2*b^2 + 2*c^2 - a, a + 2*b + 2*c - 1) => 0
G: set([a + 2*b + 2*c - 1, 2*a*b + 2*b*c - b, a^2 + 2*b^2 + 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c])

(a + 2*b + 2*c - 1, 2*a*b + 2*b*c - b) => -5*b*c - 6*c^2 - 63*b + 2*c
G: set([a + 2*b + 2*c - 1, 2*a*b + 2*b*c - b, -5*b*c - 6*c^2 - 63*b + 2*c, a^2 + 2*b^2 + 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c])

(2*a*b + 2*b*c - b, a + 2*b + 2*c - 1) => 0
G: set([a + 2*b + 2*c - 1, 2*a*b + 2*b*c - b, -5*b*c - 6*c^2 - 63*b + 2*c, a^2 + 2*b^2 + 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c])

(2*a*b + 2*b*c - b, -5*b*c - 6*c^2 - 63*b + 2*c) => -22*c^3 + 24*c^2 - 60*b - 62*c
G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b*c - b, a^2 + 2*b^2 + 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c])

(2*a*b + 2*b*c - b, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c) => 0
G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b*c - b, a^2 + 2*b^2 + 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c])
\end{verbatim}  

(continues on next page)
(2*a*b + 2*b^2*c - b, a^2 + 2*b^2 + 2*c^2 - a) => 0
G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b^2*c - b, a^2 + 2*b^2 - 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c, -5*b*c - 6*c^2 - 63*b + 2*c])

(a + 2*b + 2*c - 1, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c) => 0
G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b^2*c - b, a^2 + 2*b^2 - 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c, -5*b*c - 6*c^2 - 63*b + 2*c])

(a^2 + 2*b^2 + 2*c^2 - a, 2*a*b + 2*b^2*c - b) => 0
G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b^2*c - b, a^2 + 2*b^2 - 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c, -5*b*c - 6*c^2 - 63*b + 2*c])

(-2*b^2 - 6*b*c - 6*c^2 + b + 2*c, -5*b*c - 6*c^2 - 63*b + 2*c) => 0
G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b^2*c - b, a^2 + 2*b^2 - 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c, -5*b*c - 6*c^2 - 63*b + 2*c])

(a + 2*b + 2*c - 1, -5*b*c - 6*c^2 - 63*b + 2*c) => 0
G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b^2*c - b, a^2 + 2*b^2 - 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c, -5*b*c - 6*c^2 - 63*b + 2*c])

(a^2 + 2*b^2 + 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c, -5*b*c - 6*c^2 - 63*b + 2*c) => 0
G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b^2*c - b, a^2 + 2*b^2 - 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c, -5*b*c - 6*c^2 - 63*b + 2*c])

(-2*b^2 - 6*b*c - 6*c^2 + b + 2*c, -22*c^3 + 24*c^2 - 60*b - 62*c) => 0
G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b^2*c - b, a^2 + 2*b^2 - 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c, -5*b*c - 6*c^2 - 63*b + 2*c])

(a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b^2*c - b, a^2 + 2*b^2 - 2*c^2 - a) => 0
G: set([a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b^2*c - b, a^2 + 2*b^2 - 2*c^2 - a, -2*b^2 - 6*b*c - 6*c^2 + b + 2*c, -5*b*c - 6*c^2 - 63*b + 2*c])

15 reductions to zero.
[a + 2*b + 2*c - 1, -22*c^3 + 24*c^2 - 60*b - 62*c, 2*a*b + 2*b^2*c - b, a^2 + 2*b^2 - 2*c^2 - a]
The original Buchberger algorithm performs 15 useless reductions to zero for this example:

```plaintext
sage: gb = buchberger(I)
...  
15 reductions to zero.
```

The ‘improved’ Buchberger algorithm in contrast only performs 1 reduction to zero:

```plaintext
sage: gb = buchberger_improved(I)
...  
1 reductions to zero.
sage: sorted(gb)
[a + 2*b + 2*c - 1, b*c + 52*c^2 + 38*b + 25*c, b^2 - 26*c^2 - 51*b + 51*c, c^3 + 22*c^2 - 55*b + 49*c]
```

AUTHORS:
- Marshall Hampton (2009-07-08): some doctest additions

sage.rings.polynomial.toy_buchberger.LCM(f, g)
sage.rings.polynomial.toy_buchberger.LM(f)
sage.rings.polynomial.toy_buchberger.LT(f)
sage.rings.polynomial.toy_buchberger.buchberger(F)

Compute a Groebner basis using the original version of Buchberger’s algorithm as presented in [BW1993], page 214.

INPUT:
- F – an ideal in a multivariate polynomial ring

OUTPUT: a Groebner basis for F

Note: The verbosity of this function may be controlled with a set_verbose() call. Any value >=1 will result in this function printing intermediate bases.

EXAMPLES:

```plaintext
sage: from sage.rings.polynomial.toy_buchberger import buchberger
sage: R.<x,y,z> = PolynomialRing(QQ)
sage: I = R.ideal([x^2 - z - 1, z^2 - y - 1, x*y^2 - x - 1])
sage: set_verbose(0)
sage: gb = buchberger(I)
sage: gb.is_groebner()  
True
sage: gb.ideal() == I  
True
```

sage.rings.polynomial.toy_buchberger.buchberger_improved(F)

Compute a Groebner basis using an improved version of Buchberger’s algorithm as presented in [BW1993], page 232.

This variant uses the Gebauer-Moeller Installation to apply Buchberger’s first and second criterion to avoid useless pairs.
INPUT:
- \( F \) – an ideal in a multivariate polynomial ring

OUTPUT: a Groebner basis for \( F \)

Note: The verbosity of this function may be controlled with a `setVerbose()` call. Any value \( \geq 1 \) will result in this function printing intermediate Groebner bases.

EXAMPLES:

```python
sage: from sage.rings.polynomial.toy_buchberger import buchberger_improved
sage: R.<x,y,z> = PolynomialRing(QQ)

sage: set_verbose(0)

sage: sorted(buchberger_improved(R.ideal([x^4 - y - z, x*y*z - 1])))
[x^4*y*z - 1, x^3 - y^2*z - y*z^2, y^3*z^2 + y^2*z^3 - x^2]
```

`sage.rings.polynomial.toy_buchberger.inter_reduction(Q)`
Compute inter-reduced polynomials from a set of polynomials.

INPUT:
- \( Q \) – a set of polynomials

OUTPUT: if \( Q \) is the set \((f_1, \ldots, f_n)\), this method returns \((g_1, \ldots, g_s)\) such that:
- \( \langle f_1, \ldots, f_n \rangle = \langle g_1, \ldots, g_s \rangle \)
- \( LM(g_i) \neq LM(g_j) \) for all \( i \neq j \)
- \( LM(g_i) \) does not divide \( m \) for all monomials \( m \) of \( \{g_1, \ldots, g_{i-1}, g_{i+1}, \ldots, g_s\} \)
- \( LC(g_i) = 1 \) for all \( i \).

EXAMPLES:

```python
sage: from sage.rings.polynomial.toy_buchberger import inter_reduction

sage: inter_reduction(set())
set()

sage: P.<x,y> = QQ[]

sage: reduced = inter_reduction(set([x^2 - 5*y^2, x^3]))

sage: reduced == set([x*y^2, x^2-5*y^2])
True

sage: reduced == inter_reduction(set([2*(x^2 - 5*y^2), x^3]))
True
```

`sage.rings.polynomial.toy_buchberger.select(P)`
Select a polynomial using the normal selection strategy.

INPUT:
- \( P \) – a list of critical pairs

OUTPUT: an element of \( P \)

EXAMPLES:
```python
sage: from sage.rings.polynomial.toy_buchberger import select
sage: R.<x,y,z> = PolynomialRing(QQ, order='lex')
sage: ps = [x^3 - z -1, z^3 - y -1, x^5 - y -2]
sage: pairs = [[ps[i], ps[j]] for i in range(3) for j in range(i+1, 3)]
sage: select(pairs)
[x^3 - z - 1, -y + z^3 - 1]
```

**sage.rings.polynomial.toy_buchberger.spol(f, g)**

Compute the S-polynomial of f and g.

**INPUT:**

- f, g – polynomials

**OUTPUT:** the S-polynomial of f and g

**EXAMPLES:**

```python
sage: R.<x,y,z> = PolynomialRing(QQ)
sage: from sage.rings.polynomial.toy_buchberger import spol
sage: spol(x^2 - z - 1, z^2 - y - 1)
x^2*y - z^3 + x^2 - z^2
```

```python
sage: G, B = update(set(), set(), x*y*z)
set()
```

**sage.rings.polynomial.toy_buchberger.update(G, B, h)**

Update G using the set of critical pairs B and the polynomial h as presented in [BW1993], page 230. For this, Buchberger's first and second criterion are tested.

This function implements the Gebauer-Moeller Installation.

**INPUT:**

- G – an intermediate Groebner basis
- B – a set of critical pairs
- h – a polynomial

**OUTPUT:** a tuple of

- an intermediate Groebner basis
- a set of critical pairs

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.toy_buchberger import update
sage: R.<x,y,z> = PolynomialRing(QQ)
sage: set_verbose(0)
sage: G, B = update(set(), set(), x*y*z - 1)
sage: G, B
(set(), {(x*y*z - 1, x*y^2 - 1)})
```

---

**Chapter 3. Multivariate Polynomials**

---
3.3.2 Educational versions of Groebner basis algorithms: triangular factorization

In this file is the implementation of two algorithms in [Laz1992].

The main algorithm is Triangular; a secondary algorithm, necessary for the first, is ElimPolMin. As per Lazard’s formulation, the implementation works with any term ordering, not only lexicographic.

Lazard does not specify a few of the subalgorithms implemented as the functions

- is_triangular,
- is_linearly_dependent, and
- linear_representation.

The implementations are not hard, and the choice of algorithm is described with the relevant function.

No attempt was made to optimize these algorithms as the emphasis of this implementation is a clean and easy presentation.

Examples appear with the appropriate function.

AUTHORS:

- John Perry (2009-02-24): initial version, but some words of documentation were stolen shamelessly from Martin Albrecht’s toy_buchberger.py.

sage.rings.polynomial.toy_variety.coefficient_matrix(polys)
Generates the matrix $M$ whose entries are the coefficients of $polys$. The entries of row $i$ of $M$ consist of the coefficients of $polys[i]$.

INPUT:

- polys - a list/tuple of polynomials

OUTPUT:

A matrix $M$ of the coefficients of $polys$.

EXAMPLES:

```
sage: from sage.rings.polynomial.toy_variety import coefficient_matrix
sage: R.<x,y> = PolynomialRing(QQ)
sage: coefficient_matrix([x^2 + 1, y^2 + 1, x*y + 1])
[1 0 0 1]
[0 0 1 1]
[0 1 0 1]
```

**Note:** This function may be merged with `sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic.coefficient_matrix()` in the future.

sage.rings.polynomial.toy_variety.elim_pol(B, n=-1)
Find the unique monic polynomial of lowest degree and lowest variable in the ideal described by $B$.

For the purposes of the triangularization algorithm, it is necessary to preserve the ring, so $n$ specifies which variable to check. By default, we check the last one, which should also be the smallest.

The algorithm may not work if you are trying to cheat: $B$ should describe the Groebner basis of a zero-dimensional ideal. However, it is not necessary for the Groebner basis to be lexicographic.

The algorithm is taken from a 1993 paper by Lazard [Laz1992].

INPUT:
• \( B \) - a list/tuple of polynomials or a multivariate polynomial ideal
• \( n \) - the variable to check (see above) (default: -1)

EXAMPLES:

```python
sage: from sage.misc.verbose import set_verbose
sage: set_verbose(0)

sage: from sage.rings.polynomial.toy_variety import elim_pol

sage: R.<x,y,z> = PolynomialRing(GF(32003))

sage: p1 = x^2*(x-1)^3*y^2*(z-3)^3
sage: p2 = z^2 - z
sage: p3 = (x-2)^2*(y-1)^3

sage: I = R.ideal(p1,p2,p3)

sage: elim_pol(I.groebner_basis())
z^2 - z
```

sage.rings.polynomial.toy_variety.is_linearly_dependent(polys)

Decides whether the polynomials of \( \text{polys} \) are linearly dependent. Here \( \text{polys} \) is a collection of polynomials.

The algorithm creates a matrix of coefficients of the monomials of \( \text{polys} \). It computes the echelon form of the matrix, then checks whether any of the rows is the zero vector.

Essentially this relies on the fact that the monomials are linearly independent, and therefore is building a linear map from the vector space of the monomials to the canonical basis of \( \mathbb{R}^n \), where \( n \) is the number of distinct monomials in \( \text{polys} \). There is a zero vector iff there is a linear dependence among \( \text{polys} \).

The case where \( \text{polys} = [] \) is considered to be not linearly dependent.

INPUT:

• \( \text{polys} \) - a list/tuple of polynomials

OUTPUT:

True if the elements of \( \text{polys} \) are linearly dependent; False otherwise.

EXAMPLES:

```python
sage: from sage.rings.polynomial.toy_variety import is_linearly_dependent

sage: R.<x,y> = PolynomialRing(QQ)

sage: B = [x^2 + 1, y^2 + 1, x*y + 1]


sage: is_linearly_dependent(B + [p])
True

sage: p = x*B[0]

sage: is_linearly_dependent(B + [p])
False

sage: is_linearly_dependent([])
False
```

sage.rings.polynomial.toy_variety.is_triangular(B)

Check whether the basis \( B \) of an ideal is triangular. That is: check whether the largest variable in \( B[i] \) with respect to the ordering of the base ring \( R \) is \( R.gens()[i] \).

The algorithm is based on the definition of a triangular basis, given by Lazard in 1992 in [Laz1992].

INPUT:

• \( B \) - a list/tuple of polynomials or a multivariate polynomial ideal
OUTPUT:

True if the basis is triangular; False otherwise.

EXAMPLES:

```python
sage: from sage.rings.polynomial.toy_variety import is_triangular
sage: R.<x,y,z> = PolynomialRing(QQ)
```

```
sage: p1 = x^2*y + z^2
sage: p2 = y*z + z^3
sage: p3 = y + z
sage: is_triangular(R.ideal(p1,p2,p3))
False
sage: p3 = z^2 - 3
sage: is_triangular(R.ideal(p1,p2,p3))
True
```

**sage.rings.polynomial.toy_variety.linear_representation(p, polys)**

Assuming that \( p \) is a linear combination of \( \text{polys} \), determine coefficients that describe the linear combination. This probably does not work for any inputs except \( p \), a polynomial, and \( \text{polys} \), a sequence of polynomials. If \( p \) is not in fact a linear combination of \( \text{polys} \), the function raises an exception.

The algorithm creates a matrix of coefficients of the monomials of \( \text{polys} \) and \( p \), with the coefficients of \( p \) in the last row. It augments this matrix with the appropriate identity matrix, then computes the echelon form of the augmented matrix. The last row should contain zeroes in the first columns, and the last columns contain a linear dependence relation. Solving for the desired linear relation is straightforward.

**INPUT:**

- \( p \) - a polynomial
- \( \text{polys} \) - a list/tuple of polynomials

**OUTPUT:**

If \( n == \text{len}(\text{polys}) \), returns \([a[0],a[1],...,a[n-1]]\) such that \( p == a[0]*\text{poly[0]} + ... + a[n-1]*\text{poly[n-1]} \).

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.toy_variety import linear_representation
sage: R.<x,y> = PolynomialRing(GF(32003))
sage: B = [x^2 + 1, y^2 + 1, x*y + 1]
sage: linear_representation(p, B)
[3, 32001, 1]
```

**sage.rings.polynomial.toy_variety.triangular_factorization(B, n=-1)**

Compute the triangular factorization of the Groebner basis \( B \) of an ideal.

This will not work properly if \( B \) is not a Groebner basis!

The algorithm used is that described in a 1992 paper by Daniel Lazard [Laz1992]. It is not necessary for the term ordering to be lexicographic.

**INPUT:**

- \( B \) - a list/tuple of polynomials or a multivariate polynomial ideal
- \( n \) - the recursion parameter (default: -1)

3.3. Educational Versions of Groebner Basis Related Algorithms 491
A list $T$ of triangular sets $T_0, T_1, \ldots$.

**EXAMPLES:**

```python
sage: from sage.misc.verbose import set_verbose
sage: set_verbose(0)
sage: from sage.rings.polynomial.toy_variety import triangular_factorization
sage: R.<x,y,z> = PolynomialRing(GF(32003))
sage: p1 = x^2*(x-1)^3*y^2*(z-3)^3
sage: p2 = z^2 - z
sage: p3 = (x-2)^2*(y-1)^3
sage: I = R.ideal(p1,p2,p3)
sage: triangular_factorization(I.groebner_basis())
[[x^2 - 4*x + 4, y, z],
 [x^5 - 3*x^4 + 3*x^3 - x^2, y - 1, z],
 [x^2 - 4*x + 4, y, z - 1],
 [x^5 - 3*x^4 + 3*x^3 - x^2, y - 1, z - 1]]
```

### 3.3.3 Educational version of the $d$-Groebner basis algorithm over PIDs

No attempt was made to optimize this algorithm as the emphasis of this implementation is a clean and easy presentation.

**Note:** The notion of ‘term’ and ‘monomial’ in [BW1993] is swapped from the notion of those words in Sage (or the other way around, however you prefer it). In Sage a term is a monomial multiplied by a coefficient, while in [BW1993] a monomial is a term multiplied by a coefficient. Also, what is called LM (the leading monomial) in Sage is called HT (the head term) in [BW1993].

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.toy_d_basis import d_basis
First, consider an example from arithmetic geometry:
sage: A.<x,y> = PolynomialRing(ZZ, 2)
sage: B.<X,Y> = PolynomialRing(Rationals(),2)
sage: f = -y^2 - y + x^3 + 7*x + 1
sage: fx = f.derivative(x)
sage: fy = f.derivative(y)
sage: I = B.ideal([B(f),B(fx),B(fy)])
sage: I.groebner_basis()
[1]
```

Since the output is 1, we know that there are no generic singularities.

To look at the singularities of the arithmetic surface, we need to do the corresponding computation over $\mathbb{Z}$:

```python
sage: I = A.ideal([f,fx,fy])
sage: gb = d_basis(I); gb
[x - 2020, y - 11313, 22627]
sage: gb[-1].factor()
11^3 * 17
```
This Groebner Basis gives a lot of information. First, the only fibers (over $\mathbb{Z}$) that are not smooth are at $11 = 0$, and $17 = 0$. Examining the Groebner Basis, we see that we have a simple node in both the fiber at 11 and at 17. From the factorization, we see that the node at 17 is regular on the surface (an $I_1$ node), but the node at 11 is not. After blowing up this non-regular point, we find that it is an $I_3$ node.

Another example. This one is from the Magma Handbook:

```
sage: P.<x, y, z> = PolynomialRing(IntegerRing(), 3, order='lex')
sage: I = ideal( x^2 - 1, y^2 - 1, 2*x*y - z)
sage: I = Ideal(d_basis(I))
sage: x.reduce(I)
x
sage: (2*x).reduce(I)
y*z
```

To compute modulo 4, we can add the generator 4 to our basis:

```
sage: I = ideal( x^2 - 1, y^2 - 1, 2*x*y - z, 4)
sage: gb = d_basis(I)
sage: R = P.change_ring(IntegerModRing(4))
sage: gb = [R(f) for f in gb if R(f)]; gb
[x^2 - 1, x*z + 2*y, 2*x - y*z, y^2 - 1, z^2, 2*z]
```

A third example is also from the Magma Handbook.

This example shows how one can use Groebner bases over the integers to find the primes modulo which a system of equations has a solution, when the system has no solutions over the rationals.

We first form a certain ideal $I$ in $\mathbb{Z}[x,y,z]$, and note that the Groebner basis of $I$ over $\mathbb{Q}$ contains 1, so there are no solutions over $\mathbb{Q}$ or an algebraic closure of it (this is not surprising as there are 4 equations in 3 unknowns).

```
sage: gb = d_basis(I); gb
[z ..., y ..., x ..., 282687803443]
```

Now for each prime $p$ dividing this integer 282687803443, the Groebner basis of $I$ modulo $p$ will be non-trivial and will thus give a solution of the original system modulo $p$.

```
sage: factor(282687803443)
101 * 103 * 27173681
```

```
sage: I.change_ring( P.change_ring( GF(101) ) ).groebner_basis()
[z - 33, y + 48, x + 19]
```

```
sage: I.change_ring( P.change_ring( GF(103) ) ).groebner_basis()
[z - 18, y + 8, x + 39]
```

```
sage: I.change_ring( P.change_ring( GF(27173681) ) ).groebner_basis()
[z + 10380032, y + 3186055, x - 536027]
```

3.3. Educational Versions of Groebner Basis Related Algorithms 493
Of course, modulo any other prime the Groebner basis is trivial so there are no other solutions. For example:

```python
sage: I.change_ring( P.change_ring( GF(3) ) ).groebner_basis()
[1]
```

**AUTHOR:**

- Martin Albrecht (2008-08): initial version

```python
sage.rings.polynomial.toy_d_basis.LC(f)
sage.rings.polynomial.toy_d_basis.LM(f)
sage.rings.polynomial.toy_d_basis.d_basis(F, strat=True)
```

Return the \( d \)-basis for the Ideal \( F \) as defined in [BW1993].

**INPUT:**

- \( F \) – an ideal
- \( \text{strat} \) – use update strategy (default: True)

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.toy_d_basis import d_basis
sage: A.<x,y> = PolynomialRing(ZZ, 2)
sage: f = -y^2 - y + x^3 + 7*x + 1
sage: fx = f.derivative(x)
sage: fy = f.derivative(y)
sage: I = A.ideal([f,fx,fy])
sage: gb = d_basis(I); gb
[x - 2020, y - 11313, 22627]
```

```python
sage.rings.polynomial.toy_d_basis.gpol(g1, g2)
```

Return the G-Polynomial of \( g_1 \) and \( g_2 \).

Let \( a_i t_i \) be \( \text{LT}(g_i) \), \( a_i = a_i c_i + a_j c_j \) with \( a = \text{GCD}(a_i, a_j) \), and \( s_i = t_i t_i \) with \( t = \text{LCM}(t_i, t_j) \). Then the G-Polynomial is defined as: \( c_1 s_1 g_1 - c_2 s_2 g_2 \).

**INPUT:**

- \( g1 \) – polynomial
- \( g2 \) – polynomial

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.toy_d_basis import gpol
sage: P.<x, y, z> = PolynomialRing(IntegerRing(), 3, order='lex')
sage: f = x^2 - 1
sage: g = 2*x*y - z
sage: gpol(f,g)
x^2*y - y
```

```python
sage.rings.polynomial.toy_d_basis.select(P)
```

The normal selection strategy.

**INPUT:**

- \( P \) – a list of critical pairs
OUTPUT:

an element of P

EXAMPLES:

```python
sage: from sage.rings.polynomial.toy_d_basis import select
sage: A.<x,y> = PolynomialRing(ZZ, 2)
sage: f = -y^2 - y + x^3 + 7*x + 1
sage: fx = f.derivative(x)
sage: fy = f.derivative(y)
sage: G = [f, fx, fy]
sage: B = set((f1, f2) for f1 in G for f2 in G if f1 != f2)
sage: select(B)
(-2*y - 1, 3*x^2 + 7)
```

sage.rings.polynomial.toy_d_basis.spol(g1, g2)

Return the S-Polynomial of \(g_1\) and \(g_2\).

Let \(a_i t_i\) be \(LT(g_i)\), \(b_i = a_i / a_i\) with \(a = LCM(a_i, a_j)\), and \(s_i = t_i / t_i\) with \(t = LCM(t_i, t_j)\). Then the S-Polynomial is defined as: \(b_1 s_1 g_1 - b_2 s_2 g_2\).

INPUT:

- \(g1\) – polynomial
- \(g2\) – polynomial

EXAMPLES:

```python
sage: from sage.rings.polynomial.toy_d_basis import spol
sage: P.<x, y, z> = PolynomialRing(IntegerRing(), 3, order='lex')
sage: f = x^2 - 1
sage: g = 2*x*y - z
sage: spol(f,g)
x*z - 2*y
```

sage.rings.polynomial.toy_d_basis.update(G, B, h)

Update \(G\) using the list of critical pairs \(B\) and the polynomial \(h\) as presented in [BW1993], page 230. For this, Buchberger's first and second criterion are tested.

This function uses the Gebauer-Moeller Installation.

INPUT:

- \(G\) – an intermediate Groebner basis
- \(B\) – a list of critical pairs
- \(h\) – a polynomial

OUTPUT:

\(G, B\) where \(G\) and \(B\) are updated

EXAMPLES:

```python
sage: from sage.rings.polynomial.toy_d_basis import update
sage: A.<x,y> = PolynomialRing(ZZ, 2)
sage: G = set([3*x^2 + 7, 2*y + 1, x^3 - y^2 + 7*x - y + 1])
sage: B = set([[]])
```

(continues on next page)
\begin{verbatim}
sage: h = x^2*y - x^2 + y - 3
sage: update(G,B,h)
(({2*y + 1, 3*x^2 + 7, x^2*y - x^2 + y - 3, x^3 - y^2 + 7*x - y + 1},
  {(x^2*y - x^2 + y - 3, 2*y + 1),
   (x^2*y - x^2 + y - 3, 3*x^2 + 7),
   (x^2*y - x^2 + y - 3, x^3 - y^2 + 7*x - y + 1)}
)\end{verbatim}
4.1 Fraction Field of Integral Domains

AUTHORS:

- William Stein (with input from David Joyner, David Kohel, and Joe Wetherell)
- Burcin Erocal
- Julian Rüth (2017-06-27): embedding into the field of fractions and its section

EXAMPLES:

Quotienting is a constructor for an element of the fraction field:

```
sage: R.<x> = QQ[]
sage: (x^2-1)/(x+1)
x - 1
sage: parent((x^2-1)/(x+1))
Fraction Field of Univariate Polynomial Ring in x over Rational Field
```

The GCD is not taken (since it doesn’t converge sometimes) in the inexact case:

```
sage: Z.<z> = CC[]
sage: I = CC.gen()
sage: (1+I+z)/(z+0.1*I)
(z + 1.00000000000000 + I)/(z + 0.100000000000000*I)
sage: (1+I*z)/(z+1.1)
(I*z + 1.00000000000000)/(z + 1.10000000000000)
```

```
sage.rings.fraction_field.FractionField(R, names=None)
Create the fraction field of the integral domain R.

INPUT:

- R – an integral domain
- names – ignored

EXAMPLES:

We create some example fraction fields:

```
sage: FractionField(IntegerRing())
Rational Field
sage: FractionField(PolynomialRing(RationalField(),'x'))
```
```
Fraction Field of Univariate Polynomial Ring in \( x \) over Rational Field

\[\text{sage: } \text{FractionField(PolynomialRing(IntegerRing(),'x'))}\]

Fraction Field of Univariate Polynomial Ring in \( x \) over Integer Ring

\[\text{sage: } \text{FractionField(PolynomialRing(RationalField(),2,'x'))}\]

Fraction Field of Multivariate Polynomial Ring in \( x_0, x_1 \) over Rational Field

Dividing elements often implicitly creates elements of the fraction field:

\[\text{sage: } x = \text{PolynomialRing(RationalField(), 'x').gen()}\]
\[\text{sage: } f = x/(x+1)\]
\[\text{sage: } g = x^3/(x+1)\]
\[\text{sage: } f/g\]
\[1/x^2\]
\[\text{sage: } g/f\]
\[x^2\]

The input must be an integral domain:

\[\text{sage: } \text{Frac(Integers(4))}\]

Traceback (most recent call last):
...
TypeError: R must be an integral domain.

class `sage.rings.fraction_field.FractionFieldEmbedding`

Bases: `sage.structure.coerce_maps.DefaultConvertMap_unique`

The embedding of an integral domain into its field of fractions.

**EXAMPLES:**

\[\text{sage: } R.<x> = \text{QQ}[]\]
\[\text{sage: } f = R.\text{fraction_field().coerce_map_from}(R); f\]

Coercion map:
From: Univariate Polynomial Ring in \( x \) over Rational Field
To: Fraction Field of Univariate Polynomial Ring in \( x \) over Rational Field

is_injective()

Return whether this map is injective.

**EXAMPLES:**

The map from an integral domain to its fraction field is always injective:

\[\text{sage: } R.<x> = \text{QQ}[]\]
\[\text{sage: } R.\text{fraction_field().coerce_map_from}(R).\text{is_injective()}\]

True

is_surjective()

Return whether this map is surjective.

**EXAMPLES:**

\[\text{sage: } R.<x> = \text{QQ}[]\]
\[\text{sage: } R.\text{fraction_field().coerce_map_from}(R).\text{is_surjective()}\]

False
section()

Return a section of this map.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: R.fraction_field().coerce_map_from(R).section()
Section map:
  From: Fraction Field of Univariate Polynomial Ring in x over Rational Field
  To:   Univariate Polynomial Ring in x over Rational Field
```

class sage.rings.fraction_field.FractionFieldEmbeddingSection

Bases: sage.categories.map.Section

The section of the embedding of an integral domain into its field of fractions.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = R.fraction_field().coerce_map_from(R).section(); f
Section map:
  From: Fraction Field of Univariate Polynomial Ring in x over Rational Field
  To:   Univariate Polynomial Ring in x over Rational Field
```

class sage.rings.fraction_field.FractionField_1poly_field

Bases: sage.rings.fraction_field.FractionField_generic

The fraction field of a univariate polynomial ring over a field.

Many of the functions here are included for coherence with number fields.

class_number()

Here for compatibility with number fields and function fields.

EXAMPLES:

```
sage: R.<t> = GF(5)[]; K = R.fraction_field()
sage: K.class_number()
1
```

function_field()

Return the isomorphic function field.

EXAMPLES:

```
sage: R.<t> = GF(5)[]
sage: K = R.fraction_field()
sage: K.function_field()
Rational function field in t over Finite Field of size 5
```

See also:

sage.rings.function_field.RationalFunctionField.field()

maximal_order()

Return the maximal order in this fraction field.

EXAMPLES:
```
sage: K = FractionField(GF(5)['t'])
sage: K.maximal_order()
Univariate Polynomial Ring in t over Finite Field of size 5
```

**ring_of_integers()**

Return the ring of integers in this fraction field.

**EXAMPLES:**

```
sage: K = FractionField(GF(5)['t'])
sage: K.ring_of_integers()
Univariate Polynomial Ring in t over Finite Field of size 5
```

```python
class sage.rings.fraction_field.FractionField_generic(R, element_class=<class 'sage.rings.fraction_field_element.FractionFieldElement'>, category=Category of quotient fields)
Bases: sage.rings.ring.Field
The fraction field of an integral domain.

**base_ring()**

Return the base ring of self.

This is the base ring of the ring which this fraction field is the fraction field of.

**EXAMPLES:**

```
sage: R = Frac(ZZ['t'])
sage: R.base_ring()
Integer Ring
```

**characteristic()**

Return the characteristic of this fraction field.

**EXAMPLES:**

```
sage: R = Frac(ZZ['t'])
sage: R.base_ring()
Integer Ring
sage: R = Frac(ZZ['t']); R.characteristic()
0
sage: R = Frac(GF(5)['w']); R.characteristic()
5
```

**construction()**

**EXAMPLES:**

```
sage: Frac(ZZ['x']).construction()
(FractionField, Univariate Polynomial Ring in x over Integer Ring)
sage: K = Frac(GF(3)['t'])
sage: f, R = K.construction()
sage: f(R)
Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 3
sage: f(R) == K
True
```

Chapter 4. Rational Functions
gen\((i=0)\)

Return the \(i\)-th generator of self.

EXAMPLES:

```python
sage: R = Frac(PolynomialRing(QQ, 'z', 10)); R
Fraction Field of Multivariate Polynomial Ring in z0, z1, z2, z3, z4, z5, z6, z7, z8, z9 over Rational Field
sage: R.0
z0
sage: R.gen(3)
z3
sage: R.3
z3
```

is_exact()

Return if self is exact which is if the underlying ring is exact.

EXAMPLES:

```python
sage: Frac(ZZ['x']).is_exact()
True
sage: Frac(CDF['x']).is_exact()
False
```

is_field\((proof=True)\)

Return True, since the fraction field is a field.

EXAMPLES:

```python
sage: Frac(ZZ).is_field()
True
```

is_finite()

Tells whether this fraction field is finite.

**Note:** A fraction field is finite if and only if the associated integral domain is finite.

EXAMPLES:

```python
sage: Frac(QQ['a','b','c']).is_finite()
False
```

ngens()

This is the same as for the parent object.

EXAMPLES:

```python
sage: R = Frac(PolynomialRing(QQ, 'z', 10)); R
Fraction Field of Multivariate Polynomial Ring in z0, z1, z2, z3, z4, z5, z6, z7, z8, z9 over Rational Field
sage: R.ngens()
10
```

random_element\((\ast\text{args}, \ast\text{kwds})\)

Return a random element in this fraction field.
The arguments are passed to the random generator of the underlying ring.

EXAMPLES:

```python
sage: F = ZZ['x'].fraction_field()
sage: F.random_element()  # random
(2*x - 8)/(-x^2 + x)
```

```python
sage: f = F.random_element(degree=5)
sage: f.numerator().degree()
5
sage: f.denominator().degree()
5
```

**ring()**

Return the ring that this is the fraction field of.

EXAMPLES:

```python
sage: R = Frac(QQ['x,y'])
sage: R
Fraction Field of Multivariate Polynomial Ring in x, y over Rational Field
sage: R.ring()
Multivariate Polynomial Ring in x, y over Rational Field
```

**some_elements()**

Return some elements in this field.

EXAMPLES:

```python
sage: R.<x> = QQ[]
sage: R.fraction_field().some_elements()
[0,
 1,
x,
2*x,
x/(x^2 + 2*x + 1),
1/x^2,
...
(2*x^2 + 2)/(x^2 + 2*x + 1),
(2*x^2 + 2)/x^3,
(2*x^2 + 2)/(x^2 - 1),
2]
```

sage.rings.fraction_field.is_FractionField(x)

Test whether or not x inherits from FractionField_generic.

EXAMPLES:

```python
sage: from sage.rings.fraction_field import is_FractionField
sage: is_FractionField(Frac(ZZ['x']))
True
sage: is_FractionField(QQ)
False
```
4.2 Fraction Field Elements

AUTHORS:

- William Stein (input from David Joyner, David Kohel, and Joe Wetherell)
- Sebastian Pancratz (2010-01-06): Rewrite of addition, multiplication and derivative to use Henrici’s algorithms [Hor1972]

class sage.rings.fraction_field_element.FractionFieldElement
  Bases: sage.structure.element.FieldElement

EXAMPLES:

```python
sage: K = FractionField(PolynomialRing(QQ, 'x'))
sage: K
Fraction Field of Univariate Polynomial Ring in x over Rational Field
sage: loads(K.dumps()) == K
True
sage: x = K.gen()
sage: f = (x^3 + x)/(17 - x^19); f
(-x^3 - x)/(x^19 - 17)
sage: loads(f.dumps()) == f
True
```

denominator()
Return the denominator of self.

EXAMPLES:

```python
sage: R.<x,y> = ZZ[]
sage: f = x/y+1; f
(x + y)/y
sage: f.denominator()
y
```

is_one()
Return True if this element is equal to one.

EXAMPLES:

```python
sage: F = ZZ['x,y'].fraction_field()
sage: x,y = F.gens()
sage: (x/x).is_one()
True
sage: (x/y).is_one()
False
```

is_square(root=False)
Return whether or not self is a perfect square.

If the optional argument root is True, then also returns a square root (or None, if the fraction field element is not square).

INPUT:

- root – whether or not to also return a square root (default: False)

OUTPUT:
• `bool` - whether or not a square
• `object` - (optional) an actual square root if found, and None otherwise.

EXAMPLES:

```python
sage: R.<t> = QQ[]
sage: (1/t).is_square()  # False
sage: (1/t^6).is_square()  # True
sage: ((1+t)^4/t^6).is_square()  # True
sage: (4*(1+t)^4/t^6).is_square()  # True
sage: (2*(1+t)^4/t^6).is_square()  # False
sage: ((1+t)/t^6).is_square()  # False
sage: (4*(1+t)^4/t^6).is_square(root=True)  # (True, (2*t^2 + 4*t + 2)/t^3)
sage: (2*(1+t)^4/t^6).is_square(root=True)  # (False, None)
```

```
sage: R.<x> = QQ[]
sage: a = 2*(x+1)^2 / (2*(x-1)^2); a  # (x^2 + 2*x + 1)/(x^2 - 2*x + 1)
sage: a.is_square()  # True
sage: (0/x).is_square()  # True
```

**is_zero()**

Return True if this element is equal to zero.

EXAMPLES:

```python
sage: F = ZZ['x,y'].fraction_field()
sage: x,y = F.gens()
sage: t = F(0)/x
sage: t.is_zero()  # True
sage: u = 1/x - 1/x
sage: u.is_zero()  # True
sage: u.parent() is F  # True
```

**nth_root(n)**

Return a n-th root of this element.

EXAMPLES:

```python
sage: R = QQ['t'].fraction_field()
sage: t = R.gen()
```
sage: p = (t+1)^3 / (t^2+t-1)^3
sage: p.nth_root(3)
(t + 1)/(t^2 + t - 1)

sage: p = (t+1) / (t-1)
sage: p.nth_root(2)
Traceback (most recent call last):
  ... ValueError: not a 2nd power

numerator()
Return the numerator of self.

EXAMPLES:

sage: R.<x,y> = ZZ[]
sage: f = x/y+1; f
(x + y)/y
sage: f.numerator()
x + y

reduce()
Reduce this fraction.
Divides out the gcd of the numerator and denominator. If the denominator becomes a unit, it becomes 1. Additionally, depending on the base ring, the leading coefficients of the numerator and the denominator may be normalized to 1.

Automatically called for exact rings, but because it may be numerically unstable for inexact rings it must be called manually in that case.

EXAMPLES:

sage: R.<x> = RealField(10)[]
sage: f = (x^2+2*x+1)/(x+1); f
(x^2 + 2.0*x + 1.0)/(x + 1.0)
sage: f.reduce(); f
x + 1.0

specialization(D=None, phi=None)
Returns the specialization of a fraction element of a polynomial ring

valuation(v=None)
Return the valuation of self, assuming that the numerator and denominator have valuation functions defined on them.

EXAMPLES:

sage: x = PolynomialRing(RationalField(),'x').gen()
sage: f = (x^3 + x)/(x^2 - 2*x^3)
sage: f
(-1/2*x^2 - 1/2)/(x^2 - 1/2*x)
sage: f.valuation()
-1
sage: f.valuation(x^2+1)
1
class sage.rings.fraction_field_element.FractionFieldElement_ipoly_field
    Bases: sage.rings.fraction_field_element.FractionFieldElement

    A fraction field element where the parent is the fraction field of a univariate polynomial ring over a field.

    Many of the functions here are included for coherence with number fields.

    is_integral()
    Returns whether this element is actually a polynomial.

    EXAMPLES:

    sage: R.<t> = QQ[]
    sage: elt = (t^2 + t - 2) / (t + 2); elt
    # == (t + 2)*(t - 1)/(t + 2)
    t - 1
    sage: elt.is_integral()
    True
    sage: elt = (t^2 - t) / (t+2); elt
    # == t*(t - 1)/(t + 2)
    (t^2 - t)/(t + 2)
    sage: elt.is_integral()
    False

    reduce()
    Pick a normalized representation of self.

    In particular, for any a == b, after normalization they will have the same numerator and denominator.

    EXAMPLES:

    For univariate rational functions over a field, we have:

    sage: R.<x> = QQ[]
    sage: (2 + 2*x) / (4*x)  # indirect doctest
    (1/2*x + 1/2)/x
    Compare with:

    sage: R.<x> = ZZ[]
    sage: (2 + 2*x) / (4*x)
    (x + 1)/(2*x)

    support()
    Returns a sorted list of primes dividing either the numerator or denominator of this element.

    EXAMPLES:

    sage: R.<t> = QQ[]
    sage: h = (t^14 + 2*t^12 - 4*t^11 - 8*t^9 + 6*t^8 + 12*t^6 - 4*t^5 - 8*t^3 + t^→2 + 2)/(t^6 + 6*t^5 + 9*t^4 - 2*t^2 - 12*t - 18)
    sage: h.support()
    [t - 1, t + 3, t^2 + 2, t^2 + t + 1, t^4 - 2]

sage.rings.fraction_field_element.is_FractionFieldElement(x)
    Return whether or not x is a FractionFieldElement.

    EXAMPLES:
sage: from sage.rings.fraction_field_element import is_FractionFieldElement
sage: R.<x> = ZZ[]
sage: is_FractionFieldElement(x/2)
False
sage: is_FractionFieldElement(2/x)
True
sage: is_FractionFieldElement(1/3)
False

sage.rings.fraction_field_element.make_element(parent, numerator, denominator)
Used for unpickling FractionFieldElement objects (and subclasses).

EXAMPLES:

```python
sage: from sage.rings.fraction_field_element import make_element
sage: R = ZZ['x,y']
sage: x,y = R.gens()
sage: F = R.fraction_field()
sage: make_element(F, 1+x, 1+y)
(x + 1)/(y + 1)
```

sage.rings.fraction_field_element.make_element_old(parent, cdict)
Used for unpickling old FractionFieldElement pickles.

EXAMPLES:

```python
sage: from sage.rings.fraction_field_element import make_element_old
sage: R.<x,y> = ZZ[
```

4.3 Univariate rational functions over prime fields

class sage.rings.fraction_field_FpT.FpT(R, names=None)
Bases: sage.rings.fraction_field.FractionField_1poly_field

This class represents the fraction field GF(p)(T) for $2 < p < \sqrt{2^{11}-1}$.

EXAMPLES:

```python
sage: R.<T> = GF(71)[
```

sage: from sage.rings.fraction_field_FpT import *
sage: R.<t> = FpT(GF(5)['t'])
sage: list(R.iter(2))[350:355]
[(t^2 + t + 1)/(t + 2),
 (t^2 + t + 2)/(t + 2),
 (t^2 + t + 4)/(t + 2),
 (t^2 + 2*t + 1)/(t + 2),
 (t^2 + 2*t + 2)/(t + 2)]

class sage.rings.fraction_field_FpT.FpTElement
    Bases: sage.structure.element.FieldElement

An element of an FpT fraction field.

    denom()
    Return the denominator of this element, as an element of the polynomial ring.

    EXAMPLES:

    sage: K = GF(11)['t'].fraction_field()
    sage: t = K.gen(0); a = (t + 1/t)^3 - 1
    sage: a.denom()
t^3

denominator()
    Return the denominator of this element, as an element of the polynomial ring.

    EXAMPLES:

    sage: K = GF(11)['t'].fraction_field()
    sage: t = K.gen(0); a = (t + 1/t)^3 - 1
    sage: a.denominator()
t^3

factor()
    EXAMPLES:

    sage: K = Frac(GF(5)['t'])
    sage: t = K.gen()
    sage: f = 2 * (t+1) * (t^2+t+1)^2 / (t-1)
    sage: factor(f)
    (2) * (t + 4)^-1 * (t + 1) * (t^2 + t + 1)^2

is_square()
    Return True if this element is the square of another element of the fraction field.

    EXAMPLES:

    sage: K = GF(13)['t'].fraction_field(); t = K.gen()
    sage: t.is_square()
    False
    sage: (1/t^2).is_square()
    True
    sage: K(0).is_square()
    True
next()

This function iterates through all polynomials, returning the “next” polynomial after this one.

The strategy is as follows:

- We always leave the denominator monic.
- We progress through the elements with both numerator and denominator monic, and with the denominator less than the numerator. For each such, we output all the scalar multiples of it, then all of the scalar multiples of its inverse.
- So if the leading coefficient of the numerator is less than p-1, we scale the numerator to increase it by 1.
- Otherwise, we consider the multiple with numerator and denominator monic.
  - If the numerator is less than the denominator (lexicographically), we return the inverse of that element.
  - If the numerator is greater than the denominator, we invert, and then increase the numerator (remaining monic) until we either get something relatively prime to the new denominator, or we reach the new denominator. In this case, we increase the denominator and set the numerator to 1.

EXAMPLES:

```
sage: from sage.rings.fraction_field_FpT import *
sage: R.<t> = FpT(GF(3)['t'])
sage: a = R(0)
sage: for _ in range(30):
    ....:     a = a.next()
    ....:     print(a)
1
2
1/t
2/t
t
2*t
1/(t + 1)
2/(t + 1)
t + 1
2*t + 2
t/(t + 1)
2*t/(t + 1)
(t + 1)/t
(2*t + 2)/t
1/(t + 2)
2/(t + 2)
t + 2
2^t + 1
t/(t + 2)
2^t/(t + 2)
(t + 2)/t
(2^t + 1)/t
(t + 1)/(t + 2)
(2^t + 2)/(t + 2)
(t + 2)/(t + 1)
(2^t + 1)/(t + 1)
```
1/t^2
2/t^2
t^2
2*t^2

numerator()
Return the numerator of this element, as an element of the polynomial ring.

EXAMPLES:

```
sage: K = GF(11)['t'].fraction_field()
sage: t = K.gen(0); a = (t + 1/t)^3 - 1
sage: a.numerator()
t^6 + 3*t^4 + 10*t^3 + 3*t^2 + 1
```

numerator()
Return the numerator of this element, as an element of the polynomial ring.

EXAMPLES:

```
sage: K = GF(11)['t'].fraction_field()
sage: t = K.gen(0); a = (t + 1/t)^3 - 1
sage: a.numerator()
t^6 + 3*t^4 + 10*t^3 + 3*t^2 + 1
```

denominator()
Return the denominator of this element, as an element of the polynomial ring.

EXAMPLES:

```
sage: K = GF(11)['t'].fraction_field()
sage: t = K.gen(0); a = (t + 1/t)^3 - 1
sage: a.denominator()
t^6 + 3*t^4 + 10*t^3 + 3*t^2 + 1
```

sqrt(extend=True, all=False)
Return the square root of this element.

INPUT:

- **extend** - bool (default: True); if True, return a square root in an extension ring, if necessary. Otherwise, raise a ValueError if the square is not in the base ring.
- **all** - bool (default: False); if True, return all square roots of self, instead of just one.

EXAMPLES:

```
sage: from sage.rings.fraction_field_FpT import *
sage: K = GF(7)['t'].fraction_field(); t = K.gen(0)
sage: p = (t + 2)^2/(3*t^3 + 1)^4
sage: p.sqrt()
(3*t + 6)/(t^6 + 3*t^4 + 3*t^2 + 1)
sage: p.sqrt()^2 == p
True
```

subs(*args, **kwds)
EXAMPLES:

```
sage: K = Frac(GF(11)['t'])
sage: t = K.gen()
sage: f = (t+1)/(t-1)
sage: f.subs(t=2)
3
sage: f.subs(X=2)
(t + 1)/(t + 10)
```
valuation($v$)
Return the valuation of self at $v$.

EXAMPLES:

```python
sage: R.<t> = GF(5)[]
sage: f = (t+1)^2 * (t^2+t+1) / (t-1)^3
sage: f.valuation(t+1)
2
sage: f.valuation(t-1)
-3
sage: f.valuation(t)
0
```

class sage.rings.fraction_field_FpT.FpT_Fp_section
Bases: sage.categories.map.Section
This class represents the section from GF(p)(t) back to GF(p)[t]

EXAMPLES:

```python
sage: R.<t> = GF(5)[]
sage: K = R.fraction_field()
sage: f = GF(5).convert_map_from(K); f
Section map:
  From: Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 5
  To:  Finite Field of size 5
sage: type(f)
<type 'sage.rings.fraction_field_FpT.FpT_Fp_section'>
```

Warning: Comparison of FpT_Fp_section objects is not currently implemented. See trac ticket #23469.

```python
sage: fprime = loads(dumps(f))
sage: fprime == f
False
sage: fprime(3) == f(3)
True
```

class sage.rings.fraction_field_FpT.FpT_Polyring_section
Bases: sage.categories.map.Section
This class represents the section from GF(p)(t) back to GF(p)[t]

EXAMPLES:

```python
sage: R.<t> = GF(5)[]
sage: K = R.fraction_field()
sage: f = R.convert_map_from(K); f
Section map:
  From: Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 5
  To:  Univariate Polynomial Ring in t over Finite Field of size 5
```

(continues on next page)
Warning: Comparison of FpT_Polyring_section objects is not currently implemented. See trac ticket #23469.

```python
sage: fprime = loads(dumps(f))
sage: fprime == f
False
sage: fprime(1+t) == f(1+t)
True
```

class sage.rings.fraction_field_FpT.FpT_iter

Bases: object

Return a class that iterates over all elements of an FpT.

EXAMPLES:

```python
sage: K = GF(3)[t].fraction_field()
sage: I = K.iter(1)
sage: list(I)
[0, 1, 2, t, t + 1, t + 2, 2*t, 2*t + 1, 2*t + 2, 1/t, 2/t, (t + 1)/t, (t + 2)/t, (2*t + 1)/t, (2*t + 2)/t, 1/(t + 1), 2/(t + 1), t/(t + 1), (t + 2)/(t + 1), 2*t/(t + 1), (2*t + 1)/(t + 1), 1/(t + 2), 2/(t + 2), t/(t + 2), (t + 1)/(t + 2), 2*t/(t + 2), (2*t + 1)/(t + 2)]
```

class sage.rings.fraction_field_FpT.Fp_FpT_coerce
Bases: sage.rings.morphism.RingHomomorphism

This class represents the coercion map from GF(p) to GF(p)(t)

EXAMPLES:

```sage
sage: R.<t> = GF(5)[]
sage: K = R.fraction_field()
sage: f = K.coerce_map_from(GF(5)); f
Ring morphism:
  From: Finite Field of size 5
  To: Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 5
sage: type(f)
<type 'sage.rings.fraction_field_FpT.Fp_FpT_coerce'>
```

section()

Return the section of this inclusion: the partially defined map from GF(p)(t) back to GF(p), defined on constant elements.

EXAMPLES:

```sage
sage: R.<t> = GF(5)[]
sage: K = R.fraction_field()
sage: f = K.coerce_map_from(R); f
Ring morphism:
  From: Univariate Polynomial Ring in t over Finite Field of size 5
  To: Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 5
sage: type(f)
<type 'sage.rings.fraction_field_FpT.Polyring_FpT_coerce'>
```

4.3. Univariate rational functions over prime fields 513
sage: type(f)
<type 'sage.rings.fraction_field_FpT.Polyring_FpT_coerce'>

section()
Return the section of this inclusion: the partially defined map from GF(p)(t) back to GF(p)[t], defined on elements with unit denominator.

EXAMPLES:

sage: R.<t> = GF(5)[]
sage: K = R.fraction_field()
sage: f = K.coerce_map_from(R)
sage: g = f.section(); g
Section map:
   From: Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 5
   To:   Univariate Polynomial Ring in t over Finite Field of size 5
sage: t = K.gen()
sage: g(t)
t
sage: g(1/t)
Traceback (most recent call last):
  ... ValueError: not integral

class sage.rings.fraction_field_FpT.ZZ_FpT_coerce
Bases: sage.rings.morphism.RingHomomorphism

This class represents the coercion map from ZZ to GF(p)(t)

EXAMPLES:

sage: R.<t> = GF(17)[]
sage: K = R.fraction_field()
sage: f = K.coerce_map_from(ZZ); f
Ring morphism:
   From: Integer Ring
   To:   Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 17
sage: type(f)
<type 'sage.rings.fraction_field_FpT.ZZ_FpT_coerce'>

section()
Return the section of this inclusion: the partially defined map from GF(p)(t) back to ZZ, defined on constant elements.

EXAMPLES:

sage: R.<t> = GF(5)[]
sage: K = R.fraction_field()
sage: f = K.coerce_map_from(ZZ)
sage: g = f.section(); g
Composite map:
   From: Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 5
   To:   Integer Ring
sage: type(g)
<type 'sage.rings.fraction_field_FpT.ZZ_FpT_coerce'>

(continues on next page)
To: Integer Ring
Defn: Section map:
   From: Fraction Field of Univariate Polynomial Ring in t over Finite
          Field of size 5
          then
   Lifting map:
      From: Finite Field of size 5
      To: Integer Ring

\begin{Verbatim}
sage: t = K.gen()
sage: g(f(1,3,reduce=False))
2
sage: g(t)
Traceback (most recent call last):
  ... ValueError: not constant
sage: g(1/t)
Traceback (most recent call last):
  ... ValueError: not integral
\end{Verbatim}

\texttt{sage.rings.fraction_field_FpT.unpickle_FpT_element}(\texttt{K}, \texttt{numer}, \texttt{denom})

Used for pickling.
CHAPTER FIVE

LAURENT POLYNOMIALS

5.1 Ring of Laurent Polynomials

If $R$ is a commutative ring, then the ring of Laurent polynomials in $n$ variables over $R$ is $R[x_1^{\pm 1}, x_2^{\pm 1}, \ldots, x_n^{\pm 1}]$. We implement it as a quotient ring

$$R[x_1, y_1, x_2, y_2, \ldots, x_n, y_n]/(x_1 y_1 - 1, x_2 y_2 - 1, \ldots, x_n y_n - 1).$$

AUTHORS:

- David Roe (2008-2-23): created
- David Loeffler (2009-07-10): cleaned up docstrings

`sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing(base_ring, *args, **kwds)`

Return the globally unique univariate or multivariate Laurent polynomial ring with given properties and variable name or names.

There are four ways to call the Laurent polynomial ring constructor:

1. `LaurentPolynomialRing(base_ring, name, sparse=False)`
2. `LaurentPolynomialRing(base_ring, names, order='degrevlex')`
3. `LaurentPolynomialRing(base_ring, name, n, order='degrevlex')`
4. `LaurentPolynomialRing(base_ring, n, name, order='degrevlex')`

The optional arguments sparse and order must be explicitly named, and the other arguments must be given positionally.

INPUT:

- `base_ring` – a commutative ring
- `name` – a string
- `names` – a list or tuple of names, or a comma separated string
- `n` – a positive integer
- `sparse` – bool (default: False), whether or not elements are sparse
- `order` – string or `TermOrder`, e.g.,
  - `'degrevlex'` (default) – degree reverse lexicographic
  - `'lex'` – lexicographic
  - `'deglex'` – degree lexicographic
- TermOrder('deglex',3) + TermOrder('deglex',3) – block ordering

OUTPUT:

LaurentPolynomialRing(base_ring, name, sparse=False) returns a univariate Laurent polynomial ring; all other input formats return a multivariate Laurent polynomial ring.

UNIQUENESS and IMMUTABILITY: In Sage there is exactly one single-variate Laurent polynomial ring over each base ring in each choice of variable and sparseness. There is also exactly one multivariate Laurent polynomial ring over each base ring for each choice of names of variables and term order.

```
sage: R.<x,y> = LaurentPolynomialRing(QQ,2); R
Multivariate Laurent Polynomial Ring in x, y over Rational Field

sage: f = x^2 - 2*y^-2
```

You can’t just globally change the names of those variables. This is because objects all over Sage could have pointers to that polynomial ring.

```
sage: R._assign_names(['z','w'])
Traceback (most recent call last):
  ...
ValueError: variable names cannot be changed after object creation.
```

EXAMPLES:

1. LaurentPolynomialRing(base_ring, name, sparse=False)

```
sage: LaurentPolynomialRing(QQ, 'w')
Univariate Laurent Polynomial Ring in w over Rational Field
```

Use the diamond brackets notation to make the variable ready for use after you define the ring:

```
sage: R.<w> = LaurentPolynomialRing(QQ)
sage: (1 + w)^3
1 + 3*w + 3*w^2 + w^3
```

You must specify a name:

```
sage: LaurentPolynomialRing(QQ)
Traceback (most recent call last):
  ...
TypeError: you must specify the names of the variables
```

```
sage: R.<abc> = LaurentPolynomialRing(QQ, sparse=True); R
Univariate Laurent Polynomial Ring in abc over Rational Field

sage: R.<w> = LaurentPolynomialRing(PolynomialRing(GF(7),'k')); R
Univariate Laurent Polynomial Ring in w over Univariate Polynomial Ring in k over Finite Field of size 7
```

Rings with different variables are different:

```
sage: LaurentPolynomialRing(QQ, 'x') == LaurentPolynomialRing(QQ, 'y')
False
```

2. LaurentPolynomialRing(base_ring, names, order='degrevlex')

```python
sage: R = LaurentPolynomialRing(QQ, 'a,b,c'); R
Multivariate Laurent Polynomial Ring in a, b, c over Rational Field

sage: S = LaurentPolynomialRing(QQ, ['a','b','c']); S
Multivariate Laurent Polynomial Ring in a, b, c over Rational Field

sage: T = LaurentPolynomialRing(QQ, ('a','b','c')); T
Multivariate Laurent Polynomial Ring in a, b, c over Rational Field

All three rings are identical.

sage: (R is S) and (S is T)
True

There is a unique Laurent polynomial ring with each term order:

```python
sage: R = LaurentPolynomialRing(QQ, 'x,y,z', order='degrevlex'); R
Multivariate Laurent Polynomial Ring in x, y, z over Rational Field

sage: S = LaurentPolynomialRing(QQ, 'x,y,z', order='invlex'); S
Multivariate Laurent Polynomial Ring in x, y, z over Rational Field

sage: S is LaurentPolynomialRing(QQ, 'x,y,z', order='invlex')
True

sage: R == S
False

3. LaurentPolynomialRing(base_ring, name, n, order='degrevlex')

If you specify a single name as a string and a number of variables, then variables labeled with numbers are created.

```python
sage: LaurentPolynomialRing(QQ, 'x', 10)
Multivariate Laurent Polynomial Ring in x0, x1, x2, x3, x4, x5, x6, x7, x8, x9 over Rational Field

sage: LaurentPolynomialRing(GF(7), 'y', 5)
Multivariate Laurent Polynomial Ring in y0, y1, y2, y3, y4 over Finite Field of size 7

sage: LaurentPolynomialRing(QQ, 'y', 3, sparse=True)
Multivariate Laurent Polynomial Ring in y0, y1, y2 over Rational Field

By calling the inject_variables() method, all those variable names are available for interactive use:

```python
sage: R = LaurentPolynomialRing(GF(7),15,'w'); R
Multivariate Laurent Polynomial Ring in w0, w1, w2, w3, w4, w5, w6, w7, w8, w9, w10, w11, w12, w13, w14 over Finite Field of size 7

sage: R.inject_variables()
Defining w0, w1, w2, w3, w4, w5, w6, w7, w8, w9, w10, w11, w12, w13, w14

sage: (w0 + 2*w8 + w13)^2
w0^2 + 4*w0*w8 + 4*w8^2 + 2*w0*w13 + 4*w8*w13 + w13^2
```

5.1. Ring of Laurent Polynomials

```python
class sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_generic(R)
Bases: sage.rings.ring.CommutativeRing, sage.structure.parent.Parent

Laurent polynomial ring (base class).```
EXAMPLES:

This base class inherits from `CommutativeRing`. Since trac ticket #11900, it is also initialised as such:

```
sage: R.<x1,x2> = LaurentPolynomialRing(QQ)
sage: R.category()
Join of Category of unique factorization domains and Category of commutative
  → algebras over (number fields and quotient fields and metric spaces) and Category
  → of infinite sets
sage: TestSuite(R).run()
```

`change_ring(base_ring=None, names=None, sparse=False, order=None)`

EXAMPLES:

```
sage: R = LaurentPolynomialRing(QQ,2,'x')
sage: R.change_ring(ZZ)
Multivariate Laurent Polynomial Ring in x0, x1 over Integer Ring
```

Check that the distinction between a univariate ring and a multivariate ring with one generator is preserved:

```
sage: P.<x> = LaurentPolynomialRing(QQ, 1)
sage: P
Multivariate Laurent Polynomial Ring in x over Rational Field
sage: K.<i> = CyclotomicField(4)
sage: P.change_ring(K)
Multivariate Laurent Polynomial Ring in x over Cyclotomic Field of order 4 and
  → degree 2
```

`characteristic()`

Returns the characteristic of the base ring.

EXAMPLES:

```
sage: LaurentPolynomialRing(QQ,2,'x').characteristic()
0
sage: LaurentPolynomialRing(GF(3),2,'x').characteristic()
3
```

`completion(p, prec=20, extras=None)`

EXAMPLES:

```
sage: P.<x>=LaurentPolynomialRing(QQ)
sage: P
Univariate Laurent Polynomial Ring in x over Rational Field
sage: PP=P.completion(x)
sage: PP
Laurent Series Ring in x over Rational Field
sage: f=1-1/x
sage: PP(f)
-x^-1 + 1
sage: 1/PP(f)
-x - x^2 - x^3 - x^4 - x^5 - x^6 - x^7 - x^8 - x^9 - x^10 - x^11 - x^12 - x^13 -
  → x^14 - x^15 - x^16 - x^17 - x^18 - x^19 - x^20 + O(x^21)
```

`construction()`

Return the construction of `self`. 

---

**Chapter 5. Laurent Polynomials**
EXAMPLES:

```python
sage: LaurentPolynomialRing(QQ,2,'x,y').construction()
(LaurentPolynomialFunctor,
 Univariate Laurent Polynomial Ring in x over Rational Field)
```

**fraction_field()**

The fraction field is the same as the fraction field of the polynomial ring.

EXAMPLES:

```python
sage: L.<x> = LaurentPolynomialRing(QQ)
sage: L.fraction_field()
Fraction Field of Univariate Polynomial Ring in x over Rational Field
sage: (x^2 - 1 + 2) / (x - 1)
(2*x + 1)/(x^2 - x)
```

**gen(i=0)**

Returns the \(i^{th}\) generator of self. If \(i\) is not specified, then the first generator will be returned.

EXAMPLES:

```python
sage: LaurentPolynomialRing(QQ,2,'x').gen()
x0
sage: LaurentPolynomialRing(QQ,2,'x').gen(0)
x0
sage: LaurentPolynomialRing(QQ,2,'x').gen(1)
x1
```

**ideal(*args, **kwds)**

EXAMPLES:

```python
sage: LaurentPolynomialRing(QQ,2,'x').ideal([1])
Ideal (1) of Multivariate Laurent Polynomial Ring in x0, x1 over Rational Field
```

**is_exact()**

Returns True if the base ring is exact.

EXAMPLES:

```python
sage: LaurentPolynomialRing(QQ,2,'x').is_exact()
True
sage: LaurentPolynomialRing(RDF,2,'x').is_exact()
False
```

**is_field(proof=True)**

EXAMPLES:

```python
sage: LaurentPolynomialRing(QQ,2,'x').is_field()
False
```

**is_finite()**

EXAMPLES:

```python
sage: LaurentPolynomialRing(QQ,2,'x').is_finite()
False
```
is_integral_domain(proof=True)
Returns True if self is an integral domain.

EXAMPLES:

```
sage: LaurentPolynomialRing(QQ,2,'x').is_integral_domain()
True
```

The following used to fail; see trac ticket #7530:

```
sage: L = LaurentPolynomialRing(ZZ, 'X')
sage: L['Y']
Univariate Polynomial Ring in Y over Univariate Laurent Polynomial Ring in X
˓→over Integer Ring
```

is_noetherian()
Returns True if self is Noetherian.

EXAMPLES:

```
sage: LaurentPolynomialRing(QQ,2,'x').is_noetherian()
Traceback (most recent call last):
... Not Implemented
```

krull_dimension()
EXAMPLES:

```
sage: LaurentPolynomialRing(QQ,2,'x').krull_dimension()
Traceback (most recent call last):
... Not Implemented
```

ngens()
Return the number of generators of self.

EXAMPLES:

```
sage: LaurentPolynomialRing(QQ,2,'x').ngens()
2
sage: LaurentPolynomialRing(QQ,1,'x').ngens()
1
```

polynomial_ring()
Returns the polynomial ring associated with self.

EXAMPLES:

```
sage: LaurentPolynomialRing(QQ,2,'x').polynomial_ring()
Multivariate Polynomial Ring in x0, x1 over Rational Field
sage: LaurentPolynomialRing(QQ,1,'x').polynomial_ring()
Multivariate Polynomial Ring in x over Rational Field
```

random_element(low_degree=-2, high_degree=2, terms=5, choose_degree=False, *args, **kwds)
EXAMPLES:
```python
sage: LaurentPolynomialRing(QQ,2,'x').random_element()
Traceback (most recent call last):
  ... 
NotImplementedError
```

**remove_var**(var)

EXAMPLES:

```python
sage: R = LaurentPolynomialRing(QQ,'x,y,z')
sage: R.remove_var('x')
Multivariate Laurent Polynomial Ring in y, z over Rational Field
sage: R.remove_var('x').remove_var('y')
Univariate Laurent Polynomial Ring in z over Rational Field
```

**term_order()**

Returns the term order of self.

EXAMPLES:

```python
sage: LaurentPolynomialRing(QQ,2,'x').term_order()
Degree reverse lexicographic term order
```

**variable_names_recursive**(depth=+ Infinity)

Return the list of variable names of this ring and its base rings, as if it were a single multi-variate Laurent polynomial.

INPUT:

- depth – an integer or Infinity.

OUTPUT:

A tuple of strings.

EXAMPLES:

```python
sage: T = LaurentPolynomialRing(QQ, 'x')
sage: S = LaurentPolynomialRing(T, 'y')
sage: R = LaurentPolynomialRing(S, 'z')
sage: R.variable_names_recursive()
('x', 'y', 'z')
sage: R.variable_names_recursive(2)
('y', 'z')
```

**class** sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_mpair(R)

Bases: sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_generic

EXAMPLES:

```python
sage: L = LaurentPolynomialRing(QQ,2,'x')
sage: type(L)
<class 'sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_mpair_with_category'>
sage: L == loads(dumps(L))
True
```
Element
alias of sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair

monomial(*args)
Return the monomial whose exponents are given in argument.

EXAMPLES:

```
sage: L = LaurentPolynomialRing(QQ, 'x', 2)
sage: L.monomial(-3, 5)
x0^-3*x1^5
sage: L.monomial(1, 1)
x0^*x1
sage: L.monomial(0, 0)
1
sage: L.monomial(-2, -3)
x0^-2*x1^-3
sage: x0, x1 = L.gens()
sage: L.monomial(-1, 2) == x0^-1 * x1^2
True
sage: L.monomial(1, 2, 3)
Traceback (most recent call last):
  ...TypeError: tuple key must have same length as ngens
```

class sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_univariate(R)
Bases: sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_generic

EXAMPLES:

```
sage: L = LaurentPolynomialRing(QQ, 'x')
sage: type(L)
<class 'sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_univariate_with_category'>
sage: L == loads(dumps(L))
True
```

Element
alias of sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate
sage.rings.polynomial.laurent_polynomial_ring.is_LaurentPolynomialRing(R)
Returns True if and only if R is a Laurent polynomial ring.

EXAMPLES:

```
sage: from sage.rings.polynomial.laurent_polynomial_ring import is_LaurentPolynomialRing
sage: P = PolynomialRing(QQ,2,'x')
sage: is_LaurentPolynomialRing(P)
False
sage: R = LaurentPolynomialRing(QQ,3,'x')
sage: is_LaurentPolynomialRing(R)
True
```
5.2 Elements of Laurent polynomial rings

class sage.rings.polynomial.laurent_polynomial.LaurentPolynomial
   Bases: sage.structure.element.CommutativeAlgebraElement

Base class for Laurent polynomials.

change_ring(R)
   Return a copy of this Laurent polynomial, with coefficients in R.

   EXAMPLES:

   sage: R.<x> = LaurentPolynomialRing(QQ)
sage: a = x^2 + 3*x^3 + 5*x^-1
sage: a.change_ring(GF(3))
2*x^-1 + x^2

   Check that trac ticket #22277 is fixed:

   sage: R.<x, y> = LaurentPolynomialRing(QQ)
sage: a = 2*x^2 + 3*x^3 + 4*x^-1
sage: a.change_ring(GF(3))
-x^2 + x^-1

dict()
   Abstract dict method.

   EXAMPLES:

   sage: R.<x> = LaurentPolynomialRing(ZZ)
sage: from sage.rings.polynomial.laurent_polynomial import LaurentPolynomial
sage: LaurentPolynomial.dict(x)
Traceback (most recent call last):
  ...
NotImplementedError

hamming_weight()
   Return the hamming weight of self.

   The hamming weight is number of non-zero coefficients and also known as the weight or sparsity.

   EXAMPLES:

   sage: R.<x> = LaurentPolynomialRing(ZZ)
sage: f = x^3 - 1
sage: f.hamming_weight()
2

map_coefficients(f, new_base_ring=None)
   Apply f to the coefficients of self.

   If f is a sage.categories.map.Map, then the resulting polynomial will be defined over the codomain of f. Otherwise, the resulting polynomial will be over the same ring as self. Set new_base_ring to override this behavior.

   INPUT:

   * f – a callable that will be applied to the coefficients of self.
• new_base_ring (optional) – if given, the resulting polynomial will be defined over this ring.

EXAMPLES:

```
sage: k.<a> = GF(9)
sage: R.<x> = LaurentPolynomialRing(k)
sage: f = x*a + a
sage: f.map_coefficients(lambda a : a + 1)
(a + 1) + (a + 1)*x
sage: R.<x,y> = LaurentPolynomialRing(k, 2)
sage: f = x*a + 2*x^3*y*a + a
sage: f.map_coefficients(lambda a : a + 1)
(2*a + 1)*x^3*y + (a + 1)*x + a + 1
```

Examples with different base ring:

```
sage: R.<r> = GF(9); S.<s> = GF(81)
sage: h = Hom(R,S)[0]; h
Ring morphism:
  From: Finite Field in r of size 3^2
  To:   Finite Field in s of size 3^4
  Defn: r |--> 2*s^3 + 2*s^2 + 1
sage: T.<X,Y> = LaurentPolynomialRing(R, 2)
sage: f = r*X+Y
sage: g = f.map_coefficients(h); g
(2*s^3 + 2*s^2 + 1)*X + Y
sage: g.parent()
Multivariate Laurent Polynomial Ring in X, Y over Finite Field in s of size 3^4
sage: h = lambda x: x.trace()
sage: g = f.map_coefficients(h); g
X - Y
sage: g.parent()
Multivariate Laurent Polynomial Ring in X, Y over Finite Field of size 3
```

**number_of_terms()**

Abstract method for number of terms

EXAMPLES:

```
sage: R.<x> = LaurentPolynomialRing(ZZ)
sage: from sage.rings.polynomial.laurent_polynomial import LaurentPolynomial
sage: LaurentPolynomial.number_of_terms(x)
Traceback (most recent call last):
  ...
NotImplementedError
```

**class** `sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair`

Bases: `sage.rings.polynomial.laurent_polynomial.LaurentPolynomial`

Multivariate Laurent polynomials.

**coefficient** *(mon)*

Return the coefficient of mon in self, where mon must have the same parent as self.
The coefficient is defined as follows. If \( f \) is this polynomial, then the coefficient \( c_m \) is sum:

\[
c_m := \sum_T \frac{T}{m}
\]

where the sum is over terms \( T \) in \( f \) that are exactly divisible by \( m \).

A monomial \( m(x,y) \) ‘exactly divides’ \( f(x,y) \) if \( m(x,y)|f(x,y) \) and neither \( x \cdot m(x,y) \) nor \( y \cdot m(x,y) \) divides \( f(x,y) \).

INPUT:

- \( \text{mon} \) – a monomial

OUTPUT:

Element of the parent of \( \text{self} \).

Note: To get the constant coefficient, call \text{constant_coefficient}().

EXAMPLES:

```python
sage: P.<x,y> = LaurentPolynomialRing(QQ)
```

The coefficient returned is an element of the parent of \( \text{self} \); in this case, \( P \).

```python
sage: f = 2 * x * y
sage: c = f.coefficient(x*y); c
2
sage: c.parent()
Multivariate Laurent Polynomial Ring in x, y over Rational Field
```

```python
sage: P.<x,y> = LaurentPolynomialRing(QQ)
sage: f = (y^2 - x^9 - 7*x*y^2 + 5*x*y)*x^-3; f
-x^6 - 7*x^-2*y^2 + 5*x^-2*y + x^-3*y^2
sage: f.coefficient(y)
5*x^-2
sage: f.coefficient(y^2)
-7*x^-2 + x^-3
sage: f.coefficient(x*y)
0
sage: f.coefficient(x^-2)
-7*y^2 + 5*y
sage: f.coefficient(x^-2*y^2)
-7
```

```python
coefficients()
```

Return the nonzero coefficients of \( \text{self} \) in a list.

The returned list is decreasingly ordered by the term ordering of \( \text{self}.parent() \).

EXAMPLES:
sage: L.<x,y,z> = LaurentPolynomialRing(QQ, order='degrevlex')
sage: f = 4*x^7*z^-1 + 3*x^3*y + 2*x^4*z^-2 + x^6*y^-7
sage: f.coefficients()
[4, 3, 2, 1]
sage: L.<x,y,z> = LaurentPolynomialRing(QQ, order='lex')
sage: f = 4*x^7*z^-1 + 3*x^3*y + 2*x^4*z^-2 + x^6*y^-7
sage: f.coefficients()
[4, 1, 2, 3]

constant_coefficient()
Return the constant coefficient of self.

EXAMPLES:

sage: P.<x,y> = LaurentPolynomialRing(QQ)
sage: f = (y^2 - x^9 - 7*x*y^2 + 5*x*y)*x^-3; f
-x^6 - 7*x^-2*y^2 + 5*x^-2*y + x^-3*y^2
sage: f.constant_coefficient()
0
sage: f = (x^3 + 2*x^-2*y+y^3)*y^-3; f
x^3*y^-3 + 1 + 2*x^-2*y^-2
sage: f.constant_coefficient()
1

degree(x=None)
Return the degree of x in self.

EXAMPLES:

sage: R.<x,y,z> = LaurentPolynomialRing(QQ)
sage: f = 4*x^7*z^-1 + 3*x^3*y + 2*x^4*z^-2 + x^6*y^-7
sage: f.degree(x)
7
sage: f.degree(y)
1
sage: f.degree(z)
0

derivative(*args)
The formal derivative of this Laurent polynomial, with respect to variables supplied in args.

Multiple variables and iteration counts may be supplied; see documentation for the global derivative() function for more details.

See also:
._derivative()

EXAMPLES:

sage: R = LaurentPolynomialRing(ZZ, 'x, y')
sage: x, y = R.gens()
sage: t = x**4*y+x*y+y**(-1)+y**(-3)
sage: t.derivative(x, x)
12*x^2*y + 2*x^-3
sage: t.derivative(y, 2)
12*y^-5
dict()  
Return self represented as a dict.

EXAMPLES:

```python
sage: L.<x,y,z> = LaurentPolynomialRing(QQ)
sage: f = 4*x^7*z^-1 + 3*x^3*y + 2*x^4*z^-2 + x^6*y^-7
sage: sorted(f.dict().items())
[((3, 1, 0), 3), ((4, 0, -2), 2), ((6, -7, 0), 1), ((7, 0, -1), 4)]
```

diff(*args)  
The formal derivative of this Laurent polynomial, with respect to variables supplied in args.

Multiple variables and iteration counts may be supplied; see documentation for the global derivative() function for more details.

See also:  
_derivative()

EXAMPLES:

```python
sage: R = LaurentPolynomialRing(ZZ, 'x, y')
sage: x, y = R.gens()
sage: t = x**4*y + x*y + y + x**(-1) + y**(-3)
sage: t.derivative(x, x)
12*x^2*y + 2*x^-3
sage: t.derivative(y, 2)
12*y^-5
```

differentiate(*args)  
The formal derivative of this Laurent polynomial, with respect to variables supplied in args.

Multiple variables and iteration counts may be supplied; see documentation for the global derivative() function for more details.

See also:  
_derivative()

EXAMPLES:

```python
sage: R = LaurentPolynomialRing(ZZ, 'x, y')
sage: x, y = R.gens()
sage: t = x**4*y + x*y + y + x**(-1) + y**(-3)
sage: t.derivative(x, x)
12*x^2*y + 2*x^-3
sage: t.derivative(y, 2)
12*y^-5
```

exponents()  
Return a list of the exponents of self.

EXAMPLES:

```python
sage: L.<w,z> = LaurentPolynomialRing(QQ)
sage: a = w^2*z^-1 + 3; a
w^2*z^-1 + 3
sage: e = a.exponents()
```
factor()  
Returns a Laurent monomial (the unit part of the factorization) and a factored multi-polynomial.  

EXAMPLES:

```
sage: L.<x,y,z> = LaurentPolynomialRing(QQ)
sage: f = 4*x^7*z^-1 + 3*x^3*y + 2*x^4*z^-2 + x^6*y^-7  
sage: f.factor()  
(x^3*y^-7*z^-2) * (4*x^4*y^7*z + 3*y^8*z^2 + 2*x*y^7 + x^3*z^2)
```

has_any_inverse()  
Returns True if self contains any monomials with a negative exponent, False otherwise.

EXAMPLES:

```
sage: L.<x,y,z> = LaurentPolynomialRing(QQ)
sage: f = 4*x^7*z^-1 + 3*x^3*y + 2*x^4*z^-2 + x^6*y^-7  
sage: f.has_any_inverse()  
True  
sage: g = x^2 + y^2  
sage: g.has_any_inverse()  
False
```

has_inverse_of(i)  
INPUT:  
- • i – The index of a generator of self.parent()  
OUTPUT:  
Returns True if self contains a monomial including the inverse of self.parent().gen(i), False otherwise.

EXAMPLES:

```
sage: L.<x,y,z> = LaurentPolynomialRing(QQ)
sage: f = 4*x^7*z^-1 + 3*x^3*y + 2*x^4*z^-2 + x^6*y^-7  
sage: f.has_inverse_of(0)  
False  
sage: f.has_inverse_of(1)  
True  
sage: f.has_inverse_of(2)  
True
```

is_constant()  
Return whether this Laurent polynomial is constant.

EXAMPLES:

```
sage: L.<a, b> = LaurentPolynomialRing(QQ)
sage: L(0).is_constant()  
True  
sage: L(42).is_constant()  
(continues on next page)
```
True
sage: a.is_constant()
False
sage: (1/b).is_constant()
False

**is_monomial()**

Return True if self is a monomial.

**EXAMPLES:**

sage: k.<y,z> = LaurentPolynomialRing(QQ)
sage: z.is_monomial()
True
sage: k(1).is_monomial()
True
sage: (z+1).is_monomial()
False
sage: (z^-2909).is_monomial()
True
sage: (38*z^-2909).is_monomial()
False

**is_square(root=False)**

Test whether this Laurent polynomial is a square.

**INPUT:**

* root - boolean (default False) - if set to True then return a pair (True, sqrt) with sqrt a square root of this Laurent polynomial when it exists or (False, None).

**EXAMPLES:**

sage: L.<x,y,z> = LaurentPolynomialRing(QQ)
sage: p = (1 + x*y + z^-3)
sage: (p**2).is_square()
True
sage: (p**2).is_square(root=True)
(True, x*y + 1 + z^-3)

sage: x.is_square()
False
sage: x.is_square(root=True)
(False, None)

sage: (x**-4 * (1 + z)).is_square(root=False)
False
sage: (x**-4 * (1 + z)).is_square(root=True)
(False, None)

**is_unit()**

Return True if self is a unit.

The ground ring is assumed to be an integral domain.

This means that the Laurent polynomial is a monomial with unit coefficient.
EXAMPLES:

```python
sage: R.<x,y,z> = LaurentPolynomialRing(QQ)
sage: f = (x^3 + y^-3)*z
sage: f.is_univariate()
False
sage: g = f(1,y,4)
sage: g.is_univariate()
True
sage: R(1).is_univariate()
True
```

`is_univariate()`

Return True if this is a univariate or constant Laurent polynomial, and False otherwise.

EXAMPLES:

```python
sage: R.<x,y,z> = LaurentPolynomialRing(QQ)
sage: f = (x^3 + y^-3)*z
sage: f.is_univariate()
False
sage: g = f(1,y,4)
sage: g.is_univariate()
True
sage: R(1).is_univariate()
True
```

`iterator_exp_coeff()`

Iterate over self as pairs of (ETuple, coefficient).

EXAMPLES:

```python
sage: P.<x,y> = LaurentPolynomialRing(QQ)
sage: f = (y^2 - x^9 - 7*x*y^3 + 5*x*y)*x^-3
sage: list(f.iterator_exp_coeff())
[[(6, 0), -1], ((-2, 3), -7), ((-2, 1), 5), ((-3, 2), 1)]
```

`monomial_coefficient(mon)`

Return the coefficient in the base ring of the monomial mon in self, where mon must have the same parent as self.

This function contrasts with the function `coefficient()` which returns the coefficient of a monomial viewing this polynomial in a polynomial ring over a base ring having fewer variables.

INPUT:

* mon – a monomial

See also:

For coefficients in a base ring of fewer variables, see `coefficient()`.

EXAMPLES:
monomials()
Return the list of monomials in self.

EXAMPLES:

```python
sage: P.<x,y> = Laurent PolynomialRing(QQ)
sage: f = (y^2 - x^9 - 7*x*y^3 + 5*x*y)*x^-3
sage: sorted(f.monomials())
[x^-3*y^2, x^-2*y, x^-2*y^3, x^6]
```

number_of_terms()
Return the number of non-zero coefficients of self.
Also called weight, hamming weight or sparsity.

EXAMPLES:

```python
sage: R.<x, y> = Laurent PolynomialRing(ZZ)
sage: f = x^3 - y
sage: f.number_of_terms()
2
sage: f = (x+1/y)^100
sage: f.number_of_terms()
101
```

The method hamming_weight() is an alias:

```python
sage: f.hamming_weight()
101
```

quo_rem(right)
Divide this Laurent polynomial by right and return a quotient and a remainder.

INPUT:

• right – a Laurent polynomial

OUTPUT:
A pair of Laurent polynomials.

EXAMPLES:

```python
sage: R.<s, t> = Laurent PolynomialRing(QQ)
sage: (s^2-t^2).quo_rem(s-t)
(s + t, 0)
sage: (s^2-t^2).quo_rem(s-t)
(s + t, -s^2 + s^-2)
```

(continues on next page)
sage: (s^-2-t^2).quo_rem(s^-1-t)
(t + s^-1, 0)

rescale_vars \(d, h=None, new\_ring=None\)
Rescale variables in a Laurent polynomial.

**INPUT:**
- \(d\) – a dict whose keys are the generator indices and values are the coefficients; so a pair \((i, v)\) means \(x_i \mapsto v x_i\)
- \(h\) – (optional) a map to be applied to coefficients done after rescaling
- \(new\_ring\) – (optional) a new ring to map the result into

**EXAMPLES:**

```python
sage: L.<x,y> = LaurentPolynomialRing(QQ, 2)
sage: p = x^-2*y + x*y^-2
sage: p.rescale_vars({0: 2, 1: 3})
2/9*x*y^-2 + 3/4*x^-2*y
sage: F = GF(2)
sage: p.rescale_vars({0: 3, 1: 7}, new_ring=L.change_ring(F))
x*y^-2 + x^-2*y
```

Test for trac ticket #30331:

```python
sage: F.<z> = CyclotomicField(3)
sage: p.rescale_vars({0: 2, 1: z}, new_ring=L.change_ring(F))
2*z*x*y^-2 + 1/4*z*x^-2*y
```

subs \((in\_dict=None, **kwds)\)
Substitute some variables in this Laurent polynomial.

Variable/value pairs for the substitution may be given as a dictionary or via keyword-value pairs. If both are present, the latter take precedence.

**INPUT:**
- \(in\_dict\) – dictionary (optional)
- \(**kwds\) – keyword arguments

**OUTPUT:**
A Laurent polynomial.

**EXAMPLES:**

```python
sage: L.<x, y, z> = LaurentPolynomialRing(QQ)
sage: f = x + 2*y + 3*z
sage: f.subs(x=1)
2*y + 3*z + 1
sage: f.subs(y=1)
x + 3*z + 2
sage: f.subs(z=1)
x + 2*y + 3
sage: f.subs(x=1, y=1, z=1)
```
\[
sage: f = x^{-1}
sage: f.subs(x=2)
1/2
sage: f.subs({x: 2})
1/2
sage: f = x + 2*y + 3*z
sage: f.subs({x: 1, y: 1, z: 1})
6
sage: f.substitute(x=1, y=1, z=1)
6
\]

**toric_coordinate_change**\((M, h=None, new\_ring=None)\)

Apply a matrix to the exponents in a Laurent polynomial.

For efficiency, we implement this directly, rather than as a substitution.

The optional argument \(h\) is a map to be applied to coefficients.

**EXAMPLES:**

\[
sage: L.<x,y> = LaurentPolynomialRing(QQ, 2)
sage: p = 2*x^2 + y - x*y
sage: p.toric_coordinate_change(Matrix([[1,-3],[1,1]]))
2*x^2*y^2 - x^{-2}*y^2 + x^{-3}*y
sage: F = GF(2)
sage: p.toric_coordinate_change(Matrix([[1,-3],[1,1]]), new\_ring=L.change\_→ring(F))
x^{-2}*y^2 + x^{-3}*y
\]

**toric_substitute**\((v, v1, a, h=None, new\_ring=None)\)

Perform a single-variable substitution up to a toric coordinate change.

The optional argument \(h\) is a map to be applied to coefficients.

**EXAMPLES:**

\[
sage: L.<x,y> = LaurentPolynomialRing(QQ, 2)
sage: p = x + y
sage: p.toric_substitute((2,3), (-1,1), 2)
1/2*x^3*y^3 + 2*x^2*y^2
sage: F = GF(5)
sage: p.toric_substitute((2,3), (-1,1), 2, new\_ring=L.change\_ring(F))
3*x^3*y^3 + 2*x^2*y^2
\]

**univariate_polynomial**\((R=None)\)

Returns a univariate polynomial associated to this multivariate polynomial.

**INPUT:**

- \(R\) - (default: None) a univariate Laurent polynomial ring

If this polynomial is not in at most one variable, then a \(ValueError\) exception is raised. The new polynomial is over the same base ring as the given \(LaurentPolynomial\) and in the variable \(x\) if no ring \(R\) is provided.

5.2. Elements of Laurent polynomial rings 535
EXAMPLES:

```python
sage: R.<x, y> = LaurentPolynomialRing(ZZ)
sage: f = 3*x^2 - 2*y^-1 + 7*x^2*y^2 + 5
sage: f.univariate_polynomial()
Traceback (most recent call last):
... TypeError: polynomial must involve at most one variable
sage: g = f(10,y); g
700*y^2 + 305 - 2*y^-1
sage: h = g.univariate_polynomial(); h
-2*y^-1 + 305 + 700*y^2
sage: h.parent()
Univariate Laurent Polynomial Ring in y over Integer Ring
sage: g.univariate_polynomial(LaurentPolynomialRing(QQ, 'z'))
-2*z^-1 + 305 + 700*z^2
```

Here’s an example with a constant multivariate polynomial:

```python
sage: g = R(1)
sage: h = g.univariate_polynomial(); h
1
sage: h.parent()
Univariate Laurent Polynomial Ring in x over Integer Ring
```

`variables(sort=True)`

Return a tuple of all variables occurring in `self`.

**INPUT:**

- `sort` – specifies whether the indices shall be sorted

**EXAMPLES:**

```python
sage: L.<x,y,z> = LaurentPolynomialRing(QQ)
sage: f = 4*x^7*z^-1 + 3*x^3*y + 2*x^4*z^-2 + x^6*y^-7
sage: f.variables()
(z, y, x)
sage: f.variables(sort=False) #random
(y, z, x)
```

```python
class sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate
Bases: sage.rings.polynomial.laurent_polynomial.LaurentPolynomial
```

A univariate Laurent polynomial in the form of $t^n \cdot f$ where $f$ is a polynomial in $t$.

**INPUT:**

- `parent` – a Laurent polynomial ring
- `f` – a polynomial (or something can be coerced to one)
- `n` – (default: 0) an integer

**AUTHORS:**

- Tom Boothby (2011) copied this class almost verbatim from `laurent_series_ring_element.pyx`, so most of the credit goes to William Stein, David Joyner, and Robert Bradshaw
- Travis Scrimshaw (09-2013): Cleaned-up and added a few extra methods
coefficients()
Return the nonzero coefficients of self.

EXAMPLES:

```
sage: R.<t> = LaurentPolynomialRing(QQ)
sage: f = -5/t^2 + t + t^2 - 10/3*t^3
sage: f.coefficients()
[-5, 1, 1, -10/3]
```

constant_coefficient()
Return the coefficient of the constant term of self.

EXAMPLES:

```
sage: R.<t> = LaurentPolynomialRing(QQ)
sage: f = 3*t^-2 - t^-1 + 3 + t^2
sage: f.constant_coefficient()
3
sage: g = -2*t^-2 + t^-1 + 3*t
sage: g.constant_coefficient()
0
```

degree()
Return the degree of self.

EXAMPLES:

```
sage: R.<x> = LaurentPolynomialRing(ZZ)
sage: g = x^2 - x^4
sage: g.degree()
4
sage: g = -10/x^5 + x^2 - x^7
sage: g.degree()
7
```

derivative(*args)
The formal derivative of this Laurent polynomial, with respect to variables supplied in args.

Multiple variables and iteration counts may be supplied. See documentation for the global `derivative()` function for more details.

See also:
_.derivative()

EXAMPLES:

```
sage: R.<x> = LaurentPolynomialRing(QQ)
sage: g = 1/x^10 - x + x^2 - x^4
sage: g.derivative()
-10*x^-11 - 1 + 2*x - 4*x^3
sage: g.derivative(x)
-10*x^-11 - 1 + 2*x - 4*x^3

sage: R.<t> = PolynomialRing(ZZ)
sage: S.<x> = LaurentPolynomialRing(R)
```

(continues on next page)


```python
sage: f = 2*t/x + (3*t^2 + 6*t)*x
sage: f.derivative()
-2*t*x^-2 + (3*t^2 + 6*t)
sage: f.derivative(x)
-2*t*x^-2 + (3*t^2 + 6*t)
sage: f.derivative(t)
2*x^-1 + (6*t + 6)*x
```

dict()
Return a dictionary representing **self**.

**EXAMPLES:**

```python
sage: R.<x,y> = ZZ[]
sage: Q.<t> = LaurentPolynomialRing(R)
sage: f = (x^3 + y/t^3)^3 + t^2; f
y^3*t^-9 + 3*x^3*y^2*t^-6 + 3*x^6*y*t^-3 + x^9 + t^2
sage: f.dict()
{-9: y^3, -6: 3*x^3*y^2, -3: 3*x^6*y, 0: x^9, 2: 1}
```

exponents()
Return the exponents appearing in **self** with nonzero coefficients.

**EXAMPLES:**

```python
sage: R.<t> = LaurentPolynomialRing(QQ)
sage: f = -5/t^2 + t + t^2 - 10/3*t^3
sage: f.exponents()
[-2, 1, 2, 3]
```

factor()
Return a Laurent monomial (the unit part of the factorization) and a factored polynomial.

**EXAMPLES:**

```python
sage: R.<t> = LaurentPolynomialRing(ZZ)
sage: f = 4*t^-7 + 3*t^3 + 2*t^4 + t^-6
sage: f.factor()
(t^-7) * (4 + t + 3*t^10 + 2*t^11)
```

gcd(right)
Return the gcd of **self** with **right** where the common divisor **d** makes both **self** and **right** into polynomials with the lowest possible degree.

**EXAMPLES:**

```python
sage: R.<t> = LaurentPolynomialRing(QQ)
sage: t.gcd(2)
1
sage: gcd(t^-2 + 1, t^-4 + 3*t^-1)
t^-4
sage: gcd((t^-2 + t)*t^3 + t^5 + t^8 + (1 + t^-2))
t^-3 + t^-1 + 1 + t^2
```

integral()
The formal integral of this Laurent series with 0 constant term.
EXAMPLES:

The integral may or may not be defined if the base ring is not a field.

```
sage: t = LaurentPolynomialRing(ZZ, 't').0
sage: f = 2*t^-3 + 3*t^2
sage: f.integral()
-t^-2 + t^3
sage: f = t^3
sage: f.integral()
Traceback (most recent call last):
  ... ArithmeticError: coefficients of integral cannot be coerced into the base ring
```

The integral of $1/t$ is $\log(t)$, which is not given by a Laurent polynomial:

```
sage: t = LaurentPolynomialRing(ZZ, 't').0
sage: f = -1/t^3 - 31/t
sage: f.integral()
Traceback (most recent call last):
  ... ArithmeticError: the integral of is not a Laurent polynomial, since $t^{-1}$ has nonzero coefficient
```

Another example with just one negative coefficient:

```
sage: A.<t> = LaurentPolynomialRing(QQ)
sage: f = -2*t^(-4)
sage: f.integral()
2/3*t^-3
sage: f.integral().derivative() == f
True
```

`inverse_of_unit()`
Return the inverse of self if a unit.

EXAMPLES:

```
sage: R.<t> = LaurentPolynomialRing(QQ)
sage: (t^-2).inverse_of_unit()
t^-2
sage: (t + 2).inverse_of_unit()
Traceback (most recent call last):
  ... ArithmeticError: element is not a unit
```

`is_constant()`
Return whether this Laurent polynomial is constant.

EXAMPLES:

```
sage: R.<x> = LaurentPolynomialRing(QQ)
sage: x.is_constant()
False
```
sage: R.one().is_constant()
True
sage: (x^-2).is_constant()
False
sage: (x^2).is_constant()
False
sage: (x^-2 + 2).is_constant()
False
sage: R(0).is_constant()
True
sage: R(42).is_constant()
True
sage: x.is_constant()
False
sage: (1/x).is_constant()
False

is_monomial()
Return True if self is a monomial; that is, if self is $x^n$ for some integer $n$.

EXAMPLES:

sage: k.<z> = LaurentPolynomialRing(QQ)
sage: z.is_monomial()
True
sage: k(1).is_monomial()
True
sage: (z+1).is_monomial()
False
sage: (z^-2909).is_monomial()
True
sage: (38*z^-2909).is_monomial()
False

is_square(root=False)
Return whether this Laurent polynomial is a square.

If root is set to True then return a pair made of the boolean answer together with None or a square root.

EXAMPLES:

sage: R.<t> = LaurentPolynomialRing(QQ)
sage: R.one().is_square()
True
sage: R(2).is_square()
False
sage: t.is_square()
False
sage: (t**-2).is_square()
True

Usage of the root option:
The answer is dependent of the base ring:

```
sage: S.<u> = LaurentPolynomialRing(QQbar)
sage: (2 + 4*u + 2*u^2).is_square()
True
```

```
sage: (2 + 4*t + 2*t^2).is_square()
False
```

```
sage: (2 + 4*u + 2*u^2).is_square()
True
```

```
is_unit()
Return True if this Laurent polynomial is a unit in this ring.

EXAMPLES:
```
sage: R.<t> = LaurentPolynomialRing(QQ)
sage: (2+t).is_unit()
False
sage: f = 2*t
sage: f.is_unit()
True
sage: 1/f
1/2*t^-1
sage: R(0).is_unit()
False
```

ALGORITHM: A Laurent polynomial is a unit if and only if its “unit part” is a unit.

```
is_zero()
Return 1 if self is 0, else return 0.

EXAMPLES:
```
sage: R.<x> = LaurentPolynomialRing(QQ)
sage: f = 1/x + x + x^2 + 3*x^4
sage: f.is_zero()
0
sage: z = 0*f
sage: z.is_zero()
1
```

```
number_of_terms()
Return the number of non-zero coefficients of self.

Also called weight, hamming weight or sparsity.

EXAMPLES:
```
5.2. Elements of Laurent polynomial rings
The method `hamming_weight()` is an alias:

```
sage: f.hamming_weight()
sage: 101
```

**polynomial_construction()**

Return the polynomial and the shift in power used to construct the Laurent polynomial $t^n u$.

**OUTPUT:**

A tuple $(u, n)$ where $u$ is the underlying polynomial and $n$ is the power of the exponent shift.

**EXAMPLES:**

```
sage: R.<x> = LaurentPolynomialRing(QQ)
sage: f = 1/x + x^2 + 3*x^4
sage: f.polynomial_construction()
(3*x^5 + x^3 + 1, -1)
```

**quo_rem(right_r)**

Attempts to divide `self` by `right` and returns a quotient and a remainder.

**EXAMPLES:**

```
sage: R.<t> = LaurentPolynomialRing(QQ)
sage: (t^-3 - t^3).quo_rem(t^-1 - t)
(1 + t^2, 0)
sage: (t^-2 + 3 + t).quo_rem(t^-4)
(1 + 3*t^2 + t^3, 0)
```

**residue()**

Return the residue of `self`.

The residue is the coefficient of $t^{-1}$.

**EXAMPLES:**

```
sage: R.<t> = LaurentPolynomialRing(QQ)
sage: f = 3*t^-2 - t^-1 + 3 + t^2
sage: f.residue()
-1
sage: g = -2*t^-2 + 4 + 3*t
sage: g.residue()
0
```

(continues on next page)
shift\( (k) \)
Return this Laurent polynomial multiplied by the power \( t^n \). Does not change this polynomial.

```
sage: R.<t> = Laurent PolynomialRing(QQ,['y'])
sage: f = (t+t^-1)^4; f
 t^-4 + 4*t^-2 + 6 + 4*t^2 + t^4
sage: f.shift(10)
t^6 + 4*t^8 + 6*t^10 + 4*t^12 + t^14
sage: f >> 10
 t^-14 + 4*t^-12 + 6*t^-10 + 4*t^-8 + t^-6
sage: f << 4
1 + 4*t^2 + 6*t^4 + 4*t^6 + t^8
```

truncate\( (n) \)
Return a polynomial with degree at most \( n - 1 \) whose \( j \)-th coefficients agree with \self \) for all \( j < n \).

```
sage: R.<x> = Laurent PolynomialRing(QQ)
sage: f = 1/x^12 + x^3 + x^5 + x^9
sage: f.truncate(10)
x^-12 + x^3 + x^5 + x^9
sage: f.truncate(5)
x^-12 + x^3
sage: f.truncate(-16)
0
```

valuation\( (p=None) \)
Return the valuation of \self \).

The valuation of a Laurent polynomial \( t^n u \) is \( n \) plus the valuation of \( u \).

```
sage: R.<x> = Laurent PolynomialRing(ZZ)
sage: f = 1/x + x^2 + 3^x^4
sage: g = 1 - x + x^2 - x^4
sage: f.valuation()
-1
sage: g.valuation()
0
```

variable_name()
Return the name of variable of \self \) as a string.

```
sage: R.<x> = Laurent PolynomialRing(QQ)
sage: f = 1/x + x^2 + 3^x^4
sage: f.variable_name()
'x'
```
variables()

Return the tuple of variables occurring in this Laurent polynomial.

EXAMPLES:

```
sage: R.<x> = LaurentPolynomialRing(QQ)
sage: f = 1/x + x^2 + 3*x^4
sage: f.variables()
(x,)
sage: R.one().variables()
()
```

5.3 MacMahon’s Partition Analysis Omega Operator

This module implements MacMahon’s Omega Operator [Mac1915], which takes a quotient of Laurent polynomials and removes all negative exponents in the corresponding power series.

5.3.1 Examples

In the following example, all negative exponents of $\mu$ are removed. The formula

$$\Omega \geq \frac{1}{(1-x\mu)(1-y/\mu)} = \frac{1}{(1-x)(1-xy)}$$

can be calculated and verified by

```
sage: L.<mu, x, y> = LaurentPolynomialRing(ZZ)
sage: MacMahonOmega(mu, 1, [1 - x*mu, 1 - y/mu])
1 * (-x + 1)^-1 * (-x*y + 1)^-1
```

5.3.2 Various

AUTHORS:

- Daniel Krenn (2016)

ACKNOWLEDGEMENT:

- Daniel Krenn is supported by the Austrian Science Fund (FWF): P 24644-N26.

5.3.3 Functions

```
sage.rings.polynomial.omega.MacMahonOmega(var, expression, denominator=None, op=<built-in function ge>, Factorization_sort=False, Factorization_simplify=True)
```

Return $\Omega_{\text{op}}$ of expression with respect to var.

To be more precise, calculate

$$\Omega_{\text{op}} \frac{n}{d_1 \ldots d_n}$$

for the numerator $n$ and the factors $d_1, \ldots, d_n$ of the denominator, all of which are Laurent polynomials in var and return a (partial) factorization of the result.

INPUT:
• var – a variable or a representation string of a variable

• expression – a Factorization of Laurent polynomials or, if denominator is specified, a Laurent polynomial interpreted as the numerator of the expression

• denominator – a Laurent polynomial or a Factorization (consisting of Laurent polynomial factors) or a tuple/list of factors (Laurent polynomials)

• op – (default: operator.ge) an operator
  At the moment only operator.ge is implemented.

• Factorization_sort (default: False) and Factorization_simplify (default: True) – are passed on to sage.structure.factorization.Factorization when creating the result

OUTPUT:
A (partial) Factorization of the result whose factors are Laurent polynomials

Note: The numerator of the result may not be factored.

REFERENCES:
• [Mac1915]
• [APR2001]

EXAMPLES:

```
sage: L.<mu, x, y, z, w> = LaurentPolynomialRing(ZZ)
sage: MacMahonOmega(mu, 1, [1 - x^mu, 1 - y/mu])
1 * (-x + 1)^-1 * (-x*y + 1)^-1
sage: MacMahonOmega(mu, 1, [1 - x^mu, 1 - y/mu, 1 - z/mu])
1 * (-x + 1)^-1 * (-x*y + 1)^-1 * (-x*z + 1)^-1
sage: MacMahonOmega(mu, 1, [1 - x^mu, 1 - y^mu, 1 - z/mu])
(-x^y*z + 1) * (-x + 1)^-1 * (-y + 1)^-1 * (-x*z + 1)^-1 * (-y*z + 1)^-1
sage: MacMahonOmega(mu, 1, [1 - x^mu, 1 - y/mu^2])
1 * (-x + 1)^-1 * (-x^2*y + 1)^-1
sage: MacMahonOmega(mu, 1, [1 - x^mu, 1 - y/mu^2, 1 - y/mu])
(x*y + 1) * (-x + 1)^-1 * (-x*y^2 + 1)^-1
sage: MacMahonOmega(mu, 1, [1 - x^mu, 1 - y^mu^2, 1 - y/mu])
(-x^2*y*z - x*y^2*z + x*y^2*z + 1) * 
(-x + 1)^-1 * (-y + 1)^-1 * (-x^2*z + 1)^-1 * (-y^2*z + 1)^-1
sage: MacMahonOmega(mu, 1, [1 - x^mu, 1 - y/mu^3])
1 * (-x + 1)^-1 * (-x^3*y + 1)^-1
sage: MacMahonOmega(mu, 1, [1 - x^mu, 1 - y^mu^3])
1 * (-x + 1)^-1 * (-x^4*y + 1)^-1
sage: MacMahonOmega(mu, 1, [1 - x^mu^3, 1 - y/mu])
(x*y^2 + x*y + 1) * (-x + 1)^-1 * (-x*y^3 + 1)^-1
sage: MacMahonOmega(mu, 1, [1 - x^mu^4, 1 - y/mu])
(x*y^3 + x*y^2 + x*y + 1) * (-x + 1)^-1 * (-x*y^4 + 1)^-1
sage: MacMahonOmega(mu, 1, [1 - x^mu^2, 1 - y/mu, 1 - z/mu])
```

(continues on next page)
\[(x^2y^z + x^y + x^z + 1) \ast
\nonumber
(-x + 1)^{-1} \ast (-x^y^z^2 + 1)^{-1} \ast (-x^z^2 + 1)^{-1}
\]
\text{sage: } MacMahonOmega(\mu, 1, [1 - x^\mu^2, 1 - y^\mu, 1 - z/\mu])
\nonumber
\[(x^y^z^2 - x^y^z + x^z + 1) \ast
\nonumber
(-x + 1)^{-1} \ast (-y + 1)^{-1} \ast (-x^z^2 + 1)^{-1} \ast (-y^z + 1)^{-1}
\]
\text{sage: } MacMahonOmega(\mu, 1, [1 - x^\mu, 1 - y^\mu, 1 - z/\mu, 1 - w/\mu])
\nonumber
\[(x^2y^z^w^2 + x^y^z^w - x^y^z^w - x^z^w + 1) \ast
\nonumber
(-x + 1)^{-1} \ast (-y + 1)^{-1} \ast (-z + 1)^{-1} \ast (-w + 1)^{-1}
\]
\text{sage: } MacMahonOmega(\mu, 1, [1 - x^\mu, 1 - y^\mu, 1 - z/\mu, 1 - w/\mu])
\nonumber
\[(x^2y^z^w^2 + x^y^z^w + x^y^z^w - x^y^z^w - x^y^z^w + 1) \ast
\nonumber
(-x + 1)^{-1} \ast (-y + 1)^{-1} \ast (-x^z + 1)^{-1} \ast (-y^z + 1)^{-1} \ast (-y^w + 1)^{-1}
\]
\text{sage: } MacMahonOmega(\mu, mu^-2, [1 - x^\mu, 1 - y/\mu])
\nonumber
x^2 \ast (-x + 1)^{-1} \ast (-x^y + 1)^{-1}
\text{sage: } MacMahonOmega(\mu, mu^-1, [1 - x^\mu, 1 - y/\mu])
\nonumber
x \ast (-x + 1)^{-1} \ast (-x^y + 1)^{-1}
\text{sage: } MacMahonOmega(\mu, mu, [1 - x^\mu, 1 - y/\mu])
\nonumber
(-x^y + y + 1) \ast (-x + 1)^{-1} \ast (-x^y + 1)^{-1}
\text{sage: } MacMahonOmega(\mu, mu^2, [1 - x^\mu, 1 - y/\mu])
\nonumber
(-x^y + y^2 + y + 1) \ast (-x + 1)^{-1} \ast (-x^y + 1)^{-1}
\text{sage: } MacMahonOmega(\mu, mu^2, (1 - x^\mu)*(1 - y/\mu)) \# not tested because not fully implemented
\nonumber
(-x^y^2 - x^y + y^2 + y + 1) \ast (-x + 1)^{-1} \ast (-x^y + 1)^{-1}
\text{sage: } MacMahonOmega(\mu, mu^2 / ((1 - x^\mu)*(1 - y/\mu))) \# not tested because not fully implemented
\nonumber
(-x^y^2 - x^y + y^2 + y + 1) \ast (-x + 1)^{-1} \ast (-x^y + 1)^{-1}

We demonstrate the different allowed input variants:

\text{sage: } MacMahonOmega(\mu, mu^-2, [1 - x^\mu, 1 - y/\mu])
\nonumber
\text{\ldots: Factorization([[\mu, 2], (1 - x^\mu, -1), (1 - y/\mu, -1)])}
\nonumber
(-x^y^2 - x^y + y^2 + y + 1) \ast (-x + 1)^{-1} \ast (-x^y + 1)^{-1}
\text{sage: } MacMahonOmega(\mu, mu^2, \ldots: Factorization([[1 - x^\mu, 1], (1 - y/\mu, 1)])
\nonumber
(-x^y^2 - x^y + y^2 + y + 1) \ast (-x + 1)^{-1} \ast (-x^y + 1)^{-1}
\text{sage: } MacMahonOmega(\mu, mu^2, [1 - x^\mu, 1 - y/\mu])
\nonumber
(-x^y^2 - x^y + y^2 + y + 1) \ast (-x + 1)^{-1} \ast (-x^y + 1)^{-1}
\text{sage: } MacMahonOmega(\mu, mu^2, (1 - x^\mu)*(1 - y/\mu)) \# not tested because not fully implemented
\nonumber
(-x^y^2 - x^y + y^2 + y + 1) \ast (-x + 1)^{-1} \ast (-x^y + 1)^{-1}
\text{sage: } MacMahonOmega(\mu, mu^2 / ((1 - x^\mu)*(1 - y/\mu))) \# not tested because not fully implemented
\nonumber
(-x^y^2 - x^y + y^2 + y + 1) \ast (-x + 1)^{-1} \ast (-x^y + 1)^{-1}

\text{sage.rings.polynomial.omega.Omega_ge}(a, exponents)
\text{Return } \Omega_\geq \text{ of the expression specified by the input.}

To be more precise, calculate
\[\Omega_\geq = \frac{\mu^a}{(1 - z_0 \mu^{e_0}) \ldots (1 - z_{n-1} \mu^{e_{n-1}})}\]
and return its numerator and a factorization of its denominator. Note that \(z_0, \ldots, z_{n-1}\) only appear in the output, but not in the input.
INPUT:

- \(a\) – an integer
- \(\text{exponents}\) – a tuple of integers

OUTPUT:

A pair representing a quotient as follows: Its first component is the numerator as a Laurent polynomial, its second component a factorization of the denominator as a tuple of Laurent polynomials, where each Laurent polynomial \(z\) represents a factor \(1 - z\).

The parents of these Laurent polynomials is always a Laurent polynomial ring in \(z_0, \ldots, z_{n-1}\) over \(\mathbb{Z}\), where \(n\) is the length of \(\text{exponents}\).

EXAMPLES:

```sage
from sage.rings.polynomial.omega import Omega_ge
sage: Omega_ge(0, (1, -2))
(1, (z0, z0^2*z1))
sage: Omega_ge(0, (1, -3))
(1, (z0, z0^3*z1))
sage: Omega_ge(0, (1, -4))
(1, (z0, z0^4*z1))
sage: Omega_ge(0, (2, -1))
(z0*z1 + 1, (z0, z0*z1^2))
sage: Omega_ge(0, (3, -1))
(z0*z1^2 + z0^3*z1 + 1, (z0, z0^3*z1^3))
sage: Omega_ge(0, (4, -1))
(z0^3*z1^3 + z0^4*z1^2 + z0^4*z1 + 1, (z0, z0^4*z1^4))
sage: Omega_ge(0, (1, 1, -2))
(-z0^2*z1*z2 - z0*z1^2*z2 + z0^2*z1^2*z2 + 1, (z0, z1, z0^2*z2, z1^2*z2))
sage: Omega_ge(0, (2, 1, -1))
(z0*z1*z2 + z0^2*z1 + z0^2*z2 + 1, (z0, z0^2*z1^2, z0^2*z2^2))
sage: Omega_ge(0, (2, -1, -1))
(-z0^2*z1^2*z2 - z0^3*z1*z2 + z0^3*z2 + 1, (z0, z1, z0^2*z2^2, z1*z2))
sage: Omega_ge(0, (2, -2))
(-z0^2*z1 + 1, (z0, z0^2*z1, z0^2*z2))
sage: Omega_ge(0, (2, -3))
(z0^4 + z0^3*z1^3 + 2*z0^2*z1^3*z2^2 - z0^5*z1^3*z2 + z0^2*z2^2 - z0^2*z2 + 1, (z0, z1, z0^2*z2, z0^2*z2, z1^3*z2))
sage: Omega_ge(0, (3, 3, -3))
(-z0^3*z1^3*z2^3 + 2*z0^2*z1^3*z2^2 - z0^5*z1^3*z2 + z0^2*z2^2 - 2*z0^2*z2 + 1, (z0, z1, z0^2*z2, z0^2*z2, z1^3*z2))
```

```
sage: homogeneous_symmetric_function(j, x)
Return a complete homogeneous symmetric polynomial (Wikipedia article Complete_homogeneous_symmetric_polynomial).
```

5.3. MacMahon’s Partition Analysis Omega Operator 547
INPUT:
- \( j \) – the degree as a nonnegative integer
- \( x \) – an iterable of variables

OUTPUT:
A polynomial of the common parent of all entries of \( x \)

EXAMPLES:

```python
sage: from sage.rings.polynomial.omega import homogeneous_symmetric_function
sage: P = PolynomialRing(ZZ, 'X', 3)
sage: homogeneous_symmetric_function(0, P.gens())
1
sage: homogeneous_symmetric_function(1, P.gens())
X0 + X1 + X2
sage: homogeneous_symmetric_function(2, P.gens())
X0^2 + X0*X1 + X1^2 + X0*X2 + X1*X2 + X2^2
sage: homogeneous_symmetric_function(3, P.gens())
X0^3 + X0^2*X1 + X0*X1^2 + X1^3 + X0^2*X2 + X0*X1*X2 + X1^2*X2 + X0*X2^2 + X1*X2^2 + X2^3
```

`sage.rings.polynomial.omega.partition(items, predicate=<class 'bool'>)`
Split items into two parts by the given predicate.

INPUT:
- \( item \) – an iterator
- \( predicate \) – a function

OUTPUT:
A pair of iterators; the first contains the elements not satisfying the \( predicate \), the second the elements satisfying the \( predicate \).

ALGORITHM:
Source of the code: `http://nedbatchelder.com/blog/201306/filter_a_list_into_two_parts.html`

EXAMPLES:

```python
sage: from sage.rings.polynomial.omega import partition
sage: E, O = partition(srange(10), is_odd)
sage: tuple(E), tuple(O)
((0, 2, 4, 6, 8), (1, 3, 5, 7, 9))
```
6.1 Infinite Polynomial Rings

By Infinite Polynomial Rings, we mean polynomial rings in a countably infinite number of variables. The implementation consists of a wrapper around the current finite polynomial rings in Sage.

AUTHORS:
• Simon King <simon.king@nuigalway.ie>
• Mike Hansen <mhansen@gmail.com>

An Infinite Polynomial Ring has finitely many generators \(x, y, \ldots\) and infinitely many variables of the form \(x_0, x_1, x_2, \ldots, y_0, y_1, y_2, \ldots\). We refer to the natural number \(n\) as the index of the variable \(x_n\).

INPUT:
• \(R\), the base ring. It has to be a commutative ring, and in some applications it must even be a field
• \(names\), a list of generator names. Generator names must be alpha-numeric.
• \(order\) (optional string). The default order is 'lex' (lexicographic). 'deglex' is degree lexicographic, and 'degrevlex' (degree reverse lexicographic) is possible but discouraged.

Each generator \(x\) produces an infinite sequence of variables \(x[1], x[2], \ldots\) which are printed on screen as \(x_1, x_2, \ldots\) and are latex typeset as \(x_1, x_2\). Then, the Infinite Polynomial Ring is formed by polynomials in these variables.

By default, the monomials are ordered lexicographically. Alternatively, degree (reverse) lexicographic ordering is possible as well. However, we do not guarantee that the computation of Groebner bases will terminate in this case.

In either case, the variables of a Infinite Polynomial Ring \(X\) are ordered according to the following rule:

\[X.gen(i)[m] > X.gen(j)[n]\] if and only if \(i < j\) or \((i=j\) and \(m > n)\)

We provide a 'dense' and a 'sparse' implementation. In the dense implementation, the Infinite Polynomial Ring carries a finite polynomial ring that comprises all variables up to the maximal index that has been used so far. This is potentially a very big ring and may also comprise many variables that are not used.

In the sparse implementation, we try to keep the underlying finite polynomial rings small, using only those variables that are really needed. By default, we use the dense implementation, since it usually is much faster.

EXAMPLES:

\[
\begin{align*}
\text{sage: } & X.<x,y> = \text{InfinitePolynomialRing}(\mathbb{Z}Z, \text{implementation='sparse'}) \\
\text{sage: } & A.<\alpha, \beta> = \text{InfinitePolynomialRing}(\mathbb{Q}Q, \text{order='deglex'}) \\
\text{sage: } & f = x[5] + 2; f
\end{align*}
\] (continues on next page)
It has some advantages to have an underlying ring that is not univariate. Hence, we always have at least two variables:

```sage
sage: g._p.parent()
Multivariate Polynomial Ring in y_1, y_0 over Integer Ring
```

Of course, we provide the usual polynomial arithmetic:

```sage
f2 = alpha[5] + 2; f2
alpha_5 + 2
g2 = 3*beta[1]; g2
3*beta_1
```

There is a permutation action on the variables, by permuting positive variable indices:

```sage
P = Permutation(((10,1)))
p^P
x_5*x_1^2 + 3*x_1^2*y_10 + 2*x_1^2
```

Note that \( x_0^P = x_0 \), since the permutations only change positive variable indices.

We also implemented ideals of Infinite Polynomial Rings. Here, it is thoroughly assumed that the ideals are set-wise invariant under the permutation action. We therefore refer to these ideals as Symmetric Ideals. Symmetric Ideals are finitely generated modulo addition, multiplication by ring elements and permutation of variables. If the base ring is a field, one can compute Symmetric Groebner Bases:

```sage
J = A*(alpha[1]*beta[2])
J.groebner_basis()
[alpha_1*beta_2, alpha_2*beta_1]
```

For more details, see `SymmetricIdeal`.

Infinite Polynomial Rings can have any commutative base ring. If the base ring of an Infinite Polynomial Ring is a (classical or infinite) Polynomial Ring, then our implementation tries to merge everything into one ring. The basic requirement is that the monomial orders match. In the case of two Infinite Polynomial Rings, the implementations must match. Moreover, name conflicts should be avoided. An overlap is only accepted if the order of variables can be uniquely inferred, as in the following example:
This is also allowed if finite polynomial rings are involved:

```python
sage: A.<a_3,a_1,b_1,c_2,c_0> = ZZ[]
sage: B.<b,c,d> = InfinitePolynomialRing(A, order='degrevlex')
sage: B
Infinite polynomial ring in b, c, d over Multivariate Polynomial Ring in a_3, a_1 over \Integer Ring
```

It is no problem if one generator of the Infinite Polynomial Ring is called `x` and one variable of the base ring is also called `x`. This is since no variable of the Infinite Polynomial Ring will be called `x`. However, a problem arises if the underlying classical Polynomial Ring has a variable `x_1`, since this can be confused with a variable of the Infinite Polynomial Ring. In this case, an error will be raised:

```python
sage: X.<x,y_1> = ZZ[]
sage: Y.<x,z> = InfinitePolynomialRing(X)
```

Note that `X` is not merged into `Y`; this is since the monomial order of `X` is ‘degrevlex’, but of `Y` is ‘lex’.

```python
sage: Y
Infinite polynomial ring in x, z over Multivariate Polynomial Ring in x, y_1 over \Integer Ring
```

The variable `x` of `X` can still be interpreted in `Y`, although the first generator of `Y` is called `x` as well:

```python
sage: x
x
sage: X('x')
x
sage: Y(X('x'))
x
sage: Y('x')
x
```

But there is only merging if the resulting monomial order is uniquely determined. This is not the case in the following examples, and thus an error is raised:

```python
sage: X.<y_1,x> = PolynomialRing(ZZ,order='lex')
sage: Y.<z,y> = InfinitePolynomialRing(X)
Traceback (most recent call last):
...  CoercionException: Overlapping variables ('y', 'z'),['y_1']) are incompatible
sage: Y.<z,y> = InfinitePolynomialRing(X)
Traceback (most recent call last):
...  CoercionException: Overlapping variables ('z', 'y'),['y_1']) are incompatible
sage: X.<x_3,y_1,y_2> = PolynomialRing(ZZ,order='lex')
sage: Y.<y> = InfinitePolynomialRing(X)
Traceback (most recent call last):
```

(continues on next page)
CoercionException: Overlapping variables ('y',),['y_1', 'y_2']) are incompatible

If the type of monomial orderings (e.g., ‘degrevlex’ versus ‘lex’) or if the implementations don’t match, there is no simplified construction available:

```python
sage: X.<x,y> = InfinitePolynomialRing(ZZ)
sage: Y.<z> = InfinitePolynomialRing(X,order='degrevlex')
sage: Y
Infinite polynomial ring in z over Infinite polynomial ring in x, y over Integer Ring
sage: Y.<z> = InfinitePolynomialRing(X,implementation='sparse')
sage: Y
Infinite polynomial ring in z over Infinite polynomial ring in x, y over Integer Ring
```

class `sage.rings.polynomial.infinite_polynomial_ring.GenDictWithBasering`(`parent`, `start`)  
Bases: object

A dictionary-like class that is suitable for usage in `sage_eval`.  
This pseudo-dictionary accepts strings as index, and then walks down a chain of base rings of (infinite) polynomial rings until it finds one ring that has the given string as variable name, which is then returned.

EXAMPLES:

```python
sage: R.<a,b> = InfinitePolynomialRing(QQ['t'])
sage: D = R.gens_dict() # indirect doctest
def next(D):
    return a  
```

def next(D):
    return a

next()  
Return a dictionary that can be used to interpret strings in the base ring of `self`.

EXAMPLES:

```python
sage: R.<a,b> = InfinitePolynomialRing(QQ['t'])
sage: D = R.gens_dict()
sage: next(D)
t^2
```

class `sage.rings.polynomial.infinite_polynomial_ring.InfiniteGenDict`(`Gens`)  
Bases: object

A dictionary-like class that is suitable for usage in `sage_eval`.

The generators of an Infinite Polynomial Ring are not variables. Variables of an Infinite Polynomial Ring are
returned by indexing a generator. The purpose of this class is to return a variable of an Infinite Polynomial Ring, given its string representation.

**EXAMPLES:**

```
sage: R.<a,b> = InfinitePolynomialRing(ZZ)
sage: D = R.gens_dict() # indirect doctest
sage: D._D
[InfiniteGenDict defined by ['a', 'b'], {'1': 1}]
sage: D._D[0]['a_15']
a_15
sage: type(_)
<class 'sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial_dense'>
sage: sage_eval('3*a_3*b_5-1/2*a_7', D._D[0])
-1/2*a_7 + 3*a_3*b_5
```

**class** `sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialGen(parent, name)`

**Bases:** `sage.structure.sage_object.SageObject`

This class provides the object which is responsible for returning variables in an infinite polynomial ring (implemented in `__getitem__()`).

**EXAMPLES:**

```
sage: X.<x1,x2> = InfinitePolynomialRing(RR)
sage: x1
x1_`

sage: x1[5]
x1_5
sage: x1 == loads(dumps(x1))
True
```

**class** `sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRingFactory`

**Bases:** `sage.structure.factory.UniqueFactory`

A factory for creating infinite polynomial ring elements. It handles making sure that they are unique as well as handling pickling. For more details, see `UniqueFactory` and `infinite_polynomial_ring`.

**EXAMPLES:**

```
sage: A.<a> = InfinitePolynomialRing(QQ)
sage: B.<b> = InfinitePolynomialRing(A)
sage: B.construction()
[Infpoly([a,b], "lex", "dense"), Rational Field]
sage: R.<a,b> = InfinitePolynomialRing(QQ)
sage: R is B
True
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: X2.<x> = InfinitePolynomialRing(QQ, implementation='sparse')
sage: X is X2
False
sage: X is loads(dumps(X))
True
```

`create_key(R, names=\"x\", order=\"lex\", implementation=\"dense\")`

Creates a key which uniquely defines the infinite polynomial ring.
create_object(version, key)
Return the infinite polynomial ring corresponding to the key key.

class sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing_dense(R, names, order)
Bases: sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing_sparse
Dense implementation of Infinite Polynomial Rings
Compared with InfinitePolynomialRing_sparse, from which this class inherits, it keeps a polynomial ring that comprises all elements that have been created so far.

collection()
Return the construction of self.

OUTPUT:
A pair F, R, where F is a construction functor and R is a ring, so that F(R) is self.

EXAMPLES:

sage: R.<x,y> = InfinitePolynomialRing(GF(5))
sage: R.construction()
[InfPoly{[x,y], "lex", "dense"}, Finite Field of size 5]

polynomial_ring()
Return the underlying finite polynomial ring.

Note: The ring returned can change over time as more variables are used.
Since the rings are cached, we create here a ring with variable names that do not occur in other doc tests, so that we avoid side effects.

EXAMPLES:

sage: X.<xx, yy> = InfinitePolynomialRing(ZZ)
sage: X.polynomial_ring()
Multivariate Polynomial Ring in xx_0, yy_0 over Integer Ring
sage: a = yy[3]
sage: X.polynomial_ring()
Multivariate Polynomial Ring in xx_3, xx_2, xx_1, xx_0, yy_3, yy_2, yy_1, yy_0 over Integer Ring

tensor_with_ring(R)
Return the tensor product of self with another ring.

INPUT:
R - a ring.

OUTPUT:
An infinite polynomial ring that, mathematically, can be seen as the tensor product of self with R.

NOTE:
It is required that the underlying ring of self coerces into R. Hence, the tensor product is in fact merely an extension of the base ring.

EXAMPLES:
sage: R.<a,b> = InfinitePolynomialRing(ZZ, implementation='sparse')
sage: R.tensor_with_ring(QQ)
Infinite polynomial ring in a, b over Rational Field
sage: R
Infinite polynomial ring in a, b over Integer Ring

The following tests against a bug that was fixed at trac ticket #10468:

sage: R.<x,y> = InfinitePolynomialRing(QQ, implementation='sparse')
sage: R.tensor_with_ring(QQ) is R
True

class sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing_sparse(R, names, order)

Bases: sage.rings.ring.CommutativeRing

Sparse implementation of Infinite Polynomial Rings.

An Infinite Polynomial Ring with generators \(x_*, y_*, \ldots\) over a field \(F\) is a free commutative \(F\)-algebra generated by \(x_0, x_1, x_2, \ldots, y_0, y_1, y_2, \ldots\) and is equipped with a permutation action on the generators, namely \(x_n^P = x_{P(n)}, y_n^P = y_{P(n)}\) for any permutation \(P\) (note that variables of index zero are invariant under such permutation).

It is known that any permutation invariant ideal in an Infinite Polynomial Ring is finitely generated modulo the permutation action – see SymmetricIdeal for more details.

Usually, an instance of this class is created using InfinitePolynomialRing with the optional parameter implementation='sparse'. This takes care of uniqueness of parent structures. However, a direct construction is possible, in principle:

sage: X.<x,y> = InfinitePolynomialRing(QQ, implementation='sparse')
sage: Y.<x,y> = InfinitePolynomialRing(QQ, implementation='sparse')
sage: X is Y
True

sage: from sage.rings.polynomial.infinite_polynomial_ring import ...

− InfinitePolynomialRing_sparse
sage: Z = InfinitePolynomialRing_sparse(QQ, ['x','y'], 'lex')

Nevertheless, since infinite polynomial rings are supposed to be unique parent structures, they do not evaluate equal.

sage: Z == X False

The last parameter ('lex' in the above example) can also be 'deglex' or 'degrevlex'; this would result in an Infinite Polynomial Ring in degree lexicographic or degree reverse lexicographic order.

See infinite_polynomial_ring for more details.

characteristic()

Return the characteristic of the base field.

EXAMPLES:

sage: X.<x,y> = InfinitePolynomialRing(GF(25,'a'))
sage: X
Infinite polynomial ring in x, y over Finite Field in a of size 5^2

(continues on next page)
construction()  
Return the construction of self.

OUTPUT:
A pair \((F, R)\), where \(F\) is a construction functor and \(R\) is a ring, so that \(F(R)\) is self.

EXAMPLES:

```python
sage: R.<x,y> = InfinitePolynomialRing(GF(5))
sage: R.construction()
[InfPoly{[x,y], "lex", "dense"}, Finite Field of size 5]
```

gen\((i=None)\)  
Return the \(i\)th 'generator' (see the description in \(\text{ngens()}\)) of this infinite polynomial ring.

EXAMPLES:

```python
sage: X = InfinitePolynomialRing(QQ)
sage: x = X.gen()
sage: x[1]
x_1
sage: X.gen() is X.gen(0)
True
sage: XX = InfinitePolynomialRing(GF(5))
sage: XX.gen(0) is XX.gen()
True
```

gens_dict()  
Return a dictionary-like object containing the infinitely many \{var_name:variable\} pairs.

EXAMPLES:

```python
sage: R = InfinitePolynomialRing(ZZ, 'a')
sage: D = R.gens_dict()
sage: D
GenDict of Infinite polynomial ring in a over Integer Ring
sage: D['a_5']
a_5
```

is_field(*args, **kwds)  
Return False since Infinite Polynomial Rings are never fields.

Since Infinite Polynomial Rings must have at least one generator, they have infinitely many variables and thus never are fields.

EXAMPLES:

```python
sage: R.<x, y> = InfinitePolynomialRing(QQ)
sage: R.is_field()
False
```
**is_integral_domain**(*args, **kwds*)

An infinite polynomial ring is an integral domain if and only if the base ring is. Arguments are passed to is_integral_domain method of base ring.

**EXAMPLES:**

```
sage: R.<x, y> = InfinitePolynomialRing(QQ)
sage: R.is_integral_domain()
True
```

**is_noetherian**()

Return False, since polynomial rings in infinitely many variables are never Noetherian rings.

Since Infinite Polynomial Rings must have at least one generator, they have infinitely many variables and are thus not noetherian, as a ring.

**Note:** Infinite Polynomial Rings over a field $F$ are noetherian as $F(G)$ modules, where $G$ is the symmetric group of the natural numbers. But this is not what the method is_noetherian() is answering.

**krull_dimension**(*args, **kwds*)

Return Infinity, since polynomial rings in infinitely many variables have infinite Krull dimension.

**EXAMPLES:**

```
sage: R.<x, y> = InfinitePolynomialRing(QQ)
sage: R.krull_dimension()
+Infinity
```

**ngens**()

Return the number of generators for this ring.

Since there are countably infinitely many variables in this polynomial ring, by ‘generators’ we mean the number of infinite families of variables. See infinite_polynomial_ring for more details.

**EXAMPLES:**

```
sage: X.<x> = InfinitePolynomialRing(ZZ)
sage: X.ngens()
1
sage: X.<x1,x2> = InfinitePolynomialRing(QQ)
sage: X.ngens()
2
```

**one**()

**order**()

Return Infinity, since polynomial rings have infinitely many elements.

**EXAMPLES:**

```
sage: R.<x> = InfinitePolynomialRing(GF(2))
sage: R.order()
+Infinity
```

**tensor_with_ring**(*R*)

Return the tensor product of self with another ring.
INPUT:
\( R \) - a ring.

OUTPUT:
An infinite polynomial ring that, mathematically, can be seen as the tensor product of \( \text{self} \) with \( R \).

NOTE:
It is required that the underlying ring of \( \text{self} \) coerces into \( R \). Hence, the tensor product is in fact merely an extension of the base ring.

EXAMPLES:

```python
sage: R.<a,b> = InfinitePolynomialRing(ZZ)
sage: R.tensor_with_ring(QQ)
Infinite polynomial ring in a, b over Rational Field
sage: R
Infinite polynomial ring in a, b over Integer Ring
```

The following tests against a bug that was fixed at trac ticket \#10468:

```python
sage: R.<x,y> = InfinitePolynomialRing(QQ)
sage: R.tensor_with_ring(QQ) is R
True
```

\texttt{varname_key(x)}

Key for comparison of variable names.

INPUT:
\( x \) – a string of the form \( a+\_'+\text{str}(n) \), where \( a \) is the name of a generator, and \( n \) is an integer

RETURN:
a key used to sort the variables

THEORY:
The order is defined as follows:

\( x < y \iff \text{the string } x.\text{split}(\_'\)[0] \text{ is later in the list of generator names of self than } y.\text{split}(\_'\)[0], \text{ or } (x.\text{split}(\_'\)[0]==y.\text{split}(\_'\)[0] \text{ and } \text{int}(x.\text{split}(\_'\)[1])<\text{int}(y.\text{split}(\_'\)[1]))\)

EXAMPLES:

```python
sage: X.<alpha,beta> = InfinitePolynomialRing(ZZ)
sage: X.varname_key('alpha_1')
(0, 1)
sage: X.varname_key('beta_10')
(-1, 10)
sage: X.varname_key('beta_1')
(-1, 1)
sage: X.varname_key('alpha_10')
(0, 10)
sage: X.varname_key('alpha_1')
(0, 1)
sage: X.varname_key('alpha_10')
(0, 10)
```
6.2 Elements of Infinite Polynomial Rings

AUTHORS:

• Simon King <simon.king@nuigalway.ie>
• Mike Hansen <mhansen@gmail.com>

An Infinite Polynomial Ring has generators $x_*, y_*, \ldots$, so that the variables are of the form $x_0, x_1, x_2, \ldots, y_0, y_1, y_2, \ldots, \ldots$ (see \texttt{infinite_polynomial_ring}). Using the generators, we can create elements as follows:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: a = x[3]
sage: b = y[4]
sage: a
x_3
sage: b
y_4
sage: c = a*b + a^3 - 2*b^4
sage: c
x_3^3 + x_3*y_4 - 2*y_4^4
```

Any Infinite Polynomial Ring $X$ is equipped with a monomial ordering. We only consider monomial orderings in which:

$$X\text{.gen}(i)[m] > X\text{.gen}(j)[n] \iff i<j, \text{ or } i==j \text{ and } m>n$$

Under this restriction, the monomial ordering can be lexicographic (default), degree lexicographic, or degree reverse lexicographic. Here, the ordering is lexicographic, and elements can be compared as usual:

```
sage: X._order
'lex'
sage: a > b
True
```

Note that, when a method is called that is not directly implemented for ‘InfinitePolynomial’, it is tried to call this method for the underlying \texttt{classical} polynomial. This holds, e.g., when applying the \texttt{latex} function:

```
sage: latex(c)
x_{3}^{3} + x_{3} y_{4} - 2 y_{4}^{4}
```

There is a permutation action on Infinite Polynomial Rings by permuting the indices of the variables:

```
sage: P = Permutation(((4,5),(2,3)))
sage: c^P
x_2^3 + x_2*y_5 - 2*y_5^4
```

Note that $P(0)==0$, and thus variables of index zero are invariant under the permutation action. More generally, if $P$ is any callable object that accepts non-negative integers as input and returns non-negative integers, then $c^P$ means to apply $P$ to the variable indices occurring in $c$.

```
sage: rings.polynomial.infinite_polynomial_element.InfinitePolynomial(A, p)
Create an element of a Polynomial Ring with a Countably Infinite Number of Variables.

Usually, an InfinitePolynomial is obtained by using the generators of an Infinite Polynomial Ring (see \texttt{infinite_polynomial_ring}) or by conversion.

INPUT:

• $A$ – an Infinite Polynomial Ring.
• *p* – a *classical* polynomial that can be interpreted in A.

**ASSUMPTIONS:**

In the dense implementation, it must be ensured that the argument *p* coerces into A._P by a name preserving conversion map.

In the sparse implementation, in the direct construction of an infinite polynomial, it is *not* tested whether the argument *p* makes sense in A.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.infinite_polynomial_element import...
    → InfinitePolynomial
sage: X.<alpha> = InfinitePolynomialRing(ZZ)

Currently, *P* and *X._P* (the underlying polynomial ring of *X*) both have two variables:

```python
sage: X._P
Multivariate Polynomial Ring in alpha_1, alpha_0 over Integer Ring
```

By default, a coercion from *P* to *X._P* would not be name preserving. However, this is taken care for; a name preserving conversion is impossible, and by consequence an error is raised:

```python
sage: InfinitePolynomial(X, (alpha_1+alpha_2)^2)
Traceback (most recent call last):
...<br>
TypeError: Could not find a mapping of the passed element to this ring.
```

When extending the underlying polynomial ring, the construction of an infinite polynomial works:

```python
sage: alpha[2]
alpha_2
sage: InfinitePolynomial(X, (alpha_1+alpha_2)^2)
alpha_2^2 + 2*alpha_2*alpha_1 + alpha_1^2
```

In the sparse implementation, it is not checked whether the polynomial really belongs to the parent, and when it does not, the results may be unexpected due to coercions:

```python
sage: Y.<alpha,beta> = InfinitePolynomialRing(GF(2), implementation='sparse')
sage: a = (alpha_1+alpha_2)^2
sage: InfinitePolynomial(Y, a)
alpha_0^2 + beta_0^2
```

However, it is checked when doing a conversion:

```python
sage: Y(a)
alpha_2^2 + alpha_1^2
```

**class** `sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial_dense(A, p)`

**Bases:** `sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial_sparse`

Element of a dense Polynomial Ring with a Countably Infinite Number of Variables.

**INPUT:**

• *A* – an Infinite Polynomial Ring in dense implementation
• \( p \) – a classical polynomial that can be interpreted in \( A \).

Of course, one should not directly invoke this class, but rather construct elements of \( A \) in the usual way.

This class inherits from \texttt{InfinitePolynomial\_sparse}. See there for a description of the methods.

\begin{Verbatim}
\texttt{class} \ \texttt{sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial\_sparse}(A, p) \\
\texttt{Bases:} \ \texttt{sage.structure.element.RingElement}
\end{Verbatim}

Element of a sparse Polynomial Ring with a Countably Infinite Number of Variables.

INPUT:

• \( A \) – an Infinite Polynomial Ring in sparse implementation

• \( p \) – a classical polynomial that can be interpreted in \( A \).

Of course, one should not directly invoke this class, but rather construct elements of \( A \) in the usual way.

EXAMPLES:

\begin{Verbatim}
sage: A.<a> = QQ[]
sage: B.<b,c> = InfinitePolynomialRing(A,implementation='sparse')
sage: p = a*b[100] + 1/2*c[4]
sage: p
a*b_100 + 1/2*c_4
sage: p.parent()
Infinite polynomial ring in b, c over Univariate Polynomial Ring in a over Rational Field
sage: p.polynomial().parent()
Multivariate Polynomial Ring in b_100, b_0, c_4, c_0 over Univariate Polynomial Ring in a over Rational Field
\end{Verbatim}

\begin{Verbatim}
\texttt{coefficient(monomial)}

Returns the coefficient of a monomial in this polynomial.

INPUT:

• A monomial (element of the parent of self) or

• a dictionary that describes a monomial (the keys are variables of the parent of self, the values are the corresponding exponents)

EXAMPLES:

We can get the coefficient in front of monomials:

\begin{Verbatim}
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: a
2*x_1
sage: a.coefficient(x[0])
2
sage: a.coefficient(x[1])
2*x_0 + 1
sage: a.coefficient(x[2])
1
sage: a.coefficient(x[0]*x[1])
2
\end{Verbatim}

We can also pass in a dictionary:
footprint()
Leading exponents sorted by index and generator.

OUTPUT:
D – a dictionary whose keys are the occurring variable indices.
D[s] is a list [i_1, ..., i_n], where i_j gives the exponent of self.parent().gen(j)[s] in the leading term of self.

EXAMPLES:

```python
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: sorted(p.footprint().items())
[(1, [2, 3]), (30, [1, 0])]
```

gcd(x)
computes the greatest common divisor

EXAMPLES:

```python
sage: R.<x> = InfinitePolynomialRing(QQ)
sage: p1 = x[0]+x[1]**2
sage: gcd(p1, p1+3)
1
sage: gcd(p1, p1) == p1
True
```

is_nilpotent()
Return True if self is nilpotent, i.e., some power of self is 0.

EXAMPLES:

```python
sage: R.<x> = InfinitePolynomialRing(QQbar)
sage: (x[0]+x[1]).is_nilpotent()
False
sage: R(0).is_nilpotent()
True
sage: _.<x> = InfinitePolynomialRing(Zmod(4))
sage: (2*x[0]).is_nilpotent()
True
sage: (2+x[4]*x[7]).is_nilpotent()
False
sage: _.<y> = InfinitePolynomialRing(Zmod(100))
sage: (5+2*y[0] + 10*(y[0]^2+y[1]^2)).is_nilpotent()
False
True
```

is_unit()
Answer whether self is a unit.

EXAMPLES:
sage: R1.<x,y> = InfinitePolynomialRing(ZZ)
sage: R2.<a,b> = InfinitePolynomialRing(QQ)
sage: (1+x[2]).is_unit()
False
sage: R1(1).is_unit()
True
sage: R1(2).is_unit()
False
sage: R2(2).is_unit()
True
sage: (1+a[2]).is_unit()
False

Check that trac ticket #22454 is fixed:
sage: _.<x> = InfinitePolynomialRing(Zmod(4))
sage: (1 + 2*x[0]).is_unit()
True
sage: (x[0]*x[1]).is_unit()
False
sage: _.<x> = InfinitePolynomialRing(Zmod(900))
sage: (7+150*x[0] + 30*x[1] + 120*x[1]*x[100]).is_unit()
True

lc()
The coefficient of the leading term of self.

EXAMPLES:
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: p.lc()
3

lm()
The leading monomial of self.

EXAMPLES:
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: p.lm()
x_10*x_1^2*y_1^3

lt()
The leading term (= product of coefficient and monomial) of self.

EXAMPLES:
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: p.lt()
3*x_10*x_1^2*y_1^3

max_index()Return the maximal index of a variable occurring in self, or -1 if self is scalar.
EXAMPLES:

```python
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: p.max_index()
4
sage: x[0].max_index()
0
sage: X(10).max_index()
-1
```

`polynomial()`

Return the underlying polynomial.

EXAMPLES:

```python
sage: X.<x,y> = InfinitePolynomialRing(GF(7))
sage: p = x[2]*y[1]+3*y[0]
sage: p
x_2*y_1 + 3*y_0
sage: p.polynomial()
x_2*y_1 + 3*y_0
sage: p.polynomial().parent()
Multivariate Polynomial Ring in x_2, x_1, x_0, y_2, y_1, y_0 over Finite Field of size 7
sage: p.parent()
Infinite polynomial ring in x, y over Finite Field of size 7
```

`reduce(I, tailreduce=False, report=None)`

Symmetrical reduction of `self` with respect to a symmetric ideal (or list of Infinite Polynomials).

INPUT:

- `I` – a `SymmetricIdeal` or a list of Infinite Polynomials.
- `tailreduce` – (bool, default `False`) Tail reduction is performed if this parameter is `True`.
- `report` – (object, default `None`) If not `None`, some information on the progress of computation is printed, since reduction of huge polynomials may take a long time.

OUTPUT:

Symmetrical reduction of `self` with respect to `I`, possibly with tail reduction.

THEORY:

Reducing an element `p` of an Infinite Polynomial Ring `X` by some other element `q` means the following:

1. Let `M` and `N` be the leading terms of `p` and `q`.
2. Test whether there is a permutation `P` that does not diminish the variable indices occurring in `N` and preserves their order, so that there is some term `T ∈ X` with `TN^P = M`. If there is no such permutation, return `p`.
3. Replace `p` by `p − Tq^P` and continue with step 1.

EXAMPLES:

```python
sage: X.<x,y> = InfinitePolynomialRing(QQ)
```

(continues on next page)
sage: p.reduce([y[2]*x[1]^2])
x_3^3*y_2 + y_3*y_1^2

The preceding is correct: If a permutation turns \( y[2]*x[1]^2 \) into a factor of the leading monomial \( y[2]*x[3]^3 \) of \( p \), then it interchanges the variable indices 1 and 2; this is not allowed in a symmetric reduction. However, reduction by \( y[1]*x[2]^2 \) works, since one can change variable index 1 into 2 and 2 into 3:

sage: p.reduce([y[1]*x[2]^2])
y_3*y_1^2

The next example shows that tail reduction is not done, unless it is explicitly advised. The input can also be a Symmetric Ideal:

sage: I = (y[3])*X
sage: p.reduce(I)
x_3^3*y_2 + y_3*y_1^2
sage: p.reduce(I, tailreduce=True)
x_3^3*y_2

Last, we demonstrate the report option:

sage: p.reduce(I, tailreduce=True, report=True)
:T[2]:>
x_1^2 + y_2^2

The output ‘:’ means that there was one reduction of the leading monomial. ‘T[2]’ means that a tail reduction was performed on a polynomial with two terms. At ‘>’, one round of the reduction process is finished (there could only be several non-trivial rounds if \( I \) was generated by more than one polynomial).

**ring()**

The ring which \( self \) belongs to.

This is the same as \( self.parent() \).

**EXAMPLES:**

sage: X.<x,y> = InfinitePolynomialRing(ZZ,implementation='sparse')
sage: p.ring()
Infinite polynomial ring in x, y over Integer Ring

**squeezed()**

Reduce the variable indices occurring in \( self \).

**OUTPUT:**

Apply a permutation to \( self \) that does not change the order of the variable indices of \( self \) but squeezes them into the range 1,2,...

**EXAMPLES:**

sage: X.<x,y> = InfinitePolynomialRing(QQ,implementation='sparse')

(continues on next page)
\textbf{sage:} \texttt{p.squeezed()}
\begin{verbatim}
x_2*y_4 + x_1*y_3
\end{verbatim}

\textbf{stretch}(k)
Stretch self by a given factor.

\textbf{INPUT:}

k – an integer.

\textbf{OUTPUT:}

Replace \( v_n \) with \( v_{n,k} \) for all generators \( v_n \) occurring in self.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: a.stretch(2)
x_2 + x_0
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: a = x[0] + x[1] + y[0]*y[1]; a
x_1 + x_0 + y_1*y_0
sage: a.stretch(2)
x_2 + x_0 + y_2*y_0
\end{verbatim}

\textbf{symmetric_cancellation_order}(other)
Comparison of leading terms by Symmetric Cancellation Order, \(<_{sc}\).

\textbf{INPUT:}

self, other – two Infinite Polynomials

\textbf{ASSUMPTION:}
Both Infinite Polynomials are non-zero.

\textbf{OUTPUT:}

\((c, \sigma, w)\), where

\begin{itemize}
  \item \( c = -1,0,1 \), or None if the leading monomial of self is smaller, equal, greater, or incomparable with respect to other in the monomial ordering of the Infinite Polynomial Ring
  \item \( \sigma \) is a permutation witnessing self \(<_{sc}\) other (resp. self \(>_{sc}\) other) or is 1 if self.\texttt{lm()} == other.\texttt{lm()}
  \item \( w \) is 1 or is a term so that \( w*\texttt{self.lt()}^\sigma == \texttt{other.lt()} \) if \( c \leq 0 \), and \( w*\texttt{other.lt()}^\sigma == \texttt{self.lt()} \) if \( c = 1 \)
\end{itemize}

\textbf{THEORY:}
If the Symmetric Cancellation Order is a well-quasi-ordering then computation of Groebner bases always terminates. This is the case, e.g., if the monomial order is lexicographic. For that reason, lexicographic order is our default order.

\textbf{EXAMPLES:}
tail()

The tail of self (this is self minus its leading term).

EXAMPLES:

```python
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: p.tail()
2*x_10*y_30
```

variables()

Return the variables occurring in self (tuple of elements of some polynomial ring).

EXAMPLES:

```python
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: p.variables()
(x_3, x_2, x_1)
sage: x[1].variables()
(x_1,)
sage: X(1).variables()
()
```

6.3 Symmetric Ideals of Infinite Polynomial Rings

This module provides an implementation of ideals of polynomial rings in a countably infinite number of variables that are invariant under variable permutation. Such ideals are called ‘Symmetric Ideals’ in the rest of this document. Our implementation is based on the theory of M. Aschenbrenner and C. Hillar.

AUTHORS:

- Simon King <simon.king@nuigalway.ie>

EXAMPLES:

Here, we demonstrate that working in quotient rings of Infinite Polynomial Rings works, provided that one uses symmetric Groebner bases.

```python
sage: R.<x> = InfinitePolynomialRing(QQ)
sage: I = R.ideal([x[1]*x[2] + x[3]])
```

Note that I is not a symmetric Groebner basis:
\begin{verbatim}
 sage: G = R*I.groebner_basis()
sage: G
Symmetric Ideal (x_1^2 + x_1, x_2 - x_1) of Infinite polynomial ring in x over Rational → Field
 sage: Q = R.quotient(G)
 sage: Q(p)
-2*x_1 + 3

By the second generator of G, variable \(x_n\) is equal to \(x_1\) for any positive integer \(n\). By the first generator of G, \(x_1^3\) is equal to \(x_1\) in Q. Indeed, we have
\end{verbatim}

\begin{verbatim}
 sage: Q(p)*x[2] == Q(p)*x[1]*x[3]*x[5]
True
\end{verbatim}

class sage.rings.polynomial.symmetric_ideal.SymmetricIdeal

    Bases: sage.rings.ideal.Ideal_generic

    Ideal in an Infinite Polynomial Ring, invariant under permutation of variable indices

    THEORY:

    An Infinite Polynomial Ring with finitely many generators \(x_*, y_*\), ... over a field \(F\) is a free commutative \(F\)-algebra generated by infinitely many ‘variables’ \(x_0, x_1, x_2, ..., y_0, y_1, y_2, ...\). We refer to the natural number \(n\) as the index of the variable \(x_n\). See more detailed description at \textit{infinite_polynomial_ring}

    Infinite Polynomial Rings are equipped with a permutation action by permuting positive variable indices, i.e., \(x^n_P = x_P(n), y^n_P = y_P(n), \) for any permutation \(P\). Note that the variables \(x_0, y_0, ...\) of index zero are invariant under that action.

    A Symmetric Ideal is an ideal in an infinite polynomial ring \(X\) that is invariant under the permutation action. In other words, if \(S_\infty\) denotes the symmetric group of 1, 2, ..., then a Symmetric Ideal is a right \(X[S_\infty]\)-submodule of \(X\).

    It is known by work of Aschenbrenner and Hillar [AB2007] that an Infinite Polynomial Ring \(X\) with a single generator \(x_*\) is Noetherian, in the sense that any Symmetric Ideal \(I \subset X\) is finitely generated modulo addition, multiplication by elements of \(X\), and permutation of variable indices (hence, it is a finitely generated right \(X[S_\infty]\)-module).

    Moreover, if \(X\) is equipped with a lexicographic monomial ordering with \(x_1 < x_2 < x_3...\) then there is an algorithm of Buchberger type that computes a Groebner basis \(G\) for \(I\) that allows for computation of a unique normal form, that is zero precisely for the elements of \(I\) – see [AB2008]. See \textit{groebner_basis()} for more details.

    Our implementation allows more than one generator and also provides degree lexicographic and degree reverse lexicographic monomial orderings – we do, however, not guarantee termination of the Buchberger algorithm in these cases.

    EXAMPLES:
\end{verbatim}

\begin{verbatim}
 sage: X.<x,y> = InfinitePolynomialRing(QQ)
 sage: I == loads(dumps(I))
True
 sage: latex(I)
\left(x_{\ast\ast\ast-\{\ast\\ast\} \ y_{\ast\ast\ast-\{\ast\\ast\} \ y_{\ast\ast\ast-\{\ast\\ast\} \ y_{\ast\ast\ast-\{\ast\\ast\}}\right)\Bold{Q}\{x_\ast, y_\ast\][\mathfrak{n\ right}]
\end{verbatim}
The default ordering is lexicographic. We now compute a Groebner basis:

```
sage: J = I.groebner_basis() ; J  # about 3 seconds
[x_1*y_2*y_1 + 2*x_1*y_2, x_2*y_2*y_1 + 2*x_2*y_1, x_2*x_1*y_1^2 + 2*x_2*x_1*y_1, x_→2*x_1*y_2 - x_2*x_1*y_1]
```

Note that even though the symmetric ideal can be generated by a single polynomial, its reduced symmetric Groebner basis comprises four elements. Ideal membership in $I$ can now be tested by commuting symmetric reduction modulo $J$:

```
sage: I.reduce(J)
Symmetric Ideal (0) of Infinite polynomial ring in x, y over Rational Field
```

The Groebner basis is not point-wise invariant under permutation:

```
sage: P=Permutation([2, 1])
sage: J[2]
x_2*x_1*y_1^2 + 2*x_2*x_1*y_1
sage: J[2]^P
x_2*x_1*y_2^2 + 2*x_2*x_1*y_2
sage: J[2]^P in J
False
```

However, any element of $J$ has symmetric reduction zero even after applying a permutation. This even holds when the permutations involve higher variable indices than the ones occurring in $J$:

```
sage: [[(p^P).reduce(J) for p in J] for P in Permutations(3)]
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

Since $I$ is not a Groebner basis, it is no surprise that it cannot detect ideal membership:

```
sage: [p.reduce(I) for p in J]
[0, x_2*y_2*y_1 + 2*x_2*y_1, x_2*x_1*y_1^2 + 2*x_2*x_1*y_1, x_2*x_1*y_2 - x_2*x_1*y_→1]
```

Note that we give no guarantee that the computation of a symmetric Groebner basis will terminate in any order different from lexicographic.

When multiplying Symmetric Ideals or raising them to some integer power, the permutation action is taken into account, so that the product is indeed the product of ideals in the mathematical sense.

```
sage: I=X*(x[1])
sage: I*I
Symmetric Ideal (x_1^2, x_2*x_1) of Infinite polynomial ring in x, y over Rational Field
sage: I^3
Symmetric Ideal (x_1^3, x_2*x_1^2, x_2^2*x_1, x_3*x_2*x_1) of Infinite polynomial ring in x, y over Rational Field
sage: I*I == X*(x[1]^2)
False
```

```
groebner_basis(tailreduce=False, reduced=True, algorithm=None, report=None, use_full_group=False)
    Return a symmetric Groebner basis (type Sequence) of self.

    INPUT:
    • tailreduce – (bool, default False) If True, use tail reduction in intermediate computations
```
• **reduced** – (bool, default True) If True, return the reduced normalised symmetric Groebner basis.
• **algorithm** – (string, default None) Determine the algorithm (see below for available algorithms).
• **report** – (object, default None) If not None, print information on the progress of computation.
• **use_full_group** – (bool, default False) If True then proceed as originally suggested by [AB2008].
  Our default method should be faster; see `symmetrisation()` for more details.

The computation of symmetric Groebner bases also involves the computation of classical Groebner bases, i.e., of Groebner bases for ideals in polynomial rings with finitely many variables. For these computations, Sage provides the following **ALGORITHMS**:

- **autoselect** (default)
- **`singular:groebner`**  Singular’s `groebner` command
- **`singular:std`**  Singular’s `std` command
- **`singular:stdhilb`**  Singular’s `stdhilb` command
- **`singular:stdfglm`**  Singular’s `stdfglm` command
- **`singular:slimgb`**  Singular’s `slimgb` command
- **`libsingular:std`**  libSingular’s `std` command
- **`libsingular:slimgb`**  libSingular’s `slimgb` command
- **`toy:buchberger`**  Sage’s toy/educational `buchberger` without strategy
- **`toy:buchberger2`**  Sage’s toy/educational `buchberger` with strategy
- **`toy:d_basis`**  Sage’s toy/educational `d_basis` algorithm
- **`macaulay2:gb`**  Macaulay2’s `gb` command (if available)
- **`magma:GroebnerBasis`**  Magma’s `GroebnerBasis` command (if available)

If only a system is given - e.g. ‘magma’ - the default algorithm is chosen for that system.

**Note:** The Singular and libSingular versions of the respective algorithms are identical, but the former calls an external Singular process while the later calls a C function, i.e. the calling overhead is smaller.

**EXAMPLES:**

```python
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: I1 = X*(x[1]+x[2],x[1]*x[2])
sage: I1.groebner_basis()
[x_1]
sage: I2.groebner_basis()
[x_1*y_2 + y_2^2*y_1, x_2*y_1 + y_2*y_1^2]
```

Note that a symmetric Groebner basis of a principal ideal is not necessarily formed by a single polynomial.

When using the algorithm originally suggested by Aschenbrenner and Hillar, the result is the same, but the computation takes much longer:

```python
sage: I2.groebner_basis(use_full_group=True)
[x_1*y_2 + y_2^2*y_1, x_2*y_1 + y_2*y_1^2]
```

Last, we demonstrate how the report on the progress of computations looks like:
\begin{verbatim}
sage: I1.groebner_basis(report=True, reduced=True)
Symmetric interreduction
[1/2] >
[2/2] :>
[1/2] >
[2/2] >
Symmetrise 2 polynomials at level 2
Apply permutations
>
>
Symmetric interreduction
[1/3] >
[2/3] >
[3/3] :>
-> 0
[1/2] >
[2/2] >
Symmetrisation done
Classical Groebner basis
-> 2 generators
Symmetric interreduction
[1/2] >
[2/2] >
Symmetrise 2 polynomials at level 3
Apply permutations
>
>
::>
>
::>
Symmetric interreduction
[1/4] >
[2/4] :>
-> 0
[3/4] :>
-> 0
[4/4] :
-> 0
[1/1] >
Apply permutations
:
:
:
Symmetric interreduction
[1/1] >
Classical Groebner basis
-> 1 generators
Symmetric interreduction
[1/1] >
Symmetrise 1 polynomials at level 4
Apply permutations
>
\end{verbatim}
(continues on next page)
The Aschenbrenner-Hillar algorithm is only guaranteed to work if the base ring is a field. So, we raise a TypeError if this is not the case:

```
sage: R.<x,y> = InfinitePolynomialRing(ZZ)
sage: I = R*[x[1]+x[2],y[1]]
sage: I.groebner_basis()
Traceback (most recent call last):
...  
TypeError: The base ring (= Integer Ring) must be a field
```

**interreduced_basis()**

A fully symmetrically reduced generating set (type `Sequence`) of self.

This does essentially the same as `interreduction()` with the option ‘tailreduce’, but it returns a `Sequence` rather than a `SymmetricIdeal`.

**EXAMPLES:**

```
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: I=X*(x[1]+x[2],x[1]*x[2])
sage: I.interreduced_basis()
[-x_1^2, x_2 + x_1]
```

**interreduction(tailreduce=True, sorted=False, report=None, RStrat=None)**

Return symmetrically interreduced form of self

**INPUT:**

- `tailreduce` – (bool, default `True`) If `True`, the interreduction is also performed on the non-leading monomials.
- `sorted` – (bool, default `False`) If `True`, it is assumed that the generators of self are already increasingly sorted.
- `report` – (object, default `None`) If not `None`, some information on the progress of computation is printed.
- `RStrat` – (`SymmetricReductionStrategy`, default `None`) A reduction strategy to which the polynomials resulting from the interreduction will be added. If `RStrat` already contains some polynomials, they will be used in the interreduction. The effect is to compute in a quotient ring.

**OUTPUT:**
A Symmetric Ideal $J$ (sorted list of generators) coinciding with self as an ideal, so that any generator is symmetrically reduced w.r.t. the other generators. Note that the leading coefficients of the result are not necessarily 1.

**EXAMPLES:**

```python
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: I=X*(x[1]+x[2],x[1]*x[2])
sage: I.interreduction()
Symmetric Ideal (-x_1^2, x_2 + x_1) of Infinite polynomial ring in x over Rational Field
```

Here, we show the `report` option:

```python
sage: I.interreduction(report=True)
Symmetric interreduction
[1/2] >
[2/2] :>
[1/2] >
>
Symmetric Ideal (-x_1^2, x_2 + x_1) of Infinite polynomial ring in x over Rational Field
```

[$m/n$] indicates that polynomial number $m$ is considered and the total number of polynomials under consideration is $n$. ‘$-> 0’ is printed if a zero reduction occurred. The rest of the report is as described in `sage.rings.polynomial.symmetric_reduction.SymmetricReductionStrategy.reduce()`.

Last, we demonstrate the use of the optional parameter $\text{RStrat}$:

```python
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: R = SymmetricReductionStrategy(X)
sage: R
Symmetric Reduction Strategy in Infinite polynomial ring in x over Rational Field
sage: I.interreduction(RStrat=R)
Symmetric Ideal (-x_1^2, x_2 + x_1) of Infinite polynomial ring in x over Rational Field
sage: R
Symmetric Reduction Strategy in Infinite polynomial ring in x over Rational Field, modulo $x_1^2, x_2 + x_1$
sage: R = SymmetricReductionStrategy(X,[x[1]^2])
sage: I.interreduction(RStrat=R)
Symmetric Ideal (x_2 + x_1) of Infinite polynomial ring in x over Rational Field
```

`is_maximal()`

Answers whether self is a maximal ideal.

**ASSUMPTION:**

self is defined by a symmetric Groebner basis.

**NOTE:**
It is not checked whether self is in fact a symmetric Groebner basis. A wrong answer can result if this assumption does not hold. A :exc:`NotImplementedError` is raised if the base ring is not a field, since symmetric Groebner bases are not implemented in this setting.

EXAMPLES:

```sage
sage: R.<x,y> = InfinitePolynomialRing(QQ)
sage: I = R.ideal([x[1]+y[2], x[2]-y[1]])
sage: I = R*I.groebner_basis()
sage: I
Symmetric Ideal (y_1, x_1) of Infinite polynomial ring in x, y over Rational Field

sage: I = R.ideal([x[1]+y[2], x[2]-y[1]])
sage: I.is_maximal()
False
```

The preceding answer is wrong, since it is not the case that $I$ is given by a symmetric Groebner basis:

```sage
sage: I = R*I.groebner_basis()
sage: I
Symmetric Ideal (y_1, x_1) of Infinite polynomial ring in x, y over Rational Field

sage: I.is_maximal()
True
```

.. automethod:: normalisation

```
normalisation()

Return an ideal that coincides with self, so that all generators have leading coefficient 1.
Possibly occurring zeroes are removed from the generator list.

EXAMPLES:

```sage
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: I = X*(1/2*x[1]+2/3*x[2], 0, 4/5*x[1]*x[2])
sage: I.normalisation()
Symmetric Ideal (x_2 + 3/4*x_1, x_2*x_1) of Infinite polynomial ring in x over Rational Field
```

.. automethod:: reduce

```
reduce(I, tailreduce=False)

Symmetric reduction of self by another Symmetric Ideal or list of Infinite Polynomials, or symmetric reduction of a given Infinite Polynomial by self.

INPUT:

* I – an Infinite Polynomial, or a Symmetric Ideal or a list of Infinite Polynomials.

* tailreduce – (bool, default False) If True, the non-leading terms will be reduced as well.

OUTPUT:

Symmetric reduction of self with respect to I.

THEORY:

Reduction of an element $p$ of an Infinite Polynomial Ring $X$ by some other element $q$ means the following:

1. Let $M$ and $N$ be the leading terms of $p$ and $q$.

2. Test whether there is a permutation $P$ that does not does not diminish the variable indices occurring in $N$ and preserves their order, so that there is some term $T \in X$ with $TN^P = M$. If there is no such permutation, return $p$
3. Replace $p$ by $p - Tq^P$ and continue with step 1.

EXAMPLES:

```sage
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: I.reduce([x[1]^2*y[2]])
Symmetric Ideal (x_3^2*y_1 + y_3*y_1^2) of Infinite polynomial ring in x, y
  → over Rational Field
```

The preceding is correct, since any permutation that turns $x[1]^2*y[2]$ into a factor of $x[3]^2*y[2]$ interchanges the variable indices 1 and 2 – which is not allowed. However, reduction by $x[2]^2*y[1]$ works, since one can change variable index 1 into 2 and 2 into 3:

```sage
sage: I.reduce([x[2]^2*y[1]])
Symmetric Ideal (y_3*y_1^2) of Infinite polynomial ring in x, y over Rational Field
```

The next example shows that tail reduction is not done, unless it is explicitly advised. The input can also be a symmetric ideal:

```sage
sage: J = (y[2])*X
sage: I.reduce(J)
Symmetric Ideal (x_3^2*y_1 + y_3*y_1^2) of Infinite polynomial ring in x, y over Rational Field
sage: I.reduce(J, tailreduce=True)
Symmetric Ideal (x_3^2*y_1) of Infinite polynomial ring in x, y over Rational Field
```

`squeezed()`
Reduce the variable indices occurring in `self`.

OUTPUT:
A Symmetric Ideal whose generators are the result of applying `squeezed()` to the generators of `self`.

NOTE:
The output describes the same Symmetric Ideal as `self`.

EXAMPLES:

```sage
sage: X.<x,y> = InfinitePolynomialRing(QQ, implementation='sparse')
sage: I = X*(x[1000]^y[100], x[50]*y[1000])
sage: I.squeezed()
Symmetric Ideal (x_2^y_1, x_1^y_2) of Infinite polynomial ring in x, y over Rational Field
```

`symmetric_basis()`
A symmetrised generating set (type `Sequence`) of `self`.

This does essentially the same as `symmetrisation()` with the option `tailreduce`, and it returns a `Sequence` rather than a `SymmetricIdeal`.

EXAMPLES:

```sage
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: I = X*(x[1]+x[2], x[1]*x[2])
```

(continues on next page)
sage: I.symmetric_basis()
[x_1^2, x_2 + x_1]

\texttt{symmetrisation}(N=None, \texttt{tailreduce}=False, \texttt{report}=None, \texttt{use_full_group}=False)

Apply permutations to the generators of self and interreduce

\textbf{INPUT:}

- \texttt{N} – (integer, default \texttt{None}) Apply permutations in \textit{Sym}(N). If it is not given then it will be replaced by the maximal variable index occurring in the generators of \textit{self.interreduction().squeezed()}.  
- \texttt{tailreduce} – (bool, default \texttt{False}) If \texttt{True}, perform tail reductions.  
- \texttt{report} – (object, default \texttt{None}) If not \texttt{None}, report on the progress of computations.  
- \texttt{use_full_group} (optional) – If \texttt{True}, apply all elements of \textit{Sym}(N) to the generators of \textit{self.squeezed()}, interreduce, and repeat until the result stabilises, which is often much faster than applying all of \textit{Sym}(N), and we are convinced that both methods yield the same result.

\textbf{OUTPUT:}

A symmetrically interreduced symmetric ideal with respect to which any \textit{Sym}(N)-translate of a generator of self is symmetrically reducible, where by default \texttt{N} is the maximal variable index that occurs in the generators of \textit{self.interreduction().squeezed()}.  

\textbf{NOTE:}

If \textit{I} is a symmetric ideal whose generators are monomials, then \textit{I.symmetrisation()} is its reduced Groebner basis. It should be noted that without symmetrisation, monomial generators, in general, do not form a Groebner basis.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: I = X*(x[1]+x[2], x[1]*x[2])
sage: I.symmetrisation()
Symmetric Ideal (-x_1^2, x_2 + x_1) of Infinite polynomial ring in x over \texttt{Rational Field}
sage: I.symmetrisation(N=3)
Symmetric Ideal (-2*x_1) of Infinite polynomial ring in x over Rational Field
sage: I.symmetrisation(N=3, use_full_group=True)
Symmetric Ideal (-2*x_1) of Infinite polynomial ring in x over Rational Field
\end{verbatim}

\section*{6.4 Symmetric Reduction of Infinite Polynomials}

\texttt{SymmetricReductionStrategy} provides a framework for efficient symmetric reduction of Infinite Polynomials, see \texttt{infinite_polynomial_element}.

\textbf{AUTHORS:}

- Simon King <simon.king@nuigalway.ie>

\textbf{THEORY:}

According to M. Aschenbrenner and C. Hillar [AB2007], Symmetric Reduction of an element \textit{p} of an Infinite Polynomial Ring \textit{X} by some other element \textit{q} means the following:
1. Let $M$ and $N$ be the leading terms of $p$ and $q$.
2. Test whether there is a permutation $P$ that does not diminish the variable indices occurring in $N$ and preserves their order, so that there is some term $T \in X$ with $TN^P = M$. If there is no such permutation, return $p$.
3. Replace $p$ by $p - Tq^P$ and continue with step 1.

When reducing one polynomial $p$ with respect to a list $L$ of other polynomials, there usually is a choice of order on which the efficiency crucially depends. Also it helps to modify the polynomials on the list in order to simplify the basic reduction steps.

The preparation of $L$ may be expensive. Hence, if the same list is used many times then it is reasonable to perform the preparation only once. This is the background of `SymmetricReductionStrategy`.

Our current strategy is to keep the number of terms in the polynomials as small as possible. For this, we sort $L$ by increasing number of terms. If several elements of $L$ allow for a reduction of $p$, we choose the one with the smallest number of terms. Later on, it should be possible to implement further strategies for choice.

When adding a new polynomial $q$ to $L$, we first reduce $q$ with respect to $L$. Then, we test heuristically whether it is possible to reduce the number of terms of the elements of $L$ by reduction modulo $q$. That way, we see best chances to keep the number of terms in intermediate reduction steps relatively small.

**EXAMPLES:**

First, we create an infinite polynomial ring and one of its elements:

```python
sage: X.<x,y> = InfinitePolynomialRing(QQ)
```

We want to symmetrically reduce it by another polynomial. So, we put this other polynomial into a list and create a Symmetric Reduction Strategy object:

```python
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
dsage: S = SymmetricReductionStrategy(X, [y[2]^2*x[1]])
sage: S
```

```python
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field, → modulo x_1*y_2^2
```

```python
sage: S.reduce(p)
x_3*y_1^2 + y_3*y_1
```

The preceding is correct, since any permutation that turns $y[2]^2*x[1]$ into a factor of $y[1]^2*x[3]$ interchanges the variable indices 1 and 2 – which is not allowed in a symmetric reduction. However, reduction by $y[1]^2*x[2]$ works, since one can change variable index 1 into 2 and 2 into 3. So, we add this to $S$:

```python
sage: S.add_generator(y[1]^2*x[2])
sage: S
```

```python
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field, → modulo x_2*y_1^2, x_1*y_2^2
```

```python
sage: S.reduce(p)
y_3*y_1
```

The next example shows that tail reduction is not done, unless it is explicitly advised:

```python
x_3 + 2*x_2*y_1^2 + 3*x_1*y_2^2
```

(continues on next page)
However, it is possible to ask for tailreduction already when the Symmetric Reduction Strategy is created:

```python
sage: S2
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field, modulo
x_2*y_1^2,
with tailreduction
x_3
```

### class sage.rings.polynomial.symmetric_reduction.SymmetricReductionStrategy

A framework for efficient symmetric reduction of InfinitePolynomial, see `infinite_polynomial_element`.

**INPUT:**

- **Parent** – an Infinite Polynomial Ring, see `infinite_polynomial_element`.
- **L** – (list, default the empty list) List of elements of `Parent` with respect to which will be reduced.
- **good_input** – (bool, default `None`) If this optional parameter is true, it is assumed that each element of `L` is symmetrically reduced with respect to the previous elements of `L`.

**EXAMPLES:**

```python
sage: X.<y> = InfinitePolynomialRing(QQ)
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
y_3 + 3*y_2^2*y_1 + 2*y_2*y_1^2
y_3
```

#### add_generator(p, good_input=None)

Add another polynomial to `self`.

**INPUT:**

- **p** – An element of the underlying infinite polynomial ring.
- **good_input** – (bool, default `None`) If `True`, it is assumed that `p` is reduced with respect to `self`. Otherwise, this reduction will be done first (which may cost some time).

**Note:** Previously added polynomials may be modified. All input is prepared in view of an efficient symmetric reduction.

**EXAMPLES:**
```python
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: S = SymmetricReductionStrategy(X)
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field, modulo x_3*y_1 + x_1*y_1 + y_3
```

Note that the first added polynomial will be simplified when adding a suitable second polynomial:

```python
sage: S.add_generator(x[2]+x[1])
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field, modulo y_3, x_2 + x_1
```

By default, reduction is applied to any newly added polynomial. This can be avoided by specifying the optional parameter 'good_input':

```python
sage: S.add_generator(y[2]+y[1]*x[2])
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field, modulo y_3, x_1*y_1 - y_2, x_2 + x_1
```

```python
sage: S.reduce(x[3]+x[2])
-2*x_1
```

```python
sage: S.add_generator(x[3]+x[2], good_input=True)
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field, modulo y_3, x_3 + x_2, x_1*y_1 - y_2, x_2 + x_1
```

In the previous example, `x[3] + x[2]` is added without being reduced to zero.

### gens()

Return the list of Infinite Polynomials modulo which self reduces.

#### EXAMPLES:

```python
sage: X.<y> = InfinitePolynomialRing(QQ)
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
```
Symmetric Reduction Strategy in Infinite polynomial ring in y over Rational Field, modulo y_2*y_1^2, y_2^2*y_1

sage: S.gens()
[y_2*y_1^2, y_2^2*y_1]

reduce(p, notail=False, report=None)
Symmetric reduction of an infinite polynomial.

INPUT:
- p – an element of the underlying infinite polynomial ring.
- notail – (bool, default False) If True, tail reduction is avoided (but there is no guarantee that there will be no tail reduction at all).
- report – (object, default None) If not None, print information on the progress of the computation.

OUTPUT:
Reduction of p with respect to self.

Note: If tail reduction shall be forced, use tailreduce().

EXAMPLES:

sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: S = SymmetricReductionStrategy(X, [y[3]], tailreduce=True)
sage: S.reduce(y[4]*x[1] + y[1]*x[4])
x_4*y_1
sage: S.reduce(y[4]*x[1] + y[1]*x[4], notail=True)
x_4*y_1 + x_1*y_4

Last, we demonstrate the ‘report’ option:

::>
x_1*y_1 + y_4 - y_3*y_1 - y_1

Each ‘:’ indicates that one reduction of the leading monomial was performed. Eventually, the ‘>’ indicates that the computation is finished.

reset()
Remove all polynomials from self.
EXAMPLES:

```
sage: X.<y> = InfinitePolynomialRing(QQ)
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in y over Rational Field, modulo y_2*y_1^2, y_2^2*y_1
sage: S.reset()
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in y over Rational Field
```

```
def setgens(L):
    Define the list of Infinite Polynomials modulo which self reduces.

    INPUT:

    L – a list of elements of the underlying infinite polynomial ring.

    Note: It is not tested if L is a good input. That method simply assigns a copy of L to the generators of self.
```

```
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: X.<y> = InfinitePolynomialRing(QQ)
sage: R = SymmetricReductionStrategy(X)
sage: R.setgens(S.gens())
sage: R
Symmetric Reduction Strategy in Infinite polynomial ring in y over Rational Field, modulo y_2*y_1^2, y_2^2*y_1
sage: R.gens() is S.gens()
False
sage: R.gens() == S.gens()
True
```

```
def tailreduce(p, report=None)
    Symmetric reduction of an infinite polynomial, with forced tail reduction.

    INPUT:

    • p – an element of the underlying infinite polynomial ring.
      • report – (object, default None) If not None, print information on the progress of the computation.

    OUTPUT:

    Reduction (including the non-leading elements) of p with respect to self.
```

```
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: X.<y> = InfinitePolynomialRing(QQ)
sage: R = SymmetricReductionStrategy(X)
sage: R.setgens(S.gens())
sage: R
Symmetric Reduction Strategy in Infinite polynomial ring in y over Rational Field, modulo y_2*y_1^2, y_2^2*y_1
sage: R.gens() is S.gens()
False
sage: R.gens() == S.gens()
True
```

6.4. Symmetric Reduction of Infinite Polynomials
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: S = SymmetricReductionStrategy(X, [y[3]])
sage: S.reduce(y[4]*x[1] + y[1]*x[4])
x_4*y_1 + x_1*y_4
sage: S.tailreduce(y[4]*x[1] + y[1]*x[4])
x_4*y_1

Last, we demonstrate the ‘report’ option:

sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field, modulo y_3 + y_2, x_2 + y_1, x_1*y_2 + y_4 + y_1^2
T[3]:>
T[3]:>
x_1*y_1 - y_2 + y_1^2 - y_1

The protocol means the following:

- ‘T[3]’ means that we currently do tail reduction for a polynomial with three terms.
- ‘:::>’ means that there were three reductions of leading terms.
- The tail of the result of the preceding reduction still has three terms. One reduction of leading terms was possible, and then the final result was obtained.
7.1 Boolean Polynomials

Elements of the quotient ring

\[ \mathbb{F}_2[x_1, \ldots, x_n]/ \langle x_1^2 + x_1, \ldots, x_n^2 + x_n \rangle. \]

are called boolean polynomials. Boolean polynomials arise naturally in cryptography, coding theory, formal logic, chip design and other areas. This implementation is a thin wrapper around the PolyBoRi library by Michael Brickenstein and Alexander Dreyer.

“Boolean polynomials can be modelled in a rather simple way, with both coefficients and degree per variable lying in \{0, 1\}. The ring of Boolean polynomials is, however, not a polynomial ring, but rather the quotient ring of the polynomial ring over the field with two elements modulo the field equations \(x^2 = x\) for each variable \(x\). Therefore, the usual polynomial data structures seem not to be appropriate for fast Groebner basis computations. We introduce a specialised data structure for Boolean polynomials based on zero-suppressed binary decision diagrams (ZDDs), which is capable of handling these polynomials more efficiently with respect to memory consumption and also computational speed. Furthermore, we concentrate on high-level algorithmic aspects, taking into account the new data structures as well as structural properties of Boolean polynomials.” - [BD2007]

For details on the internal representation of polynomials see

http://polybori.sourceforge.net/zdd.html

AUTHORS:

• Michael Brickenstein: PolyBoRi author
• Alexander Dreyer: PolyBoRi author
• Burcin Erocal <burcin@erocal.org>: main Sage wrapper author
• Martin Albrecht <malb@informatik.uni-bremen.de>: some contributions to the Sage wrapper
• Simon King <simon.king@uni-jena.de>: Adopt the new coercion model. Fix conversion from univariate polynomial rings. Pickling of BooleanMonomialMonoid (via UniqueRepresentation) and BooleanMonomial.
• Charles Bouillaguet <charles.bouillaguet@gmail.com>: minor changes to improve compatibility with MPolynomial and make the variety() function work on ideals of BooleanPolynomial’s.

EXAMPLES:

Consider the ideal

\[ \langle ab + cd + 1, ace + de, abc + ce, bc + cde + 1 \rangle. \]

First, we compute the lexicographical Groebner basis in the polynomial ring

\[ R = \mathbb{F}_2[a, b, c, d, e]. \]
If one wants to solve this system over the algebraic closure of $F_2$ then this Groebner basis was the one to consider. If one wants solutions over $F_2$ only then one adds the field polynomials to the ideal to force the solutions in $F_2$.

We can express the restriction to $F_2$ by considering the quotient ring. If $I$ is an ideal in $F[x_1, ..., x_n]$ then the ideals in the quotient ring $F[x_0, ..., x_n]/I$ are in one-to-one correspondence with the ideals of $F[x_0, ..., x_n]$ containing $I$ (that is, the ideals $J$ satisfying $I \subseteq J \subseteq P$).

This quotient ring is exactly what PolyBoRi handles well:

Note that $d^2 + d$ is not representable in $B = Q$. Also note, that PolyBoRi cannot play out its strength in such small examples, i.e. working in the polynomial ring might be faster for small examples like this.
7.1.1 Implementation specific notes

PolyBoRi comes with a Python wrapper. However this wrapper does not match Sage’s style and is written using Boost. Thus Sage’s wrapper is a reimplementation of Python bindings to PolyBoRi’s C++ library. This interface is written in Cython like all of Sage’s C/C++ library interfaces. An interface in PolyBoRi style is also provided which is effectively a reimplementation of the official Boost wrapper in Cython. This means that some functionality of the official wrapper might be missing from this wrapper and this wrapper might have bugs not present in the official Python interface.

7.1.2 Access to the original PolyBoRi interface

The re-implementation PolyBoRi’s native wrapper is available to the user too:

```python
sage: from sage.rings.polynomial.pbori import *
sage: declare_ring([Block('x',2), Block('y',3)], globals())
Boolean PolynomialRing in x0, x1, y0, y1, y2
sage: r
Boolean PolynomialRing in x0, x1, y0, y1, y2

sage: [Variable(i, r) for i in range(r.ngens())]
[x(0), x(1), y(0), y(1), y(2)]
```

For details on this interface see:


Also, the interface provides functions for compatibility with Sage accepting convenient Sage data types which are slower than their native PolyBoRi counterparts. For instance, sets of points can be represented as tuples of tuples (Sage) or as `BooleSet` (PolyBoRi) and naturally the second option is faster.

```python
class sage.rings.polynomial.pbori.pbori.BooleConstant
    Bases: object

    Construct a boolean constant (modulo 2) from integer value:

    INPUT:
    • i - an integer

    EXAMPLES:

    sage: from sage.rings.polynomial.pbori.pbori import BooleConstant
    sage: [BooleConstant(i) for i in range(5)]
    [0, 1, 0, 1, 0]

    deg()
    Get degree of boolean constant.

    EXAMPLES:

    sage: from sage.rings.polynomial.pbori.pbori import BooleConstant
    sage: BooleConstant(0).deg()
    -1
    sage: BooleConstant(1).deg()
    0

    has_constant_part()
    This is true for `BooleConstant(1)`.
```
EXAMPLES:

```python
sage: from sage.rings.polynomial.pbori.pbori import BooleConstant
sage: BooleConstant(1).has_constant_part()
True
sage: BooleConstant(0).has_constant_part()
False
```

`is_constant()`

This is always true for in this case.

EXAMPLES:

```python
sage: from sage.rings.polynomial.pbori.pbori import BooleConstant
sage: BooleConstant(1).is_constant()
True
sage: BooleConstant(0).is_constant()
True
```

`is_one()`

Check whether boolean constant is one.

EXAMPLES:

```python
sage: from sage.rings.polynomial.pbori.pbori import BooleConstant
sage: BooleConstant(0).is_one()
False
sage: BooleConstant(1).is_one()
True
```

`is_zero()`

Check whether boolean constant is zero.

EXAMPLES:

```python
sage: from sage.rings.polynomial.pbori.pbori import BooleConstant
sage: BooleConstant(1).is_zero()
False
sage: BooleConstant(0).is_zero()
True
```

`variables()`

Get variables (return always and empty tuple).

EXAMPLES:

```python
sage: from sage.rings.polynomial.pbori.pbori import BooleConstant
sage: BooleConstant(0).variables()
()
sage: BooleConstant(1).variables()
()
```

```python
class sage.rings.polynomial.pbori.pbori.BooleSet
    Bases: object
    
    Return a new set of boolean monomials. This data type is also implemented on the top of ZDDs and allows to see polynomials from a different angle. Also, it makes high-level set operations possible, which are in most cases
```
faster than operations handling individual terms, because the complexity of the algorithms depends only on the structure of the diagrams.

Objects of type \texttt{BooleanPolynomial} can easily be converted to the type \texttt{BooleSet} by using the member function \texttt{BooleanPolynomial.set()}. 

INPUT:

- \texttt{param} - either a \texttt{CCuddNavigator}, a \texttt{BooleSet} or \texttt{None}.
- \texttt{ring} - a boolean polynomial ring.

EXAMPLES:

```sage
sage: from sage.rings.polynomial.pbori.pbori import BooleSet
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: BS = BooleSet(a.set())
sage: BS
{{a}}
sage: BS = BooleSet((a*b + c + 1).set())
sage: BS
{{a,b}, {c}, {}}
sage: from sage.rings.polynomial.pbori.pbori import *

```

\textbf{Note:} \texttt{BooleSet} prints as \{\} but are not Python dictionaries.

\textbf{cartesian_product}(\textit{rhs})

Return the Cartesian product of this set and the set \textit{rhs}.

The Cartesian product of two sets \textit{X} and \textit{Y} is the set of all possible ordered pairs whose first component is a member of \textit{X} and whose second component is a member of \textit{Y}.

\[ X \times Y = \{(x, y)|x \in X \text{ and } y \in Y\}. \]

EXAMPLES:

```sage
sage: B = BooleanPolynomialRing(5, 'x')
sage: x0,x1,x2,x3,x4 = B.gens()
sage: f = x1*x2+x2*x3
sage: s = f.set(); s
{{x1,x2}, {x2,x3}}
sage: g = x4 + 1
sage: t = g.set(); t
{{x4}, {}}
sage: s.cartesian_product(t)
{{x1,x2,x4}, {x1,x2}, {x2,x3,x4}, {x2,x3}}
```

\textbf{change}(\textit{ind})

Swaps the presence of \texttt{x_i} in each entry of the set.

EXAMPLES:
diff(rhs)
Return the set theoretic difference of this set and the set rhs.

The difference of two sets $X$ and $Y$ is defined as:

$$ X \setminus Y = \{ x \mid x \in X \text{ and } x \notin Y \}.$$  

EXAMPLES:

```
sage: B = BooleanPolynomialRing(5, 'x')
sage: x0,x1,x2,x3,x4 = B.gens()
sage: f = x1*x2 + x2*x3
sage: s = f.set(); s
{{x1, x2}, {x2, x3}}
sage: g = x2*x3 + 1
sage: t = g.set(); t
{{x2, x3}, {}}
sage: s.diff(t)
{{x1, x2}}
```
**empty()**

Return True if this set is empty.

**EXAMPLES:**

```python
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: BS = (a*b + c).set()
sage: BS.empty()
False
sage: BS = B(0).set()
sage: BS.empty()
True
```

**include_divisors()**

Extend this set to include all divisors of the elements already in this set and return the result as a new set.

**EXAMPLES:**

```python
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: f = a^2*d*e + a*f + b*d*e + c*d*e + 1
sage: s = f.set(); s
{{a,d,e}, {a,f}, {b,d,e}, {c,d,e}, {}}
sage: s.include_divisors()
{{a,d,e}, {a,d}, {a,e}, {a,f}, {a}, {b,d,e}, {b,d}, {b,e}, {b}, {c,d,e}, {c,d}, {c,e}, {c}, {d,e}, {d}, {e}, {f}, {}}
```

**intersect(other)**

Return the set theoretic intersection of this set and the set rhs.

The union of two sets $X$ and $Y$ is defined as:

$$X \cap Y = \{x | x \in X \text{ and } x \in Y\}.$$  

**EXAMPLES:**

```python
sage: B = BooleanPolynomialRing(5, 'x')
sage: x0, x1, x2, x3, x4 = B.gens()
sage: f = x1*x2*x3 + x3
sage: s = f.set(); s
{{x1,x2}, {x2,x3}}
sage: g = x2*x3 + 1
sage: t = g.set(); t
{{x2,x3}, {}}
sage: s.intersect(t)
{{x2,x3}}
```
minimal_elements()  
Return a new set containing a divisor of all elements of this set.

EXAMPLES:

```
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: f = a*d*e + a*f + a*b*d*e + a*c*d*e + a
sage: s = f.set(); s
{{a,b,d,e}, {a,c,d,e}, {a,d,e}, {a,f}, {a}}
sage: s.minimal_elements()
{{a}}
```

multiples_of(m)  
Return those members which are multiples of m.

INPUT:
- m - a boolean monomial

EXAMPLES:

```
sage: B = BooleanPolynomialRing(5, 'x')
sage: x0, x1, x2, x3, x4 = B.gens()
sage: f = x1*x2+x2*x3
sage: s = f.set()
sage: s.multiples_of(x1.lm())
{{x1,x2}}
```

n_nodes()  
Return the number of nodes in the ZDD.

EXAMPLES:

```
sage: B = BooleanPolynomialRing(5, 'x')
sage: x0, x1, x2, x3, x4 = B.gens()
sage: f = x1*x2+x2*x3
sage: s = f.set(); s
{{x1,x2}, {x2,x3}}
sage: s.n_nodes()
4
```

navigation()  
Navigators provide an interface to diagram nodes, accessing their index as well as the corresponding then- and else-branches.

You should be very careful and always keep a reference to the original object, when dealing with navigators, as navigators contain only a raw pointer as data. For the same reason, it is necessary to supply the ring as argument, when constructing a set out of a navigator.

EXAMPLES:

```
sage: from sage.rings.polynomial.pbori.pbori import BooleSet
sage: B = BooleanPolynomialRing(5, 'x')
sage: x0, x1, x2, x3, x4 = B.gens()
sage: f = x1*x2+x2*x3*x4+x2*x4+x3+x4+1
sage: s = f.set(); s
{{x1,x2}, {x2,x3,x4}, {x2,x4}, {x3}, {x4}, {}}
```
sage: nav = s.navigation()
sage: BooleSet(nav, s.ring())
\{\{x_1,x_2\}, \{x_2,x_3,x_4\}, \{x_2,x_4\}, \{x_3\}, \{x_4\}, \{\}\}

sage: nav.value()
1

sage: nav_else = nav.else_branch()

sage: BooleSet(nav_else, s.ring())
\{\{x_2,x_3,x_4\}, \{x_2,x_4\}, \{x_3\}, \{x_4\}, \{\}\}

sage: nav_else.value()
2

\textbf{ring()}

Return the parent ring.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: B = BooleanPolynomialRing(5, 'x')
sage: x0, x1, x2, x3, x4 = B.gens()
sage: f = x1*x2 + x2*x3 + x4 + x2*x4 + x3 + x4 + 1
sage: f.set().ring() is B
True
\end{verbatim}

\textbf{set()}

Return self.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: BS = (a+b+c).set()
sage: BS.set() is BS
True
\end{verbatim}

\textbf{size_double()}

Return the size of this set as a floating point number.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: B = BooleanPolynomialRing(5, 'x')
sage: x0, x1, x2, x3, x4 = B.gens()
sage: f = x1*x2 + x2*x3
sage: s = f.set()
sage: s.size_double()
2.0
\end{verbatim}

\textbf{stable_hash()}

A hash value which is stable across processes.

\textbf{EXAMPLES:}
sage: B.<x,y> = BooleanPolynomialRing()
sage: x.set() is x.set()
False
sage: x.set().stable_hash() == x.set().stable_hash()
True

Note: This function is part of the upstream PolyBoRi interface. In Sage all hashes are stable.

subset0(i)
Return a set of those elements in this set which do not contain the variable indexed by i.

INPUT:
• i - an index

EXAMPLES:

sage: BooleanPolynomialRing(5, 'x')
Boolean PolynomialRing in x0, x1, x2, x3, x4
sage: B = BooleanPolynomialRing(5, 'x')
sage: B.inject_variables()
Defining x0, x1, x2, x3, x4
sage: f = x1*x2+x2*x3
sage: s = f.set(); s
{{x1,x2}, {x2,x3}}
sage: s.subset0(1)
{{x2,x3}}

subset1(i)
Return a set of those elements in this set which do contain the variable indexed by i and evaluate the variable indexed by i to 1.

INPUT:
• i - an index

EXAMPLES:

sage: BooleanPolynomialRing(5, 'x')
Boolean PolynomialRing in x0, x1, x2, x3, x4
sage: B = BooleanPolynomialRing(5, 'x')
sage: B.inject_variables()
Defining x0, x1, x2, x3, x4
sage: f = x1*x2+x2*x3
sage: s = f.set(); s
{{x1,x2}, {x2,x3}}
sage: s.subset1(1)
{{x2}}

union(rhs)
Return the set theoretic union of this set and the set rhs.

The union of two sets $X$ and $Y$ is defined as:

$$X \cup Y = \{x | x \in X \text{ or } x \in Y\}.$$

EXAMPLES:
```python
sage: B = BooleanPolynomialRing(5, 'x')
sage: x0, x1, x2, x3, x4 = B.gens()
sage: f = x1*x2 + x2*x3
sage: s = f.set(); s
{{x1,x2}, {x2,x3}}
sage: g = x2*x3 + 1
sage: t = g.set(); t
{{x2,x3}, {}}
sage: s.union(t)
{{x1,x2}, {x2,x3}, {}}
```

```
vars()
Return the variables in this set as a monomial.

EXAMPLES:
```
```python
sage: B.<a,b,c,d,e,f> = Boolean PolynomialRing(order='lex')
sage: f = a + b*e + d*f + e + 1
sage: s = f.set()
sage: s
{{a}, {b,e}, {d,f}, {e}, {}}
sage: s.vars()
a*b*d*e*f
```

```python
class sage.rings.polynomial.pbori.pbori.BooleSetIterator
    Bases: object
    Helper class to iterate over boolean sets.

class sage.rings.polynomial.pbori.pbori.BooleanMonomial
    Bases: sage.structure.element.MonoidElement
    Construct a boolean monomial.

    INPUT:
    • parent - parent monoid this element lives in

    EXAMPLES:
```
```python
sage: from sage.rings.polynomial.pbori.pbori import BooleanMonomialMonoid,
    BooleanMonomial
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: M = BooleanMonomialMonoid(P)
sage: BooleanMonomial(M)
1
```

**Note:** Use the `BooleanMonomialMonoid__call__()` method and not this constructor to construct these objects.

```
de()  
Return degree of this monomial.

    EXAMPLES:
```
```
```
sage: from sage.rings.polynomial.pbori.pbori import BooleanMonomialMonoid
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: M = BooleanMonomialMonoid(P)
sage: M(x*y).deg()
2
sage: M(x*y*z).deg()
3
```

**Note:** This function is part of the upstream PolyBoRi interface.

---

### degree(x=None)

Return the degree of this monomial in `x`, where `x` must be one of the generators of the polynomial ring.

**INPUT:**

- `x` - boolean multivariate polynomial (a generator of the polynomial ring). If `x` is not specified (or is `None`), return the total degree of this monomial.

**EXAMPLES:**

```
sage: from sage.rings.polynomial.pbori.pbori import BooleanMonomialMonoid
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: M = BooleanMonomialMonoid(P)
sage: M(x*y).degree()
2
sage: M(x*y).degree(x)
1
sage: M(x*y).degree(z)
0
```

---

### divisors()

Return a set of boolean monomials with all divisors of this monomial.

**EXAMPLES:**

```
sage: B.<x,y,z> = BooleanPolynomialRing(3)
sage: f = x*y
sage: m = f.lm()
sage: m.divisors()
{{x,y}, {x}, {y}, {}}
```

---

### gcd(rhs)

Return the greatest common divisor of this boolean monomial and `rhs`.

**INPUT:**

- `rhs` - a boolean monomial

**EXAMPLES:**

```
sage: B.<a,b,c,d> = BooleanPolynomialRing()
sage: a,b,c,d = a.lm(), b.lm(), c.lm(), d.lm()
sage: (a*b).gcd(b*c)
b
```
\texttt{sage}: (a^b^c).gcd(d)
\texttt{1}

\textbf{index()}
Return the variable index of the first variable in this monomial.

\textbf{EXAMPLES:}
\begin{verbatim}
\texttt{sage}: B.<x,y,z> = BooleanPolynomialRing(3)
\texttt{sage}: f = x*y
\texttt{sage}: m = f.lm()
\texttt{sage}: m.index()
\texttt{0}
\end{verbatim}

\textbf{Note:} This function is part of the upstream PolyBoRi interface.

\textbf{iterindex()}
Return an iterator over the indices of the variables in self.

\textbf{EXAMPLES:}
\begin{verbatim}
\texttt{sage}: from sage.rings.polynomial.pbori.pbori import BooleanMonomialMonoid
\texttt{sage}: P.<x,y,z> = BooleanPolynomialRing(3)
\texttt{sage}: M = BooleanMonomialMonoid(P)
\texttt{sage}: list(M(x*z).iterindex())
\texttt{[0, 2]}
\end{verbatim}

\textbf{multiples}(\texttt{rhs})
Return a set of boolean monomials with all multiples of this monomial up to the bound \texttt{rhs}.

\textbf{INPUT:}
\begin{itemize}
  \item \texttt{rhs} - a boolean monomial
\end{itemize}

\textbf{EXAMPLES:}
\begin{verbatim}
\texttt{sage}: B.<x,y,z> = BooleanPolynomialRing(3)
\texttt{sage}: f = x
\texttt{sage}: m = f.lm()
\texttt{sage}: g = x*y*z
\texttt{sage}: n = g.lm()
\texttt{sage}: m.multiples(n)
\texttt{\{x,y,z\}, \{x,y\}, \{x,z\}, \{x\}}
\texttt{sage}: n.multiples(m)
\texttt{\{\{x,y,z\}\}}
\end{verbatim}

\textbf{Note:} The returned set always contains \texttt{self} even if the bound \texttt{rhs} is smaller than \texttt{self}.

\textbf{navigation()}
Navigators provide an interface to diagram nodes, accessing their index as well as the corresponding then- and else-branches.
You should be very careful and always keep a reference to the original object, when dealing with navigators, as navigators contain only a raw pointer as data. For the same reason, it is necessary to supply the ring as argument, when constructing a set out of a navigator.

EXAMPLES:

```python
sage: from sage.rings.polynomial.pbori.pbori import BooleSet
sage: B = BooleanPolynomialRing(5, 'x')
sage: x0, x1, x2, x3, x4 = B.gens()
sage: f = x1*x2+x2*x3*x4+x2*x4+x3+x4+1
sage: m = f.lm(); m
x1*x2
sage: nav = m.navigation()
sage: BooleSet(nav, B)
{{x1,x2}}
sage: nav.value()
1
```

**reducible_by**(rhs)

Return True if self is reducible by rhs.

**INPUT:**

• rhs - a boolean monomial

**EXAMPLES:**

```python
sage: B.<x,y,z> = BooleanPolynomialRing(3)
sage: f = x*y
sage: m = f.lm()
sage: m.reducible_by((x*y).lm())
True
sage: m.reducible_by((x*z).lm())
False
```

**ring()**

Return the corresponding boolean ring.

**EXAMPLES:**

```python
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: a.lm().ring() is B
True
```

**set()**

Return a boolean set of variables in this monomials.

**EXAMPLES:**

```python
sage: B.<x,y,z> = BooleanPolynomialRing(3)
sage: f = x*y
sage: m = f.lm()
sage: m.set()
{{x,y}}
```
**stable_hash()**
A hash value which is stable across processes.

EXAMPLES:

```python
sage: B.<x,y> = BooleanPolynomialRing()
sage: x.lm() == x.lm()  # False
sage: x.lm().stable_hash() == x.lm().stable_hash()  # True
```

**Note:** This function is part of the upstream PolyBoRi interface. In Sage all hashes are stable.

**variables()**
Return a tuple of the variables in this monomial.

EXAMPLES:

```python
sage: from sage.rings.polynomial.pbori.pbori import BooleanMonomialMonoid
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: M = BooleanMonomialMonoid(P)
sage: M(x*z).variables()  # indirect doctest
(x, z)
```

**class** sage.rings.polynomial.pbori.pbori.BooleanMonomialIterator

Bases: object

An iterator over the variable indices of a monomial.

**class** sage.rings.polynomial.pbori.pbori.BooleanMonomialMonoid(polring)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.monoids.monoid.Monoid_class

Construct a boolean monomial monoid given a boolean polynomial ring.

This object provides a parent for boolean monomials.

INPUT:

- `polring` - the polynomial ring our monomials lie in

EXAMPLES:

```python
sage: from sage.rings.polynomial.pbori.pbori import BooleanMonomialMonoid
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: M = BooleanMonomialMonoid(P)
sage: M
MonomialMonoid of Boolean PolynomialRing in x, y
sage: M.gens()
(x, y)
sage: type(M.gens(0))
<type 'sage.rings.polynomial.pbori.pbori.BooleanMonomial'>
```

Since trac ticket #9138, boolean monomial monoids are unique parents and are fit into the category framework.
```
sage: loads(dumps(M)) is M
True
sage: TestSuite(M).run()
```

**gen**(*i=*0)

Return the i-th generator of self.

**INPUT:**

- *i* - an integer

**EXAMPLES:**

```
sage: from sage.rings.polynomial.pbori.pbori import BooleanMonomialMonoid
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: M = BooleanMonomialMonoid(P)
sage: M.gen(0)
x
sage: M.gen(2)
z

sage: P = BooleanPolynomialRing(1000, 'x')
sage: M = BooleanMonomialMonoid(P)
sage: M.gen(50)
x50
```

**gens()**

Return the tuple of generators of this monoid.

**EXAMPLES:**

```
sage: from sage.rings.polynomial.pbori.pbori import BooleanMonomialMonoid
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: M = BooleanMonomialMonoid(P)
sage: M.gens()
(x, y, z)
```

**ngens()**

Return the number of variables in this monoid.

**EXAMPLES:**

```
sage: from sage.rings.polynomial.pbori.pbori import BooleanMonomialMonoid
sage: P = BooleanPolynomialRing(100, 'x')
sage: M = BooleanMonomialMonoid(P)
sage: M.ngens()
100
```

**class** `sage.rings.polynomial.pbori.pbori.BooleanMonomialVariableIterator`

**Bases:** object

**class** `sage.rings.polynomial.pbori.pbori.BooleanMulAction`

**Bases:** sage.categories.action.Action

**class** `sage.rings.polynomial.pbori.pbori.BooleanPolynomial`

**Bases:** sage.rings.polynomial.multi_polynomial.MPolynomial

Construct a boolean polynomial object in the given boolean polynomial ring.
INPUT:

- **parent** - a boolean polynomial ring

**Note:** Do not use this method to construct boolean polynomials, but use the appropriate \_call\_ method in the parent.

**constant()**

Return True if this element is constant.

**EXAMPLES:**

```sage
sage: B.<x,y,z> = BooleanPolynomialRing(3)
sage: x.constant()  
False
sage: B(1).constant()  
True
```

**Note:** This function is part of the upstream PolyBoRi interface.

**constant_coefficient()**

Return the constant coefficient of this boolean polynomial.

**EXAMPLES:**

```sage
sage: B.<a,b> = BooleanPolynomialRing()
sage: a.constant_coefficient()  
0
sage: (a+1).constant_coefficient()  
1
```

**deg()**

Return the degree of self. This is usually equivalent to the total degree except for weighted term orderings which are not implemented yet.

**EXAMPLES:**

```sage
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: (x+y).degree()  
1
sage: P(1).degree()  
0
sage: (x*y + x + y + 1).degree()  
2
```

**Note:** This function is part of the upstream PolyBoRi interface.
**degree**($x$=None)

Return the maximal degree of this polynomial in $x$, where $x$ must be one of the generators for the parent of this polynomial.

If $x$ is not specified (or is None), return the total degree, which is the maximum degree of any monomial.

**EXAMPLES:**

```python
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: (x+y).degree()
1

sage: P(1).degree()
0

sage: (x*y + x + y + 1).degree()
2

sage: (x*y + x + y + 1).degree(x)
1
```

**elength()**

Return elimination length as used in the SlimGB algorithm.

**EXAMPLES:**

```python
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: x.elength()
1

sage: f = x*y + 1
sage: f.elength()
2
```

**REFERENCES:**


**Note:** This function is part of the upstream PolyBoRi interface.

**first_term()**

Return the first term with respect to the lexicographical term ordering.

**EXAMPLES:**

```python
sage: B.<a,b,z> = BooleanPolynomialRing(3,order='lex')
sage: f = b^2*z + a + 1
sage: f.first_term()
a
```

**Note:** This function is part of the upstream PolyBoRi interface.

**graded_part**($deg$)

Return graded part of this boolean polynomial of degree $deg$. 
INPUT:

• `deg` - a degree

EXAMPLES:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: f = a*b*c + c*d + a*b + 1
sage: f.graded_part(2)
a*b + c*d

sage: f.graded_part(0)
1
```

`has_constant_part()`

Return True if this boolean polynomial has a constant part, i.e. if `1` is a term.

EXAMPLES:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: f = a*b*c + c*d + a*b + 1
sage: f.has_constant_part()
True

sage: f = a*b*c + c*d + a*b
sage: f.has_constant_part()
False
```

`is_constant()`

Check if `self` is constant.

EXAMPLES:

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: P(1).is_constant()
True

sage: P(0).is_constant()
True

sage: x.is_constant()
False

sage: (x*y).is_constant()
False
```

`is_equal(right)`

EXAMPLES:

```
sage: B.<a,b,z> = BooleanPolynomialRing(3)
sage: f = a*z + b + 1
sage: g = b + z
sage: f.is_equal(g)
False
```

(continues on next page)
\begin{verbatim}
 sage: f.is_equal((f + 1) - 1)
 True
\end{verbatim}

**Note:** This function is part of the upstream PolyBoRi interface.

### is_homogeneous()

Return True if this element is a homogeneous polynomial.

**EXAMPLES:**

\begin{verbatim}
 sage: P.<x, y> = BooleanPolynomialRing()
 sage: (x+y).is_homogeneous()
 True
 sage: P(0).is_homogeneous()
 True
 sage: (x+1).is_homogeneous()
 False
\end{verbatim}

### is_one()

Check if self is 1.

**EXAMPLES:**

\begin{verbatim}
 sage: P.<x,y> = BooleanPolynomialRing(2)
 sage: P(1).is_one()
 True
 sage: P.one().is_one()
 True
 sage: x.is_one()
 False
 sage: P(0).is_one()
 False
\end{verbatim}

### is_pair()

Check if self has exactly two terms.

**EXAMPLES:**

\begin{verbatim}
 sage: P.<x,y> = BooleanPolynomialRing(2)
 sage: P(0).is_pair()
 False
 sage: x.is_pair()
 False
 sage: P(1).is_pair()
 False
 sage: (x*y).is_pair()
 False
\end{verbatim}
sage: (x + y).is_pair()
True

sage: (x + 1).is_pair()
True

sage: (x*y + 1).is_pair()
True

sage: (x + y + 1).is_pair()
False

sage: ((x + 1)*(y + 1)).is_pair()
False

is_singleton()
Check if self has at most one term.

EXAMPLES:

sage: P.<x,y> = BooleanPolynomialRing(2)
sage: P(0).is_singleton()
True

sage: x.is_singleton()
True

sage: P(1).is_singleton()
True

sage: (x*y).is_singleton()
True

sage: (x + y).is_singleton()
False

sage: (x + 1).is_singleton()
False

sage: (x*y + 1).is_singleton()
False

sage: (x + y + 1).is_singleton()
False

sage: ((x + 1)*(y + 1)).is_singleton()
False

is_singleton_or_pair()
Check if self has at most two terms.

EXAMPLES:
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: P(0).is_singleton_or_pair()
True

sage: x.is_singleton_or_pair()
True

sage: P(1).is_singleton_or_pair()
True

sage: (x*y).is_singleton_or_pair()
True

sage: (x + y).is_singleton_or_pair()
True

sage: (x + 1).is_singleton_or_pair()
True

sage: (x*y + 1).is_singleton_or_pair()
True

sage: (x + y + 1).is_singleton_or_pair()
False

sage: ((x + 1)*(y + 1)).is_singleton_or_pair()
False

is_unit()
Check if self is invertible in the parent ring.

Note that this condition is equivalent to being 1 for boolean polynomials.

EXAMPLES:

sage: P.<x,y> = BooleanPolynomialRing(2)
sage: P.one().is_unit()
True

sage: x.is_unit()
False

is_univariate()
Return True if self is a univariate polynomial.

This means that self contains at most one variable.

EXAMPLES:

sage: P.<x,y,z> = BooleanPolynomialRing()
sage: f = x + 1
sage: f.is_univariate()
True

sage: f = y*x + 1
sage: f.is_univariate()
(continues on next page)
is_zero()  
Check if self is zero.

EXAMPLES:

```python
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: P(0).is_zero()
True
sage: x.is_zero()
False
sage: P(1).is_zero()
False
```

lead()  
Return the leading monomial of boolean polynomial, with respect to the order of parent ring.

EXAMPLES:

```python
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: (x+y+y*z).lead()
x
sage: P.<x,y,z> = BooleanPolynomialRing(3, order='deglex')
sage: (x+y+y*z).lead()
y*z
```

Note: This function is part of the upstream PolyBoRi interface.

lead_deg()  
Return the total degree of the leading monomial of self.

EXAMPLES:

```python
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: p = x + y*z
sage: p.lead_deg()
1
sage: P.<x,y,z> = BooleanPolynomialRing(3, order='deglex')
sage: p = x + y*z
sage: p.lead_deg()
2
sage: P(0).lead_deg()
0
```
Note: This function is part of the upstream PolyBoRi interface.

lead_divisors()
Return a BooleSet of all divisors of the leading monomial.

EXAMPLES:

```python
sage: B.<a,b,z> = BooleanPolynomialRing(3)
sage: f = a*b + z + 1
sage: f.lead_divisors()
{{a,b}, {a}, {b}, {}}
```

Note: This function is part of the upstream PolyBoRi interface.

lex_lead()
Return the leading monomial of boolean polynomial, with respect to the lexicographical term ordering.

EXAMPLES:

```python
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: (x+y+y*z).lex_lead()
x
sage: P.<x,y,z> = BooleanPolynomialRing(3, order='deglex')
sage: (x+y+y*z).lex_lead()
x
sage: P(0).lex_lead()
0
```

Note: This function is part of the upstream PolyBoRi interface.

lex_lead_deg()
Return degree of leading monomial with respect to the lexicographical ordering.

EXAMPLES:

```python
sage: B.<x,y,z> = BooleanPolynomialRing(3,order='lex')
sage: f = x + y*z
sage: f
x + y*z
sage: f.lex_lead_deg()
1
sage: B.<x,y,z> = BooleanPolynomialRing(3,order='deglex')
sage: f = x + y*z
sage: f
y*z + x
sage: f.lex_lead_deg()
1
```
Note: This function is part of the upstream PolyBoRi interface.

\textbf{lm()}

Return the leading monomial of this boolean polynomial, with respect to the order of parent ring.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: (x+y+y*z).lm() x
\end{verbatim}

\begin{verbatim}
sage: P.<x,y,z> = BooleanPolynomialRing(3, order='deglex')
sage: (x+y+y*z).lm() y*z
\end{verbatim}

\begin{verbatim}
sage: P(0).lm() 0
\end{verbatim}

\textbf{lt()}

Return the leading term of this boolean polynomial, with respect to the order of the parent ring.

Note that for boolean polynomials this is equivalent to returning leading monomials.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: (x+y+y*z).lt() x
\end{verbatim}

\begin{verbatim}
sage: P.<x,y,z> = BooleanPolynomialRing(3, order='deglex')
sage: (x+y+y*z).lt() y*z
\end{verbatim}

\textbf{map_every_x_to_x_plus_one()}

Map every variable $x_i$ in this polynomial to $x_i + 1$.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: B.<a,b,z> = BooleanPolynomialRing(3)
sage: f = a*b + z + 1; f
a*b + z + 1
\end{verbatim}

\begin{verbatim}
sage: f.map_every_x_to_x_plus_one()
a*b + a + b + z + 1
\end{verbatim}

\textbf{monomial_coefficient(mon)}

Return the coefficient of the monomial $\text{mon}$ in $\text{self}$, where $\text{mon}$ must have the same parent as $\text{self}$.

\textbf{INPUT:}

- $\text{mon}$ - a monomial

\textbf{EXAMPLES:}

sage: P.<x,y> = BooleanPolynomialRing(2)
sage: x.monomial_coefficient(x)
1
sage: x.monomial_coefficient(y)
0
sage: R.<x,y,z,a,b,c>=BooleanPolynomialRing(6)
sage: f=(1-x)*(1+y); f
x*y + x + y + 1
sage: f.monomial_coefficient(1)
1
sage: f.monomial_coefficient(0)
0

monomials()
Return a list of monomials appearing in self ordered largest to smallest.

EXAMPLES:

sage: P.<a,b,c> = BooleanPolynomialRing(3,order='lex')
sage: f = a + c*b
sage: f.monomials()
[a, b*c]
sage: P.<a,b,c> = BooleanPolynomialRing(3,order='deglex')
sage: f = a + c*b
sage: f.monomials()
[b*c, a]
sage: P.zero().monomials()
[]

n_nodes()
Return the number of nodes in the ZDD implementing this polynomial.

EXAMPLES:

sage: B = BooleanPolynomialRing(5,'x')
sage: x0,x1,x2,x3,x4 = B.gens()
sage: f = x1*x2 + x2*x3 + 1
sage: f.n_nodes()
4

Note: This function is part of the upstream PolyBoRi interface.

n_vars()
Return the number of variables used to form this boolean polynomial.

EXAMPLES:

sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: f = a*b*c + 1

(continues on next page)
Note: This function is part of the upstream PolyBoRi interface.

navigation()  
Navigators provide an interface to diagram nodes, accessing their index as well as the corresponding then- and else-branches.

You should be very careful and always keep a reference to the original object, when dealing with navigators, as navigators contain only a raw pointer as data. For the same reason, it is necessary to supply the ring as argument, when constructing a set out of a navigator.

EXAMPLES:

```python
sage: from sage.rings.polynomial.pbori.pbori import BooleSet
sage: B = BooleanPolynomialRing(5,'x')
sage: x0,x1,x2,x3,x4 = B.gens()
sage: f = x1*x2+x2*x3*x4+x2*x4+x3+x4+1
sage: nav = f.navigation()
sage: BooleSet(nav, B)
{{x1, x2}, {x2, x3, x4}, {x2, x4}, {x3}, {x4}, {}}
```

```python
sage: nav.value()
1
```

```python
sage: nav_else = nav.else_branch()
```

```python
sage: BooleSet(nav_else, B)
{{x2, x3, x4}, {x2, x4}, {x3}, {x4}, {}}
```

```python
sage: nav_else.value()
2
```

Note: This function is part of the upstream PolyBoRi interface.

nvariables()  
Return the number of variables used to form this boolean polynomial.

EXAMPLES:

```python
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: f = a*b*c + 1
sage: f.nvariables()
3
```

reduce(I)  
Return the normal form of self w.r.t. I, i.e. return the remainder of self with respect to the polynomials in I. If the polynomial set/list I is not a Groebner basis the result is not canonical.

INPUT:
• I - a list/set of polynomials in self.parent(). If I is an ideal, the generators are used.

EXAMPLES:

```python
sage: B.<x0,x1,x2,x3> = BooleanPolynomialRing(4)
sage: I = B.ideal((x0 + x1 + x2 + x3,
....: x0*x1 + x1*x2 + x0*x3 + x2*x3,
....: x0*x1*x2 + x0*x1*x3 + x0*x2*x3 + x1*x2*x3,
....: x0*x1*x2*x3 + 1))
sage: gb = I.groebner_basis()
sage: f,g,h,i = I.gens()
sage: f.reduce(gb)
0
sage: p = f*g + x0*h + x2*i
sage: p.reduce(gb)
0
sage: p.reduce(I)
x1*x2*x3 + x2
sage: p.reduce([])
x0*x1*x2 + x0*x1*x3 + x0*x2*x3 + x2
```

Note: If this function is called repeatedly with the same I then it is advised to use PolyBoRi’s `GroebnerStrategy` object directly, since that will be faster. See the source code of this function for details.

**reducible_by**(rhs)
Return True if this boolean polynomial is reducible by the polynomial rhs.

INPUT:
• rhs - a boolean polynomial

EXAMPLES:

```python
sage: B.<a,b,c,d> = BooleanPolynomialRing(4,order='deglex')
sage: f = (a*b + 1)*(c + 1)
sage: f.reducible_by(d)
False
sage: f.reducible_by(c)
True
sage: f.reducible_by(c + 1)
True
```

Note: This function is part of the upstream PolyBoRi interface.

**ring**()
Return the parent of this boolean polynomial.

EXAMPLES:

```python
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: a.ring() is B
True
```
**set()**

Return a BooleSet with all monomials appearing in this polynomial.

**EXAMPLES:**

```
sage: B.<a,b,z> = BooleanPolynomialRing(3)
sage: (a*b+z+1).set()
{{a,b}, {z}, {}}
```

**spoly**

Return the S-Polynomial of this boolean polynomial and the other boolean polynomial \( \text{rhs} \).

**EXAMPLES:**

```
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: f = a*b*c + c*d + a*b + 1
sage: g = c*d + b
sage: f.spoly(g)
a*b + a*c*d + c*d + 1
```

**Note:** This function is part of the upstream PolyBoRi interface.

**stable_hash**

A hash value which is stable across processes.

**EXAMPLES:**

```
sage: B.<x,y> = BooleanPolynomialRing()
sage: x is B.gen(0)
False
sage: x.stable_hash() == B.gen(0).stable_hash()
True
```

**Note:** This function is part of the upstream PolyBoRi interface. In Sage all hashes are stable.

**subs**(in\_dict=None, **kwds)

Fixes some given variables in a given boolean polynomial and returns the changed boolean polynomials. The polynomial itself is not affected. The variable, value pairs for fixing are to be provided as dictionary of the form \{variable:value\} or named parameters (see examples below).

**INPUT:**

- in\_dict - (optional) dict with variable:value pairs
- **kwds** - names parameters

**EXAMPLES:**

```
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: f = x*y + z + y*z + 1
sage: f.subs(x=1)
y*z + y + z + 1
sage: f.subs(x=0)
y*z + z + 1
```
This method can work fully symbolic:

```
sage: f.subs(x=var('a'), y=var('b'), z=var('c'))
a*b + b*c + c + 1
sage: f.subs({x:var('a'), y:var('b'), z:var('c')})
a*b + b*c + c + 1
```

**terms()**
Return a list of monomials appearing in self ordered largest to smallest.

EXAMPLES:

```
sage: P.<a,b,c> = BooleanPolynomialRing(3, order='lex')
sage: f = a + c*b
sage: f.terms()
[a, b*c]
sage: P.<a,b,c> = BooleanPolynomialRing(3, order='deglex')
sage: f = a + c*b
sage: f.terms()
[b*c, a]
```

**total_degree()**
Return the total degree of self.

EXAMPLES:

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: (x+y).total_degree()
1
```

```
sage: P(1).total_degree()
0
```

```
sage: (x*y + x + y + 1).total_degree()
2
```

**univariate_polynomial**(R=None)
Return a univariate polynomial associated to this multivariate polynomial.
If this polynomial is not in at most one variable, then a ValueError exception is raised. This is checked using the \texttt{is_univariate()} method. The new Polynomial is over GF(2) and in the variable \texttt{x} if no ring \texttt{R} is provided.

```sage
sage: R.<x, y> = BooleanPolynomialRing() sage: f = x - y + x*y + 1 sage: f.univariate_polynomial() Traceback (most recent call last): ... ValueError: polynomial must involve at most one variable sage: g = f.subs({x:0}); g y + 1 sage: g.univariate_polynomial() y + 1 sage: g.univariate_polynomial(GF(2)['foo']) foo + 1
```

Here’s an example with a constant multivariate polynomial:

```sage
sage: g = R(1) sage: h = g.univariate_polynomial(); h 1 sage: h.parent() Univariate Polynomial Ring in x over Finite Field of size 2 (using GF2X)
```

\textbf{variable} \textit{(i=0)}

Return the \textit{i}-th variable occurring in \texttt{self}. The index \texttt{i} is the index in \texttt{self.variables()}

\textbf{EXAMPLES:}

```sage
sage: P.<x,y,z> = BooleanPolynomialRing(3) sage: f = x*z + z + 1 sage: f.variables() (x, z) sage: f.variable(1) z
```

\textbf{variables}()

Return a tuple of all variables appearing in \texttt{self}.

\textbf{EXAMPLES:}

```sage
sage: P.<x,y,z> = BooleanPolynomialRing(3) sage: (x + y).variables() (x, y) sage: (x*y + z).variables() (x, y, z) sage: P.zero().variables() () sage: P.one().variables() ()
```

\textbf{vars_as_monomial}()

Return a boolean monomial with all variables appearing in \texttt{self}.

\textbf{EXAMPLES:}

```sage
sage: P.<x,y,z> = BooleanPolynomialRing(3) sage: (x + y).vars_as_monomial() x*y
```

(continues on next page)
sage: (x*y + z).vars_as_monomial()
x*y*z

sage: P.zero().vars_as_monomial()
1

sage: P.one().vars_as_monomial()
1

Note: This function is part of the upstream PolyBoRi interface.

```python
zeros_in(s)
```
Return a set containing all elements of \( s \) where this boolean polynomial evaluates to zero.

If \( s \) is given as a BooleSet, then the return type is also a BooleSet. If \( s \) is a set/list/tuple of tuple this function returns a tuple of tuples.

INPUT:

\[ \cdot s \text{ - candidate points for evaluation to zero} \]

EXAMPLES:

```python
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: f = a*b + c + d + 1
```

Now we create a set of points:

```python
sage: s = a*b + a*b*c + c*d + 1
sage: s = s.set(); s
{{a,b,c}, {a,b}, {c,d}, {}}
```

This encodes the points (1,1,1,0), (1,1,0,0), (0,0,1,1) and (0,0,0,0). But of these only (1,1,0,0) evaluates to zero.

```python
sage: f.zeros_in(s)
{{a,b}}
```

```python
sage: f.zeros_in([(1,1,1,0), (1,1,0,0), (0,0,1,1), (0,0,0,0)])
((1, 1, 0, 0),)
```

```python
class sage.rings.polynomial.pbori.pbori.BooleanPolynomialEntry
Bases: object

p
```

```python
class sage.rings.polynomial.pbori.pbori.BooleanPolynomialIdeal(ring, gens=[], coerce=True)
Bases: sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal
```

Construct an ideal in the boolean polynomial ring.

INPUT:

\[ \cdot \text{ring - the ring this ideal is defined in} \]
\[ \cdot \text{gens - a list of generators} \]
• coerce - coerce all elements to the ring ring (default: True)

EXAMPLES:

```python
code: sage: P.<x0, x1, x2, x3> = BooleanPolynomialRing(4)
code: sage: I = P.ideal(x0*x1*x2*x3 + x0*x1*x3 + x0*x1 + x0*x2 + x0)
code: sage: I
Ideal (x0*x1*x2*x3 + x0*x1*x3 + x0*x1 + x0*x2 + x0) of Boolean PolynomialRing in x0, x1, x2, x3
code: sage: loads(dumps(I)) == I
True
```

dimension()

Return the dimension of self, which is always zero.

groebner_basis(algorithm='polybori', **kwds)

Return a Groebner basis of this ideal.

INPUT:

- algorithm - either "polybori" (built-in default) or "magma" (requires Magma).
- red_tail - tail reductions in intermediate polynomials, this option affects mainly heuristics. The reducedness of the output polynomials can only be guaranteed by the option redsb (default: True)
- mnsb - return a minimal Groebner basis (default: True)
- redsb - return a minimal Groebner basis and all tails are reduced (default: True)
- deg_bound - only compute Groebner basis up to a given degree bound (default: False)
- faugere - turn off or on the linear algebra (default: False)
- linear_algebra_in_last_block - this affects the last block of block orderings and degree orderings. If it is set to True linear algebra takes affect in this block. (default: True)
- gauss_on_linear - perform Gaussian elimination on linear polynomials (default: True)
- selection_size - maximum number of polynomials for parallel reductions (default: 1000)
- heuristic - Turn off heuristic by setting heuristic=False (default: True)
- lazy - (default: True)
- invert - setting invert=True input and output get a transformation x+1 for each variable x, which shouldn't effect the calculated GB, but the algorithm.
- other_ordering_first - possible values are False or an ordering code. In practice, many Boolean examples have very few solutions and a very easy Groebner basis. So, a complex walk algorithm (which cannot be implemented using the data structures) seems unnecessary, as such Groebner bases can be converted quite fast by the normal Buchberger algorithm from one ordering into another ordering. (default: False)
- prot - show protocol (default: False)
- full_prot - show full protocol (default: False)

EXAMPLES:

```python
code: sage: P.<x0, x1, x2, x3> = BooleanPolynomialRing(4)
code: sage: I = P.ideal(x0*x1*x2*x3 + x0*x1*x3 + x0*x1 + x0*x2 + x0)
code: sage: I.groebner_basis()
[x0*x1 + x0*x2 + x0, x0*x2*x3 + x0*x3]
```
Another somewhat bigger example:

```python
sage: sr = mq.SR(2,1,1,4,gf2=True, polybori=True)
sage: F, s = sr.polynomial_system()
sage: I = F.ideal()
sage: I.groebner_basis()
Polynomial Sequence with 36 Polynomials in 36 Variables
```

We compute the same example with Magma:

```python
sage: sr = mq.SR(2,1,1,4,gf2=True, polybori=True)
sage: F, s = sr.polynomial_system()
sage: I = F.ideal()
sage: I.groebner_basis(algorithm='magma', prot='sage')  # optional - magma
Leading term degree: 3. Critical pairs: 101 (all pairs of current degree → eliminated by criteria).

Highest degree reached during computation: 3.
Polynomial Sequence with 35 Polynomials in 36 Variables
```

```python
interreduced_basis()
```

If this ideal is spanned by \((f_1, \ldots, f_n)\) this method returns \((g_1, \ldots, g_s)\) such that:

* \(<f_1, \ldots, f_n> = <g_1, \ldots, g_s>
* \(LT(g_i) \neq LT(g_j)\) for all \(i \neq j\)
* \(LT(g_i)\) does not divide \(m\) for all monomials \(m\) of \(\{g_1, \ldots, g_{i-1}, g_{i+1}, \ldots, g_s\}\)

```python
EXAMPLES:
```
```python
sage: sr = mq.SR(1, 1, 1, 4, gf2=True, polybori=True)
sage: F, s = sr.polynomial_system()
sage: I = F.ideal()
sage: I.interreduced_basis()
[k100 + 1, k101 + k001 + 1, k102, k103 + 1, x100 + k001 + 1, x101 + k001, x102, ...
 → x103 + k001, w100 + 1, w101 + k001 + 1, w102 + 1, w103 + 1, s000 + k001, s001,
 → + k001 + 1, s002, s003 + k001 + 1, k000 + 1, k002 + 1, k003 + 1]
```

```python
reduce()
```

Reduce an element modulo the reduced Groebner basis for this ideal. This returns 0 if and only if the element is in this ideal. In any case, this reduction is unique up to monomial orders.

```python
EXAMPLES:
```
```python
sage: P = PolynomialRing(GF(2),10, 'x')
sage: B = BooleanPolynomialRing(10, 'x')
sage: I = sage.rings.ideal.Cyclic(P)
sage: I = B.ideal([B(f) for f in I.gens()])
sage: gb = I.groebner_basis()
sage: I.reduce(gb[0])
(continues on next page)
variety(**kwds)

Return the variety associated to this boolean ideal.

EXAMPLES:

A simple example:

```python
sage: from sage.doctest.fixtures import reproducible_repr
sage: R.<x,y,z> = BooleanPolynomialRing()
sage: I = ideal([ x*y*z + x*z + y + 1, x+y+z+1 ])
sage: print(reproducible_repr(I.variety()))
[(x: 0, y: 1, z: 0), (x: 1, y: 1, z: 1)]
```

class sage.rings.polynomial.pbori.pbori.BooleanPolynomialIterator

Bases: object

Iterator over the monomials of a boolean polynomial.

class sage.rings.polynomial.pbori.pbori.BooleanPolynomialRing

Bases: sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base

Construct a boolean polynomial ring with the following parameters:

INPUT:

- n - number of variables (an integer > 1)
- names - names of ring variables, may be a string or list/tuple
- order - term order (default: lex)

EXAMPLES:

```python
sage: R.<x, y, z> = BooleanPolynomialRing()
sage: R
Boolean PolynomialRing in x, y, z
sage: p = x*y + x*z + y*z
sage: x*p
x*y*z + x*y + x*z
sage: R.term_order()
Lexicographic term order
sage: R = BooleanPolynomialRing(5, 'x', order='deglex(3),deglex(2)')
sage: R.term_order()
Block term order with blocks:
(Degree lexicographic term order of length 3,
 Degree lexicographic term order of length 2)
```
sage: R = BooleanPolynomialRing(3, 'x', order='deglex')
sage: R.term_order()
Degree lexicographic term order

change_ring(base_ring=None, names=None, order=None)

Return a new multivariate polynomial ring with base ring base_ring, variable names set to names, and
term ordering given by order.

When base_ring is not specified, this function returns a BooleanPolynomialRing isomorphic to self.
Otherwise, this returns a MPolynomialRing. Each argument above is optional.

INPUT:
• base_ring – a base ring
• names – variable names
• order – a term order

EXAMPLES:

sage: P.<x, y, z> = BooleanPolynomialRing()
sage: P.term_order()
Lexicographic term order
sage: R = P.change_ring(names=('a', 'b', 'c'), order="deglex")
sage: R
Boolean PolynomialRing in a, b, c
sage: R.term_order()
Degree lexicographic term order
sage: T = P.change_ring(base_ring=GF(3))
sage: T
Multivariate Polynomial Ring in x, y, z over Finite Field of size 3
sage: T.term_order()
Lexicographic term order

clone(ordering=None, names=[], blocks=[])

Shallow copy this boolean polynomial ring, but with different ordering, names or blocks if given.

ring.clone(ordering=..., names=..., block=...) generates a shallow copy of ring, but with different ordering, names or blocks if given.

EXAMPLES:

sage: B.<a,b,c> = BooleanPolynomialRing()
sage: B.clone()
Boolean PolynomialRing in a, b, c
sage: B.<x,y,z> = BooleanPolynomialRing(3, order='deglex')
sage: y*z > x
True

Now we call the clone method and generate a compatible, but 'lex' ordered, ring:

sage: C = B.clone(ordering=0)
sage: C(y*z) > C(x)
False

Now we change variable names:
sage: P.<x0,x1> = BooleanPolynomialRing(2)
sage: P
Boolean PolynomialRing in x0, x1

sage: Q = P.clone(names=['t'])
sage: Q
Boolean PolynomialRing in t, x1

We can also append blocks to block orderings this way:

sage: R.<x1,x2,x3,x4> = BooleanPolynomialRing(order='deglex(1),deglex(3)')
sage: x2 > x3*x4
False

Now we call the internal method and change the blocks:

sage: S = R.clone(blocks=[3])
sage: S(x2) > S(x3*x4)
True

Note: This is part of PolyBoRi’s native interface.

construction()
A boolean polynomial ring is the quotient of a polynomial ring, in a special implementation.

Before trac ticket #15223, the boolean polynomial rings returned the construction of a polynomial ring, which was of course wrong.

Now, a QuotientFunctor is returned that knows about the ”_phr” implementation.

EXAMPLES:

sage: P.<x0, x1, x2, x3> = BooleanPolynomialRing(4,order='degneglex(2), →degneglex(2)')
sage: F,O = P.construction()
sage: O
Multivariate Polynomial Ring in x0, x1, x2, x3 over Finite Field of size 2
sage: F
QuotientFunctor
sage: F(O) is P
True

cover_ring()
Return \( R = \mathbb{F}_2[x_1, x_2, \ldots, x_n] \) if \( x_1, x_2, \ldots, x_n \) is the ordered list of variable names of this ring. \( R \) also has the same term ordering as this ring.

EXAMPLES:

sage: B.<x,y> = BooleanPolynomialRing(2)
sage: R = B.cover_ring(); R
Multivariate Polynomial Ring in x, y over Finite Field of size 2
sage: B.term_order() == R.term_order()
True

7.1. Boolean Polynomials
The cover ring is cached:

```python
sage: B.cover_ring() is B.cover_ring()
True
```

defining_ideal()
Return \( I = \langle x_i^2 + x_i \rangle \subseteq R \) where \( R = \text{self.cover_ring()} \), and \( x_i \) any element in the set of variables of this ring.

EXAMPLES:

```python
sage: B.<x,y> = BooleanPolynomialRing(2)
sage: I = B.defining_ideal(); I
Ideal (x^2 + x, y^2 + y) of Multivariate Polynomial Ring in x, y over Finite Field of size 2
```

gen(i=0)
Return the \( i \)-th generator of this boolean polynomial ring.

INPUT:

- \( i \) - an integer or a boolean monomial in one variable

EXAMPLES:

```python
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: P.gen()
x
sage: P.gen(2)
z
sage: m = x.monomials()[0]
sage: P.gen(m)
x
```

gens()
Return the tuple of variables in this ring.

EXAMPLES:

```python
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: P.gens()
(x, y, z)
sage: P = BooleanPolynomialRing(10, 'x')
sage: P.gens()
(x0, x1, x2, x3, x4, x5, x6, x7, x8, x9)
```

get_base_order_code()

EXAMPLES:

```python
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: B.get_base_order_code()
0
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing(order='deglex')
sage: B.get_base_order_code()
```

(continues on next page)
sage: T = TermOrder('deglex', 2) + TermOrder('deglex', 2)
sage: B.<a,b,c,d> = BooleanPolynomialRing(4, order=T)
sage: B.get_base_order_code()
1

Note: This function which is part of the PolyBoRi upstream API works with a current global ring. This notion is avoided in Sage.

get_order_code()

EXAMPLES:

sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: B.get_order_code()
0
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing(order='deglex')
sage: B.get_order_code()
1

Note: This function which is part of the PolyBoRi upstream API works with a current global ring. This notion is avoided in Sage.

has_degree_order()

Return checks whether the order code corresponds to a degree ordering.

EXAMPLES:

sage: P.<x,y> = BooleanPolynomialRing(2)
sage: P.has_degree_order()
False

id()

Return a unique identifier for this boolean polynomial ring.

EXAMPLES:

sage: P.<x,y> = BooleanPolynomialRing(2)
sage: print("id: {}".format(P.id()))
id: ...
sage: P = BooleanPolynomialRing(10, 'x')
sage: Q = BooleanPolynomialRing(20, 'x')
sage: P.id() != Q.id()
True

ideal(*gens, **kwds)

Create an ideal in this ring.

INPUT:
• gens - list or tuple of generators
• coerce - bool (default: True) automatically coerce the given polynomials to this ring to form the ideal

EXAMPLES:

```python
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: P.ideal(x+y)
Ideal (x + y) of Boolean PolynomialRing in x, y, z
```

```python
sage: P.ideal(x*y, y*z)
Ideal (x*y, y*z) of Boolean PolynomialRing in x, y, z
```

```python
sage: P.ideal([x+y, z])
Ideal (x + y, z) of Boolean PolynomialRing in x, y, z
```

**interpolation_polynomial** *(zeros, ones)*

Return the lexicographically minimal boolean polynomial for the given sets of points.

Given two sets of points zeros - evaluating to zero - and ones - evaluating to one -, compute the lexicographically minimal boolean polynomial satisfying these points.

**INPUT:**

• zeros - the set of interpolation points mapped to zero
• ones - the set of interpolation points mapped to one

**EXAMPLES:**

First we create a random-ish boolean polynomial.

```python
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing(6)
sage: f = a*b*c*e + a*d*e + a*f + b + c + e + f + 1
```

Now we find interpolation points mapping to zero and to one.

```python
sage: zeros = set([(1, 0, 1, 0, 0, 0), (1, 0, 0, 0, 1, 0),
........ (0, 0, 1, 1, 1, 1), (1, 0, 1, 1, 1, 1),
........ (0, 0, 0, 1, 1, 1), (0, 1, 1, 1, 1, 1),
........ (1, 1, 1, 1, 0, 1)])
sage: ones = set([(0, 0, 0, 0, 0, 0), (1, 0, 1, 0, 1, 0),
........ (0, 0, 0, 1, 1, 1), (1, 0, 0, 1, 0, 1),
........ (0, 0, 0, 0, 1, 1), (0, 1, 1, 0, 1, 1),
........ (0, 1, 1, 1, 1, 1), (1, 1, 1, 0, 1, 0)])
```

```python
sage: [f(*p) for p in zeros]
[0, 0, 0, 0, 0, 0, 0, 0]
sage: [f(*p) for p in ones]
[1, 1, 1, 1, 1, 1, 1, 1]
```

Finally, we find the lexicographically smallest interpolation polynomial using PolyBoRi.

```python
sage: g = B.interpolation_polynomial(zeros, ones); g
b*f + c + d*f + d + e*f + e + 1
```

```python
sage: [g(*p) for p in zeros]
[0, 0, 0, 0, 0, 0, 0, 0]
```

(continues on next page)
Alternatively, we can work with PolyBoRi’s native BooleSet’s. This example is from the PolyBoRi tutorial:

```
sage: B = BooleanPolynomialRing(4,"x0,x1,x2,x3")
sage: x = B.gen
sage: V=(x(0)+x(1)+x(2)+x(3)+1).set(); V
{{x0}, {x1}, {x2}, {x3}, {}}
sage: f=x(0)*x(1)+x(1)+x(2)+1
sage: z = f.zeros_in(V); z
{{x1}, {x2}}
sage: o = V.diff(z); o
{{x0}, {x3}, {}}
sage: B.interpolation_polynomial(z,o)
x1 + x2 + 1
```

ALGORITHM: Calls `interpolate_smallest_lex` as described in the PolyBoRi tutorial.

### `n_variables()`

Return the number of variables in this boolean polynomial ring.

**Examples:**

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: P.n_variables() 2

sage: P = BooleanPolynomialRing(1000, 'x')
sage: P.n_variables() 1000
```

**Note:** This is part of PolyBoRi’s native interface.

### `ngens()`

Return the number of variables in this boolean polynomial ring.

**Examples:**

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: P.ngens() 2

sage: P = BooleanPolynomialRing(1000, 'x')
sage: P.ngens() 1000
```

### `one()`

**Examples:**
sage: P.<x0,x1> = BooleanPolynomialRing(2)
sage: P.one()
1

random_element (degree=None, terms=None, choose_degree=False, vars_set=None)

Return a random boolean polynomial. Generated polynomial has the given number of terms, and at most
given degree.

INPUT:

- **degree** - maximum degree (default: 2 for len(var_set) > 1, 1 otherwise)
- **terms** – number of terms requested (default: 5). If more terms are requested than exist, then this
  parameter is silently reduced to the maximum number of available terms.
- **choose_degree** - choose degree of monomials randomly first, rather than monomials uniformly ran-
dom
- **vars_set** - list of integer indices of generators of self to use in the generated polynomial

EXAMPLES:

sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: P.random_element(degree=3, terms=4)
x*y*z + x*z + x + y*z
sage: P.random_element(degree=1, terms=2)
z + 1

In corner cases this function will return fewer terms by default:

sage: P = BooleanPolynomialRing(2, 'y')
sage: P.random_element()
y0*y1 + y0
sage: P = BooleanPolynomialRing(1, 'y')
sage: P.random_element()
y

We return uniformly random polynomials up to degree 2:

sage: B.<a,b,c,d> = BooleanPolynomialRing()
sage: B.random_element(terms=Infinity)
a*b + a*c + a*d + b*c + b*d + d

remove_var(order=None, *var)

Remove a variable or sequence of variables from this ring.

If **order** is not specified, then the subring inherits the term order of the original ring, if possible.

EXAMPLES:

sage: R.<x,y,z,w> = BooleanPolynomialRing()
sage: R.remove_var(z)
Boolean PolynomialRing in x, y, w
sage: R.remove_var(z,x)
Boolean PolynomialRing in y, w

(continues on next page)
Removing all variables results in the base ring:

```python
sage: R.remove_var(y,z,x)
Boolean PolynomialRing in w
```

If possible, the term order is kept:

```python
sage: R.<x,y,z,w> = BooleanPolynomialRing(order='deglex')
```

```python
sage: R.remove_var(y).term_order()  
Degree lexicographic term order
```

```python
sage: R.<x,y,z,w> = BooleanPolynomialRing(order='lex')
```

```python
sage: R.remove_var(y).term_order()  
Lexicographic term order
```

Be careful with block orders when removing variables:

```python
sage: R.<x,y,z,u,v> = BooleanPolynomialRing(order='deglex(2),deglex(3)')
sage: R.remove_var(x,y,z)
```

Traceback (most recent call last):
...

```python
ValueError: impossible to use the original term order (most likely because it was a block order). Please specify the term order for the subring
```

```python
sage: R.<x,y,z,w> = BooleanPolynomialRing(order='deglex')
```

```python
sage: R.remove_var(x,y,z, order='deglex')
```

Boolean PolynomialRing in u, v

---

**variable(i=0)**

Return the i-th generator of this boolean polynomial ring.

**INPUT:**

- i - an integer or a boolean monomial in one variable

**EXAMPLES:**

```python
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: P.variable()  
x
```

```python
sage: P.variable(2)  
z
```

```python
sage: m = x.monomials()[0]
sage: P.variable(m)  
x
```

---

**zero()**

**EXAMPLES:**

```python
sage: P.<x0,x1> = BooleanPolynomialRing(2)
sage: P.zero()  
0
```

---

**class** `sage.rings.polynomial.pbori.pbori.BooleanPolynomialVector`

**Bases:** `object`

A vector of boolean polynomials.

**EXAMPLES:**

---

7.1. Boolean Polynomials
```python
sage: B.<a,b,c,d,e,f> = Boolean PolynomialRing()
sage: from sage.rings.polynomial.pbori.pbori import Boolean Polynomial Vector
sage: l = [B.random_element() for _ in range(3)]
sage: v = Boolean Polynomial Vector(l)
sage: len(v)
3
sage: v[0]
a*b + a + b*e + c*d + e*f
sage: list(v)
[a*b + a + b*e + c*d + e*f, a*d + c*d + d*f + e + f, a*c + a*e + b*c + c*f + f]
```

**append (el)**

Append the element el to this vector.

```
sage: B.<a,b,c,d,e,f> = Boolean Polynomial Ring()
sage: v = Boolean Polynomial Vector()
sage: for i in range(5):
    ....: v.append(B.random_element())
sage: list(v)
[a*b + a + b*e + c*d + e*f, a*d + c*d + d*f + e + f, a*c + a*e + b*c + c*f + f, ...
 →a*c + a*d + a*e + a*f + b*e, b*c + b*d + c*d + c + 1]
```

**class sage.rings.polynomial.pbori.pbori.Boolean Polynomial Vector Iterator**

Bases: object

**class sage.rings.polynomial.pbori.pbori.CCudd Navigator**

Bases: object

**constant()**

**else_branch()**

**terminal_one()**

**then_branch()**

**value()**

**class sage.rings.polynomial.pbori.pbori.FGLM Strategy**

Bases: object

Strategy object for the FGLM algorithm to translate from one Groebner basis with respect to a term ordering A to another Groebner basis with respect to a term ordering B.

**main()**

Execute the FGLM algorithm.

```
sage: from sage.rings.polynomial.pbori.pbori import *
sage: B.<x,y,z> = Boolean Polynomial Ring()
sage: ideal = Boolean Polynomial Vector([x+z, y+z])
sage: list(ideal)
[x + z, y + z]
```
sage: old_ring = B
sage: new_ring = B.clone(ordering=dp_asc)

sage: list(FGLMStrategy(old_ring, new_ring, ideal).main())
[y + x, z + x]

class sage.rings.polynomial.pbori.pbori.GroebnerStrategy
Bases: object

A Groebner strategy is the main object to control the strategy for computing Groebner bases.

Note: This class is mainly used internally.

add_as_you_wish(p)
Add a new generator but let the strategy object decide whether to perform immediate interreduction.

INPUT:
• p - a polynomial

EXAMPLES:

sage: from sage.rings.polynomial.pbori.pbori import *
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: gbs = GroebnerStrategy(B)
sage: gbs.add_as_you_wish(a + b)
sage: list(gbs)
[a + b]
sage: gbs.add_as_you_wish(a + c)

Note that nothing happened immediately but that the generator was indeed added:

sage: list(gbs)
[a + b]
sage: gbs.symmGB_F2()
sage: list(gbs)
[a + c, b + c]

add_generator(p)
Add a new generator.

INPUT:
• p - a polynomial

EXAMPLES:

sage: from sage.rings.polynomial.pbori.pbori import *
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: gbs = GroebnerStrategy(B)
sage: gbs.add_generator(a + b)
sage: list(gbs)
[a + b]
sage: gbs.add_generator(a + c)
Traceback (most recent call last):
add_generator_delayed(p)
Add a new generator but do not perform interreduction immediately.

INPUT:
• p - a polynomial

EXAMPLES:

```sage
from sage.rings.polynomial.pbori.pbori import *
B.<a,b,c,d,e,f> = BooleanPolynomialRing()
gbs = GroebnerStrategy(B)
gbs.add_generator(a + b)
list(gbs)
[a + b]
gbs.add_generator_delayed(a + c)
list(gbs)
[a + b]
list(gbs.all_generators())
[a + b, a + c]
```

all_generators()

EXAMPLES:

```sage
from sage.rings.polynomial.pbori.pbori import *
B.<a,b,c,d,e,f> = BooleanPolynomialRing()
gbs = GroebnerStrategy(B)
gbs.add_as_you_wish(a + b)
list(gbs)
[a + b]
gbs.add_as_you_wish(a + c)
list(gbs)
[a + b]
list(gbs.all_generators())
[a + b, a + c]
```

all_spolys_in_next_degree()
clean_top_by_chain_criterion()

contains_one()
Return True if 1 is in the generating system.

EXAMPLES:

We construct an example which contains 1 in the ideal spanned by the generators but not in the set of generators:

```sage
B.<a,b,c,d,e,f> = BooleanPolynomialRing()
from sage.rings.polynomial.pbori.pbori import GroebnerStrategy
```
sage: gb = GroebnerStrategy(B)
sage: gb.add_generator(a*c + a*f + d*f + d + f)
sage: gb.add_generator(b*c + b*e + c + d + 1)
sage: gb.add_generator(a*f + a + c + d + 1)
sage: gb.add_generator(a*d + a*e + b*e + c + f)
sage: gb.add_generator(b*d + c + d*f + e + f)
sage: gb.add_generator(a*b + b + c*e + e + 1)
sage: gb.add_generator(a + b + c*d + c*e + 1)
sage: gb.contains_one()
False

Still, we have that:

sage: from sage.rings.polynomial.pbori import groebner_basis
sage: groebner_basis(gb)
[1]

faugere_step_dense(v)
Reduces a vector of polynomials using linear algebra.

INPUT:
• v - a boolean polynomial vector

EXAMPLES:

sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: from sage.rings.polynomial.pbori.pbori import GroebnerStrategy
sage: gb = GroebnerStrategy(B)
sage: gb.add_generator(a*c + a*f + d*f + d + f)
sage: gb.add_generator(b*c + b*e + c + d + 1)
sage: gb.add_generator(a*f + a + c + d + 1)
sage: gb.add_generator(a*d + a*e + b*e + c + f)
sage: gb.add_generator(b*d + c + d*f + e + f)
sage: gb.add_generator(a*b + b + c*e + e + 1)
sage: gb.add_generator(a + b + c*d + c*e + 1)

sage: from sage.rings.polynomial.pbori.pbori import BooleanPolynomialVector
sage: V= BooleanPolynomialVector([b*d, a*b])
sage: list(gb.faugere_step_dense(V))
[b + c*e + e + 1, c + d*f + e + f]

implications(i)
Compute “useful” implied polynomials of i-th generator, and add them to the strategy, if it finds any.

INPUT:
• i - an index

ll_reduce_all()
Use the built-in ll-encoded BooleSet of polynomials with linear lexicographical leading term, which coincides with leading term in current ordering, to reduce the tails of all polynomials in the strategy.

minimalize()
Return a vector of all polynomials with minimal leading terms.

7.1. Boolean Polynomials
minimalize_and_tail_reduce()  
Return a vector of all polynomials with minimal leading terms and do tail reductions.

Note: Use this function if strat contains a GB.

next_spoly()  
nf(p)  
Compute the normal form of p with respect to the generating set.

INPUT:  
• p - a boolean polynomial

EXAMPLES:

```sage
sage: P = PolynomialRing(GF(2),10, 'x')
sage: B = BooleanPolynomialRing(10,'x')
sage: I = sage.rings.ideal.Cyclic(P)
sage: I = B.ideal([B(f) for f in I.gens()])
sage: gb = I.groebner_basis()
sage: from sage.rings.polynomial.pbori.pbori import GroebnerStrategy
sage: G = GroebnerStrategy(B)
sage: _ = [G.add_generator(f) for f in gb]
sage: G.nf(gb[0])
0
sage: G.nf(gb[0] + 1)
1
sage: G.nf(gb[0]*gb[1])
0
sage: G.nf(gb[0]*B.gen(1))
0
```

Note: The result is only canonical if the generating set is a Groebner basis.

npairs()  
reduction_strategy  
select(m)  
Return the index of the generator which can reduce the monomial m.

INPUT:  
• m - a BooleanMonomial

EXAMPLES:

```sage
sage: B.<a,b,c,d,e> = BooleanPolynomialRing()
sage: f = B.random_element()
```

(continues on next page)
sage: g = B.random_element()
sage: from sage.rings.polynomial.pbori.pbori import GroebnerStrategy
sage: strat = GroebnerStrategy(B)
sage: strat.add_generator(f)
sage: strat.add_generator(g)
sage: strat.select(f.lm())
@
sage: strat.select(g.lm())
1
sage: strat.select(e.lm())
-1

small_spols_in_next_degree(f, n)
some_spols_in_next_degree(n)
suggest_plugin_variable()
symmGB_F2()

Compute a Groebner basis for the generating system.

Note: This implementation is out of date, but it will revived at some point in time. Use the 
groebner_basis() function instead.

top_sugar()

variable_has_value(v)

Computes, whether there exists some polynomial of the form \(v + c\) in the Strategy – where \(c\) is a constant – in the list of generators.

INPUT:

• \(v\) - the index of a variable

EXAMPLES:

sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: from sage.rings.polynomial.pbori.pbori import GroebnerStrategy
sage: gb = GroebnerStrategy(B)
sage: gb.add_generator(a*c + a*f + d*f + d + f)
sage: gb.add_generator(b*c + b*e + c + d + 1)
sage: gb.add_generator(a*f + a + c + d + 1)
sage: gb.add_generator(a*d + a*e + b*e + c + f)
sage: gb.add_generator(b*d + c + d*f + e + f)
sage: gb.add_generator(a*b + b + c*e + e + 1)
sage: gb.variable_has_value(0)
False

sage: gb = GroebnerStrategy(B)
sage: _ = [gb.add_generator(f) for f in g]

7.1. Boolean Polynomials
class sage.rings.polynomial.pboringen.PolynomialConstruct
    Bases: object
    Implements PolyBoRi's Polynomial() constructor.

lead(x)
    Return the leading monomial of boolean polynomial x, with respect to the order of parent ring.

    EXAMPLES:
    sage: from sage.rings.polynomial.pboringen import *
    sage: B.<a,b,c> = BooleanPolynomialRing()
    sage: PolynomialConstruct().lead(a)
    a

class sage.rings.polynomial.pboringen.PolynomialFactory
    Bases: object
    Implements PolyBoRi's Polynomial() constructor and a polynomial factory for given rings.

lead(x)
    Return the leading monomial of boolean polynomial x, with respect to the order of parent ring.

    EXAMPLES:
    sage: from sage.rings.polynomial.pboringen import *
    sage: B.<a,b,c> = BooleanPolynomialRing()
    sage: PolynomialFactory().lead(a)
    a

class sage.rings.polynomial.pboringen.ReductionStrategy
    Bases: object
    Functions and options for boolean polynomial reduction.

add_generator(p)
    Add the new generator p to this strategy.

    INPUT:
• \( p \) - a boolean polynomial.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.pbori.pbori import *
sage: B.<x,y,z> = BooleanPolynomialRing()
sage: red = ReductionStrategy(B)
sage: red.add_generator(x)
sage: list([f.p for f in red])
\[ x \]
```

**can_rewrite**\((p)\)

Return True if \( p \) can be reduced by the generators of this strategy.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.pbori.pbori import *
sage: B.<a,b,c,d> = BooleanPolynomialRing()
sage: red = ReductionStrategy(B)
sage: red.add_generator(a*b + c + 1)
sage: red.add_generator(b*c + d + 1)
sage: red.can_rewrite(a*b + a)
True
sage: red.can_rewrite(b + c)
False
sage: red.can_rewrite(a*d + b*c + d + 1)
True
```

**cheap_reductions**\((p)\)

Perform ‘cheap’ reductions on \( p \).

**INPUT:**

• \( p \) - a boolean polynomial

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.pbori.pbori import *
sage: B.<a,b,c,d> = BooleanPolynomialRing()
sage: red = ReductionStrategy(B)
sage: red.add_generator(a*b + c + 1)
sage: red.add_generator(b*c + d + 1)
sage: red.add_generator(a)
sage: red.cheap_reductions(a*b + a)
\emptyset
sage: red.cheap_reductions(b + c)
b + c
sage: red.cheap_reductions(a*d + b*c + d + 1)
b*c + d + 1
```

**head_normal_form**\((p)\)

Compute the normal form of \( p \) with respect to the generators of this strategy but do not perform tail any reductions.

**INPUT:**

• \( p \) – a polynomial

**EXAMPLES:**
```python
# sage: from sage.rings.polynomial.pbori.pbori import *
# sage: B.<x,y,z> = BooleanPolynomialRing()
# sage: red = ReductionStrategy(B)
# sage: red.opt_red_tail = True
# sage: red.add_generator(x + y + 1)
# sage: red.add_generator(y*z + z)
# sage: red.head_normal_form(x + y*z)
# y + z + 1
# sage: red.nf(x + y*z)
# y + z + 1
```

**nf(p)**

Compute the normal form of \( p \) w.r.t. to the generators of this reduction strategy object.

**EXAMPLES:**

```python
# sage: from sage.rings.polynomial.pbori.pbori import *
# sage: B.<x,y,z> = BooleanPolynomialRing()
# sage: red = ReductionStrategy(B)
# sage: red.add_generator(x + y + 1)
# sage: red.add_generator(y*z + z)
# sage: red.nf(x)
# y + 1
# sage: red.nf(y*z + x)
# y + z + 1
```

**reduced_normal_form(p)**

Compute the normal form of \( p \) with respect to the generators of this strategy and perform tail reductions.

**INPUT:**

- \( p \) - a polynomial

**EXAMPLES:**

```python
# sage: from sage.rings.polynomial.pbori.pbori import *
# sage: B.<x,y,z> = BooleanPolynomialRing()
# sage: red = ReductionStrategy(B)
# sage: red.add_generator(x + y + 1)
# sage: red.add_generator(y*z + z)
# sage: red.reduced_normal_form(x)
# y + 1
# sage: red.reduced_normal_form(y*z + x)
# y + z + 1
```

```python
sage.rings.polynomial.pbori.pbori.TermOrder_from_pb_order(n, order, blocks)
```

**class** `sage.rings.polynomial.pbori.pbori.VariableBlock`

Bases: object

**class** `sage.rings.polynomial.pbori.pbori.VariableConstruct`

Bases: object
Implements PolyBoRi’s Variable() constructor.

```python
class sage.rings.polynomial.pbori.pbori.VariableFactory
    Bases: object
    Implements PolyBoRi’s Variable() constructor and a variable factory for given ring
```

```python
sage.rings.polynomial.pbori.pbori.add_up_polynomials(v, init)
Add up all entries in the vector v.
```

**INPUT:**
- `v` - a vector of boolean polynomials

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.pbori.pbori import *
sage: B.<a,b,c,d> = BooleanPolynomialRing()
sage: v = BooleanPolynomialVector()
sage: l = [B.random_element() for _ in range(5)]
sage: _ = [v.append(e) for e in l]
sage: add_up_polynomials(v, B.zero())
a*d + b*c + b*d + c + 1
```

```python
sage.rings.polynomial.pbori.pbori.contained_vars(m)
sage.rings.polynomial.pbori.pbori.easy_linear_factors(p)
sage.rings.polynomial.pbori.pbori.gauss_on_polys(inp)
Perform Gaussian elimination on the input list of polynomials.
```

**INPUT:**
- `inp` - an iterable

**EXAMPLES:**

```python
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: from sage.rings.polynomial.pbori.pbori import *
sage: l = [B.random_element() for _ in range(B.ngens())]
sage: A, v = Sequence(l, B).coefficient_matrix()
sage: A
[1 0 0 0 1 0 0 1 1 0 0 0 0 1 0 0 0]
[0 0 1 0 0 0 0 0 1 0 0 1 0 1 1 0]
[0 1 0 1 0 0 1 0 0 1 0 0 0 0 1 0]
[0 1 1 1 0 0 1 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 1 1 0 1 0 1 0 0 0 0 1]
[0 1 0 0 1 0 1 0 0 0 0 1 0 0 0 1]
sage: e = gauss_on_polys(l)
sage: E, v = Sequence(e, B).coefficient_matrix()
sage: E
[1 0 0 0 1 0 0 1 1 0 0 0 0 1 0 0 0]
[0 1 0 0 0 0 0 1 1 0 1 0 1 0 1 1]
[0 0 1 0 0 0 0 0 1 0 0 1 0 1 1 0]
[0 0 0 1 0 0 1 1 0 0 1 0 1 0 1 1]
[0 0 0 0 1 0 0 1 1 0 1 0 1 0 1 1]
```

(continues on next page)
sage.rings.polynomial.pbori.pbori.get_var_mapping(ring, other)
Return a variable mapping between variables of other and ring. When other is a parent object, the mapping defines images for all variables of other. If it is an element, only variables occurring in other are mapped.
Raises NameError if no such mapping is possible.

EXAMPLES:

```python
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: R.<z,y> = QQ[

```

sage: sage.rings.polynomial.pbori.pbori.get_var_mapping(P, R)
```python
[z, y]
```

sage: sage.rings.polynomial.pbori.pbori.get_var_mapping(P, z^2)
```python
[z, None]
```

```python
sage: R.<z,x> = BooleanPolynomialRing(2)
sage: sage.rings.polynomial.pbori.pbori.get_var_mapping(P, R)
``` python
```python
[z, x]
```

```python
sage: sage.rings.polynomial.pbori.pbori.if_then_else(root, a, b)
```
The opposite of navigating down a ZDD using navigators is to construct new ZDDs in the same way, namely giving their else- and then-branch as well as the index value of the new node.

**INPUT:**

- root - a variable
- a - the if branch, a BooleSet or a BoolePolynomial
- b - the else branch, a BooleSet or a BoolePolynomial

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.pbori.pbori import if_then_else
sage: B = BooleanPolynomialRing(6, 'x')
sage: x0, x1, x2, x3, x4, x5 = B.gens()
sage: f0 = x2*x3+x3
sage: f1 = x4
sage: if_then_else(x1, f0, f1)
``` python
```python
{{x1,x2,x3}, {x1,x3}, {x4}}
```

```python
sage: if_then_else(x1.lm().index(), f0, f1)
``` python
```python
{{x1,x2,x3}, {x1,x3}, {x4}}
```
sage: if_then_else(x5, f0, f1)
Traceback (most recent call last):
...
IndexError: index of root must be less than the values of roots of the branches.

sage.rings.polynomial.pbori.pbori.interpolate(zero, one)
Interpolate a polynomial evaluating to zero on `zero` and to one on `ones`.

**INPUT:**
- `zero` - the set of `zero`
- `one` - the set of `ones`

**EXAMPLES:**

```python
sage: B = BooleanPolynomialRing(4,"x0,x1,x2,x3")
sage: x = B.gen
sage: from sage.rings.polynomial.pbori.interpolate import *
sage: V=(x(0)+x(1)+x(2)+x(3)+1).set()
sage: V
{{x0}, {x1}, {x2}, {x3}, {}}
sage: f=x(0)*x(1)+x(1)+x(2)+1
sage: nf_lex_points(f, V)
x1 + x2 + 1
sage: z=f.zeros_in(V)
sage: z
{{x1}, {x2}}
sage: o=V.diff(z)
sage: o
{{x0}, {x3}, {}}
sage: interpolate(z,o)
x0*x1*x2 + x0*x1 + x0*x2 + x1*x2 + x1 + x2 + 1
```

sage.rings.polynomial.pbori.pbori.interpolate_smallest_lex(zero, one)
Interpolate the lexicographical smallest polynomial evaluating to zero on `zero` and to one on `ones`.

**INPUT:**
- `zero` - the set of `zeros`
- `one` - the set of `ones`

**EXAMPLES:**

Let $V$ be a set of points in $\mathbb{F}_2^n$ and $f$ a Boolean polynomial. $V$ can be encoded as a BooleSet. Then we are interested in the normal form of $f$ against the vanishing ideal of $V : I(V)$.

It turns out, that the computation of the normal form can be done by the computation of a minimal interpolation polynomial, which takes the same values as $f$ on $V$:

```python
sage: B = BooleanPolynomialRing(4,"x0,x1,x2,x3")
sage: x = B.gen
```
We take \( V = \{ e_0, e_1, e_2, e_3, 0 \} \), where \( e_i \) describes the i-th unit vector. For our considerations it does not play any role, if we suppose \( V \) to be embedded in \( \mathbb{F}_2 \) or a vector space of higher dimension:

\[
\text{sage: } V = (x(0) + x(1) + x(2) + x(3) + 1).set()
\]

\[
\text{sage: } V
\]
\[
\{\{x0\}, \{x1\}, \{x2\}, \{x3\}, {}\}
\]

\[
\text{sage: } f = x(0) \cdot x(1) + x(1) + x(2) + 1
\]

\[
\text{sage: } \text{nf_lex_points}(f, V)
\]
\[
x1 + x2 + 1
\]

In this case, the normal form of \( f \) w.r.t. the vanishing ideal of \( V \) consists of all terms of \( f \) with degree smaller or equal to 1.

It can be easily seen, that this polynomial forms the same function on \( V \) as \( f \). In fact, our computation is equivalent to the direct call of the interpolation function \text{interpolate_smallest_lex}, which has two arguments: the set of interpolation points mapped to zero and the set of interpolation points mapped to one:

\[
\text{sage: } z = f.\text{zeros_in}(V)
\]
\[
\text{sage: } z
\]
\[
\{\{x1\}, \{x2\}\}
\]

\[
\text{sage: } o = V.\text{diff}(z)
\]
\[
\text{sage: } o
\]
\[
\{\{x0\}, \{x3\}, {}\}
\]

\[
\text{sage: } \text{interpolate_smallest_lex}(z, o)
\]
\[
x1 + x2 + 1
\]

\[
\text{sage.rings.polynomial.pbori.pbori.ll_red_nf_noredsb}(p, \text{reductors})
\]

Redude the polynomial \( p \) by the set of \text{reductors} with linear leading terms.

**INPUT:**

- \( p \) - a boolean polynomial
- \( \text{reductors} \) - a boolean set encoding a Groebner basis with linear leading terms.

**EXAMPLES:**

\[
\text{sage: from sage.rings.polynomial.pbori.pbori import ll_red_nf_noredsb}
\]
\[
\text{sage: B.<a,b,c,d> = BooleanPolynomialRing()}
\]
\[
\text{sage: p = a*b + c + d + 1}
\]
\[
\text{sage: f, g = a + c + 1, b + d + 1}
\]
\[
\text{sage: reductors = f.set().union( g.set() )}
\]
\[
\text{sage: ll_red_nf_noredsb}(p, \text{reductors})
\]
\[
b*c + b*d + c + d + 1
\]

\[
\text{sage.rings.polynomial.pbori.pbori.ll_red_nf_noredsb_single_recursive_call}(p, \text{reductors})
\]

Redude the polynomial \( p \) by the set of \text{reductors} with linear leading terms.

\( \text{ll_red_nf_noredsb_single_recursive}() \) call has the same specification as \( \text{ll_red_nf_noredsb}() \), but a different implementation: It is very sensitive to the ordering of variables, however it has the property, that it needs just one recursive call.
INPUT:

- \( p \) - a boolean polynomial
- \( \text{reductors} \) - a boolean set encoding a Groebner basis with linear leading terms.

EXAMPLES:

```python
sage: from sage.rings.polynomial.pbori.pbori import ll_red_nf_noredsb_single_recursive_call
sage: B.<a,b,c,d> = BooleanPolynomialRing()
sage: p = a*b + c + d + 1
sage: f, g = a + c + 1, b + d + 1
sage: reductors = f.set().union(g.set())
```

```python
sage: ll_red_nf_noredsb_single_recursive_call(p, reductors)
b*c + b*d + c + d + 1
```

```
sage.rings.polynomial.pbori.pbori.ll_red_nf_redsb(p, reductors)
```

Reduces the polynomial \( p \) by the set of \( \text{reductors} \) with linear leading terms. It is assumed that the set \( \text{reductors} \) is a reduced Groebner basis.

INPUT:

- \( p \) - a boolean polynomial
- \( \text{reductors} \) - a boolean set encoding a reduced Groebner basis with linear leading terms.

EXAMPLES:

```python
sage: from sage.rings.polynomial.pbori.pbori import ll_red_nf_redsb
sage: B.<a,b,c,d> = BooleanPolynomialRing()
```

```python
sage: p = a*b + c + d + 1
```

```python
sage: f, g = a + c + 1, b + d + 1
```

```python
sage: reductors = f.set().union(g.set())
```

```python
sage: ll_red_nf_redsb(p, reductors)
b*c + b*d + c + d + 1
```

```
sage.rings.polynomial.pbori.pbori.map_every_x_to_x_plus_one(p)
```

Maps every variable \( x_i \) in this polynomial to \( x_i + 1 \).

EXAMPLES:

```python
sage: B.<a,b,z> = BooleanPolynomialRing(3)
```

```python
sage: f = a*b + z + 1; f
a*b + z + 1
```

```python
sage: from sage.rings.polynomial.pbori.pbori import map_every_x_to_x_plus_one
```

```python
sage: map_every_x_to_x_plus_one(f)
a*b + a + b + z + 1
```

```
sage.rings.polynomial.pbori.pbori.mod_mon_set(a_s, v_s)
sage.rings.polynomial.pbori.pbori.mod_var_set(a, v)
sage.rings.polynomial.pbori.pbori.mult_fact_sim_C(v, ring)
sage.rings.polynomial.pbori.pbori.nf3(s, p, m)
sage.rings.polynomial.pbori.pbori.parallel_reduce(inp, strat, average_steps, delay_f)
```

7.1. Boolean Polynomials
sage.rings.polynomial.pbori.pbori.random_set(variables, length)
Return a random set of monomials with length elements with each element in the variables variables.

EXAMPLES:
```
sage: from sage.rings.polynomial.pbori import random_set, set_random_seed
sage: B.<a,b,c,d,e> = BooleanPolynomialRing()
sage: (a*b*c*d).lm()
a*b*c*d
sage: set_random_seed(1337)
sage: random_set((a*b*c*d).lm(),10)
{{a,b,c,d}, {a,b}, {a,c,d}, {a,c}, {b,c,d}, {b,d}, {b}, {c,d}, {c}, {d}}
```

sage.rings.polynomial.pbori.pbori.recursively_insert(n, ind, m)

sage.rings.polynomial.pbori.pbori.red_tail(s, p)
Perform tail reduction on p using the generators of s.

INPUT:
- s - a reduction strategy
- p - a polynomial

EXAMPLES:
```
sage: from sage.rings.polynomial.pbori import *
sage: B.<x,y,z> = BooleanPolynomialRing()
sage: red = ReductionStrategy(B)
sage: red.add_generator(x + y + 1)
sage: red.add_generator(y*z + z)
sage: red_tail(red,x)
x
sage: red_tail(red,x*y + x)
x*y + y + 1
```

sage.rings.polynomial.pbori.pbori.set_random_seed(seed)
Set the PolyBoRi random seed to seed.

EXAMPLES:
```
sage: from sage.rings.polynomial.pbori import random_set, set_random_seed
sage: B.<a,b,c,d,e> = BooleanPolynomialRing()
sage: (a*b*c*d).lm()
a*b*c*d
sage: set_random_seed(1337)
sage: random_set((a*b*c*d).lm(),2)
{{b}, {c}}
sage: random_set((a*b*c*d).lm(),2)
{{a,c,d}, {c}}
sage: set_random_seed(1337)
sage: random_set((a*b*c*d).lm(),2)
{{b}, {c}}
sage: random_set((a*b*c*d).lm(),2)
{{a,c,d}, {c}}
```
sage.rings.polynomial.pbori.pbori.substitute_variables(parent, vec, poly)

var(i) is replaced by vec[i] in poly.

EXAMPLES:

```python
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: f = a*b + c + 1
sage: from sage.rings.polynomial.pbori.pbori import substitute_variables
sage: substitute_variables(B, [a,b,c],f)
a*b + c + 1
sage: substitute_variables(B, [a+1,b,c],f)
a*b + b + c + 1
sage: substitute_variables(B, [a+1,b+1,c],f)
a*b + a + b + c
sage: substitute_variables(B, [a+1,b+1,B(0)],f)
a*b + a + b
```

Substitution is also allowed with different rings:

```python
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: f = a*b + c + 1
sage: B.<w,x,y,z> = BooleanPolynomialRing(order='deglex')
sage: from sage.rings.polynomial.pbori.pbori import substitute_variables
sage: substitute_variables(B, [x,y,z], f) * w
w*x*y + w*z + w
```

top_index(s)

Return the highest index in the parameter s.

INPUT:

- s - BooleSet, BooleMonomial, BoolePolynomial

EXAMPLES:

```python
sage: B.<x,y,z> = BooleanPolynomialRing(3)
sage: from sage.rings.polynomial.pbori.pbori import top_index
sage: top_index(x.lm())
0
sage: top_index(y*z)
1
sage: top_index(x + 1)
0
```

unpickle_BooleanPolynomial(ring, string)

Unpickle boolean polynomials

EXAMPLES:

```python
sage: T = TermOrder('deglex',2)+TermOrder('deglex',2)
sage: P.<a,b,c,d> = BooleanPolynomialRing(4,order=T)
sage: loads(dumps(a+b)) == a+b # indirect doctest
True
```

unpickle_BooleanPolynomial0(ring, l)

Unpickle boolean polynomials.

7.1. Boolean Polynomials 641
EXAMPLES:

```python
sage: T = TermOrder('deglex',2)+TermOrder('deglex',2)
sage: P.<a,b,c,d> = BooleanPolynomialRing(4,order=T)
sage: loads(dumps(a+b)) == a+b  # indirect doctest
True
```

```
Unpickle boolean polynomial rings.
```

EXAMPLES:

```python
sage: T = TermOrder('deglex',2)+TermOrder('deglex',2)
sage: P.<a,b,c,d> = BooleanPolynomialRing(4,order=T)
sage: loads(dumps(P)) == P  # indirect doctest
True
```

```
Return a BooleSet encoding on which points from s the polynomial pol evaluates to zero.
```

INPUT:

- **pol** - a boolean polynomial
- **s** - a set of points encoded as a BooleSet

EXAMPLES:

```python
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: f = a*b + a*c + d + b

Now we create a set of points:

```python
sage: s = a*b + a*b*c + c*d + b*c
sage: s = s.set(); s
{{a,b,c}, {a,b}, {b,c}, {c,d}}
```

This encodes the points (1,1,1,0), (1,1,0,0), (0,0,1,1) and (0,1,1,0). But of these only (1,1,0,0) evaluates to zero.

```python
sage: from sage.rings.polynomial.pbori.pbori import zeros
sage: zeros(f, s)
{{a,b}}
```

For comparison we work with tuples:

```python
sage: f.zeros_in([(1,1,1,0), (1,1,0,0), (0,0,1,1), (0,1,1,0)])
((1, 1, 0, 0),)
```

```
Chapter 7. Boolean Polynomials
```
INDICES AND TABLES

• Index
• Module Index
• Search Page
<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sage.rings.fraction_field</td>
<td>497</td>
</tr>
<tr>
<td>sage.rings.fraction_field_element</td>
<td>503</td>
</tr>
<tr>
<td>sage.rings.fraction_field_FpT</td>
<td>507</td>
</tr>
<tr>
<td>sage.rings.invariants.invariant_theory</td>
<td>435</td>
</tr>
<tr>
<td>sage.rings.invariants.reconstruction</td>
<td>480</td>
</tr>
<tr>
<td>sage.rings.monomials</td>
<td>435</td>
</tr>
<tr>
<td>sage.rings.polynomial.complex_roots</td>
<td>219</td>
</tr>
<tr>
<td>sage.rings.polynomial.convolution</td>
<td>248</td>
</tr>
<tr>
<td>sage.rings.polynomial.cyclotomic</td>
<td>249</td>
</tr>
<tr>
<td>sage.rings.polynomial.flatten</td>
<td>432</td>
</tr>
<tr>
<td>sage.rings.polynomial.hilbert</td>
<td>431</td>
</tr>
<tr>
<td>sage.rings.polynomial.ideal</td>
<td>222</td>
</tr>
<tr>
<td>sage.rings.polynomial.infinite_polynomial_element</td>
<td>559</td>
</tr>
<tr>
<td>sage.rings.polynomial.infinite_polynomial_ring</td>
<td>549</td>
</tr>
<tr>
<td>sage.rings.polynomial.laurent_polynomial</td>
<td>525</td>
</tr>
<tr>
<td>sage.rings.polynomial.laurent_polynomial_ring</td>
<td>517</td>
</tr>
<tr>
<td>sage.rings.polynomial.multi_polynomial</td>
<td>283</td>
</tr>
<tr>
<td>sage.rings.polynomial.multi_polynomial_element</td>
<td>309</td>
</tr>
<tr>
<td>sage.rings.polynomial.multi_polynomial_ideal</td>
<td>324</td>
</tr>
<tr>
<td>sage.rings.polynomial.multi_polynomial_ideal_libsingular</td>
<td>418</td>
</tr>
<tr>
<td>sage.rings.polynomial.multi_polynomial_libsingular</td>
<td>390</td>
</tr>
<tr>
<td>sage.rings.polynomial.multi_polynomial_ring</td>
<td>305</td>
</tr>
<tr>
<td>sage.rings.polynomial.multi_polynomial_ring_base</td>
<td>273</td>
</tr>
<tr>
<td>sage.rings.polynomial.multi_polynomial_sequence</td>
<td>375</td>
</tr>
<tr>
<td>sage.rings.polynomial.omega</td>
<td>544</td>
</tr>
<tr>
<td>sage.rings.polynomial.padics.polynomial_padic</td>
<td>176</td>
</tr>
<tr>
<td>sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense</td>
<td>179</td>
</tr>
<tr>
<td>sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense_583</td>
<td>186</td>
</tr>
<tr>
<td>sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense_583</td>
<td>583</td>
</tr>
<tr>
<td>sage.rings.polynomial.polydict</td>
<td>419</td>
</tr>
<tr>
<td>sage.rings.polynomial.polynomial_compiled</td>
<td>247</td>
</tr>
<tr>
<td>sage.rings.polynomial.polynomial_element</td>
<td>31</td>
</tr>
<tr>
<td>sage.rings.polynomial.polynomial_element_generic</td>
<td>112</td>
</tr>
<tr>
<td>sage.rings.polynomial.polynomial_fateman</td>
<td>248</td>
</tr>
<tr>
<td>sage.rings.polynomial.polynomial_gf2x</td>
<td>121</td>
</tr>
<tr>
<td>sage.rings.polynomial.polynomial_integer_dense_flint</td>
<td>128</td>
</tr>
<tr>
<td>sage.rings.polynomial.polynomial_integer_dense_ntl</td>
<td>137</td>
</tr>
<tr>
<td>sage.rings.polynomial.polynomial_modn_dense_ntl</td>
<td>162</td>
</tr>
<tr>
<td>sage.rings.polynomial.polynomial_number_field</td>
<td>126</td>
</tr>
<tr>
<td>sage.rings.polynomial.polynomial_quotient_ring</td>
<td>223</td>
</tr>
<tr>
<td>sage.rings.polynomial.polynomial_quotient_ring_element</td>
<td>243</td>
</tr>
<tr>
<td>sage.rings.polynomial.polynomial_rational_flint</td>
<td>142</td>
</tr>
<tr>
<td>sage.rings.polynomial.polynomial_real_mpfr_dense</td>
<td>172</td>
</tr>
<tr>
<td>sage.rings.polynomial.polynomial_ring</td>
<td>9</td>
</tr>
<tr>
<td>sage.rings.polynomial.polynomial_ring_constructor</td>
<td>175</td>
</tr>
<tr>
<td>sage.rings.polynomial.polynomial_ring_homomorphism</td>
<td>30</td>
</tr>
<tr>
<td>sage.rings.polynomial.polynomial_singular_interface</td>
<td>175</td>
</tr>
<tr>
<td>sage.rings.polynomial.polynomial_zmod_flint</td>
<td>154</td>
</tr>
<tr>
<td>sage.rings.polynomial.polynomial_zz_pex</td>
<td>186</td>
</tr>
<tr>
<td>sage.rings.polynomial.real_roots</td>
<td>222</td>
</tr>
<tr>
<td>sage.rings.polynomial.symmetric_ideal</td>
<td>567</td>
</tr>
<tr>
<td>sage.rings.polynomial.symmetric_reduction</td>
<td>576</td>
</tr>
<tr>
<td>sage.rings.polynomial.term_order</td>
<td>253</td>
</tr>
<tr>
<td>sage.rings.polynomial.toy_buchberger</td>
<td>483</td>
</tr>
</tbody>
</table>
sage.rings.polynomial.toy_d_basis, 492
sage.rings.polynomial.toy_variety, 489
Symbols

_add_ (sage.rings.polynomial.polynomial_element.Polynomial method), 32
_rmul_ (sage.rings.polynomial.polynomial_element.Polynomial method), 32
_rmul_ (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 129
_rmul_ (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl method), 167
_rmul_ (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl method), 167
_rmul_ (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 143
_rmul_ (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 158

_lmul_ (sage.rings.polynomial.polynomial_element.Polynomial method), 32
_lmul_ (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 129
_lmul_ (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl method), 167
_lmul_ (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 143
_lmul_ (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 157

_mul_ (sage.rings.polynomial.polynomial_element.Polynomial method), 32
_mul_ (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 129
_mul_ (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl method), 167
_mul_ (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 143
_mul_ (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 157

_mul_ (sage.rings.polynomial.polynomial_element.Polynomial method), 32
_mul_ (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 129
_mul_ (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 129
_mul_ (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl method), 167
_mul_ (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 143
_mul_ (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 158

_add_trunc_ (sage.rings.polynomial.polynomial_element.Polynomial method), 33
_add_trunc_ (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 129
_add_trunc_ (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl method), 167
_add_trunc_ (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 143
_add_trunc_ (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 158

_rmul_ (sage.rings.polynomial.polynomial_element.Polynomial method), 32
_rmul_ (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 129
_rmul_ (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl method), 167
_rmul_ (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 143
_rmul_ (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 158

_sub_ (sage.rings.polynomial.polynomial_element.Polynomial method), 32
_sub_ (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 129
_sub_ (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl method), 167
_sub_ (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 143
_sub_ (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 158

A

invariant () (sage.rings.invariants.invariant_theory.BinaryQuintic method), 443

abc_pd (class in sage.rings.polynomial.polynomial_compiled), 247

add_m_mul_q () (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 396
binary_quintic() (in module sage.rings.invariants.invariant_theory_binary_quintic), 457
binary_quintic_coefficients_from_invariants() (in module sage.rings.invariants.invariant_theory_binary_quintic), 481
BinaryQuartic (class in sage.rings.invariants.invariant_theory_binary_quartic), 440
BinaryQuintic (class in sage.rings.invariants.invariant_theory_binary_quintic), 443
bitsize_doctest() (in module sage.rings.polynomial.real_roots), 193
blocks() (in module sage.rings.polynomial.term_order.TermOrder), 259
BooleanMonomial (class in sage.rings.polynomial.pbori.pbori BooleanMonomial_pbitori), 593
BooleanMonomialIterator (class in sage.rings.polynomial.pbori.pbori BooleanMonomial_iterator_pbitori), 597
BooleanMonomialMonoid (class in sage.rings.polynomial.pbori.pbori BooleanMonomial_monoid_pbitori), 597
BooleanMonomialVariableIterator (class in sage.rings.polynomial.pbori.pbori BooleanMonomial_variable_iterator_pbitori), 598
BooleanMulAction (class in sage.rings.polynomial.pbori.pbori BooleanMultiplicationAction_pbitori), 598
BooleanPolynomial (class in sage.rings.polynomial.pbori.pbori BooleanPolynomial_pbitori), 598
BooleanPolynomialEntry (class in sage.rings.polynomial.pbori.pbori BooleanPolynomial_entry_pbitori), 614
BooleanPolynomialIdeal (class in sage.rings.polynomial.pbori.pbori BooleanPolynomial_ideal_pbitori), 614
BooleanPolynomialIterator (class in sage.rings.polynomial.pbori.pbori BooleanPolynomial_iterator_pbitori), 617
BooleanPolynomialRing (class in sage.rings.polynomial.pbori.pbori BooleanPolynomial_ring_pbitori), 617
BooleanPolynomialRing_constructor() (in module sage.rings.polynomial.pbori.pbori BooleanPolynomial_ring_constructor), 61
BooleanPolynomialVector (class in sage.rings.polynomial.pbori.pbori BooleanPolynomial_vector_pbitori), 625
BooleanPolynomialVectorIterator (class in sage.rings.polynomial.pbori.pbori BooleanPolynomial_vector_iterator_pbitori), 626
BooleConstant (class in sage.rings.polynomial.pbori.pbori BooleConstant_pbitori), 626
BooleSet (class in sage.rings.polynomial.pbori.pbori BooleSet_pbitori), 585
BooleSetIterator (class in sage.rings.polynomial.pbori.pbori BooleSet_iterator_pbitori), 586
bp_done() (in module sage.rings.polynomial.real_roots.island), 205
buchberger() (in module sage.rings.polynomial.toy_buchberger), 486
buchberger_improved() (in module sage.rings.polynomial.toy_buchberger), 486
C

can_convert_to_singular() (in module sage.rings.polynomial.polynomial_singular_interface), 175
can_rewrite() (in module sage.rings.polynomial.pbori.pbori.ReductionStrategy), 633
canonical_form() (in module sage.rings.invariants.invariant_theory.BinaryQuintic), 447
cardinality() (in module sage.rings.polynomial.polynomial_quotient_ring), 233
cartesian_product() (in module sage.rings.polynomial.pbori.pbori.BooleSet), 587
CCuddNavigator (class in sage.rings.polynomial.pbori.pbori), 626
change() (in module sage.rings.polynomial.pbori.pbori.BooleSet), 587
change_ring() (in module sage.rings.polynomial.laurent_polynomial.LaurentPolynomial), 525
change_ring() (in module sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing), 520
change_ring() (in module sage.rings.polynomial.multi_polynomial.MPolynomial), 283
change_ring() (in module sage.rings.polynomial.multi_polynomial_element.MPolynomialElement), 309
change_ring() (in module sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal), 328
change_ring() (in module sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base), 274
change_ring() (in module sage.rings.polynomial.pbori.pbori.BooleanPolynomialRing), 618
change_ring() (in module sage.rings.polynomial.pbori.pbori.BooleanPolynomialRing_iterator), 617
change_ring() (in module sage.rings.polynomial.real_mpfr_dense.PolynomialRealMPFR), 172
change_ring() (in module sage.rings.polynomial.pbori.pbori.BooleanPolynomialRing_ideal), 19
change_var() (in module sage.rings.polynomial.pbori.pbori.BooleanPolynomialRing_variable), 20
change_variable_name() (in module sage.rings.polynomial.pbori.pbori.BooleanPolynomialRing_variable), 36
characteristic() (in module sage.rings.fraction_field.FractionField_generic), 500
characteristic() (in module sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing), 555
characteristic() (in module sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing), 520
characteristic() (in module sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base), 274
characteristic() (in module sage.rings.polynomial.polynomial_quotient_ring.QuotientRing_generic), 234
characteristic() (sage.rings.polynomial.polynomial_ring.PolynomialRing_general
method), 20
charpoly() (sage.rings.polynomial.polynomial_quotient_ring_element.QuotientRingElement
method), 244
cheap_reductions() (sage.rings.polynomial.pbori.pbori.ReductionStrategy
method), 633
cl_maximum_root() (in module
sage.rings.polynomial.real_roots), 194
cl_maximum_root_first_lambda() (in module
sage.rings.polynomial.real_roots), 194
cl_maximum_root_local_max() (in module
sage.rings.polynomial.real_roots), 194
class_group() (sage.rings.polynomial.polynomial_quotient_ring_element.QuotientRing_element
classmethod), 234
class_number() (sage.rings.fraction_field.FractionField_1poly_field
method), 499
clean_top_by_chain_criterion() (in module
sage.rings.polynomial.pbori.pbori.GroebnerStrategy
method), 628
clebsch_invariants() (sage.rings.invariants.invariant_theory.BinaryQuartic
method), 447
close() (sage.rings.polynomial.pbori.pbori.BooleanPolynomialRing
method), 618
coeff_pd (in sage.rings.polynomial.polynomial_compiled), 248
coefficient() (sage.rings.polynomial.polynomial_element_generic.PolynomialElement_generic
 method), 561
coefficient() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial
 method), 247
coefficient_matrix() (in module
sage.rings.polynomial.toy_variety), 489
coefficient_matrix() (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic
classmethod), 380
coefficients() (in module
sage.rings.polynomial.multi_polynomial_element_generic.PolynomialElement_generic
coeffs_bitsize() (sage.rings.polynomial.real_roots.bernstein_polynomial
method), 193
combine_to_positives() (sage.rings.polynomial.polydict.ETuple
method), 420
common_nonzero_positions() (sage.rings.polynomial.polydict.ETuple
method), 420
composition() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial
method), 37
composition() (sage.rings.polynomial.polynomial_libsingular.MPolynomial_libsingular
method), 274
composition() (sage.rings.polynomial.polynomial_ring.PolynomialRing_generic
method), 20
complex_embeddings() (sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRing_element
method), 381
complex_number() (sage.rings.polynomial.multi_polynomial_element_generic.Polynomial_element
method), 381
complex_roots() (sage.rings.polynomial.real_roots.bernstein_polynomial
method), 248
consistency() (sage.rings.polynomial.real_roots.bernstein_polynomial
method), 248
completion() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial
method), 38
complete() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial
method), 38
compose_power() (sage.rings.polynomial.polynomial_element_generic.Polynomial_element
method), 527
compose_trunc() (sage.rings.polynomial.polynomial_element_generic.Polynomial_element
method), 536
composed_op() (sage.rings.polynomial.polynomial_element_generic.Polynomial_element
method), 283
connected_components() (sage.rings.polynomial.multi_polynomial_element_generic.Polynomial_element
method), 381
connectivity_graph() (sage.rings.polynomial.multi_polynomial_element_generic.Polynomial_element
method), 381
connection_graph() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial
method), 37
<table>
<thead>
<tr>
<th>Function</th>
<th>Module (Class)</th>
</tr>
</thead>
<tbody>
<tr>
<td>diff()</td>
<td>(sage.rings.polynomial.pb.or.pb.BooleSet method), 588</td>
</tr>
<tr>
<td>down_degree()</td>
<td>(sage.rings.polynomial.real_roots.interval_bernstein_polynomial method), 202</td>
</tr>
<tr>
<td>down_degree_iter()</td>
<td>(sage.rings.polynomial.real_roots.interval_bernstein_polynomial method), 202</td>
</tr>
<tr>
<td>dimension()</td>
<td>(sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal method), 144</td>
</tr>
<tr>
<td>dual()</td>
<td>(sage.rings.invariants.invariant_theory.QuadraticForm method), 462</td>
</tr>
<tr>
<td>discriminant()</td>
<td>(sage.rings.invariants.invariant_theory.BinaryQuartic method), 421</td>
</tr>
<tr>
<td>easy_linear_factors()</td>
<td>(sage.rings.polynomial.polydict.ETuple method), 421</td>
</tr>
<tr>
<td>Eisenstein()</td>
<td>(sage.rings.invariants.invariant_theory.BinaryQuartic method), 131</td>
</tr>
<tr>
<td>EisensteinE()</td>
<td>(sage.rings.invariants.invariant_theory.BinaryQuartic method), 138</td>
</tr>
<tr>
<td>elimination_ideal()</td>
<td>(sage.rings.polynomial.real_roots), 196</td>
</tr>
<tr>
<td>elimination_ideal()</td>
<td>(sage.rings.polynomial.real_roots), 196</td>
</tr>
<tr>
<td>divide()</td>
<td>(sage.rings.polynomial.pb.or.pb.BooleSet method), 588</td>
</tr>
<tr>
<td>divide()</td>
<td>(sage.rings.polynomial.pb.or.pb.BooleSet method), 588</td>
</tr>
<tr>
<td>divide_difference()</td>
<td>(sage.rings.polynomial.pb.or.pb.BooleSet method), 588</td>
</tr>
<tr>
<td>divides()</td>
<td>(sage.rings.polynomial.multi_polynomial_libsingular.MPolynomialQuotientRing_generic method), 400</td>
</tr>
<tr>
<td>divides()</td>
<td>(sage.rings.polynomial.pb.or.pb.BooleSet method), 594</td>
</tr>
<tr>
<td>divisors()</td>
<td>(sage.rings.polynomial.pb.or.pb.BooleSet method), 594</td>
</tr>
<tr>
<td>divisors_of()</td>
<td>(sage.rings.polynomial.pb.or.pb.BooleSet method), 588</td>
</tr>
<tr>
<td>done()</td>
<td>(sage.rings.polynomial.real_roots.island method), 205</td>
</tr>
<tr>
<td>dotprod()</td>
<td>(sage.rings.polynomial.polydict.ETuple method), 422</td>
</tr>
</tbody>
</table>
groebner_basis() (sage.rings.polynomial.symmetric_invariants.invariant_theory.TernaryCubic), 559
groebner_fan() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr), 559
GroebnerStrategy (class in sage.rings.polynomial.pbori.pbori), 627
homogeneous_symmetric_function() (in module sage.rings.polynomial.omega), 547
homogenize() (sage.rings.polynomial.multi_polynomial.MPolynomial), 441
homogenize() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal), 441
hess() (sage.rings.polynomial.multi_polynomial_element.MPolynomial), 52
has_root() (sage.rings.polynomial.real_roots.real_roots.RealRoots), 209
head_normal_form() (sage.rings.polynomial.pbori.pbori.BooleanPolynomialRing), 633
hensel_lift() (sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_cdv), 113
hensel_lift() (sage.rings.polynomial.polynomial_element.Polynomial), 149
Hessian() (sage.rings.polynomial.invariants.invariant_theory.TernaryCubic), 467
hilbert_numererator() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal), 352
hilbert_poincare_series() (in module InfiniteGenDict), 559
hilbert_polynomial() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal), 352
inhomogeneous_quadratic_form() (class in sage.rings.polynomial.infinite_polynomial_element), 560

InfinitePolynomialRingFactory

InfinitePolynomialGen (class in sage.rings.polynomial.infinite_polynomial_ring), 553

InfinitePolynomialRing_dense (class in sage.rings.polynomial.infinite_polynomial_ring), 554

InfinitePolynomialRing_sparse (class in sage.rings.polynomial.infinite_polynomial_ring), 555

InfinitePolynomialRingFactory (class in sage.rings.polynomial.infinite_polynomial_ring), 553

interval_bernstein_polynomial (class in sage.rings.polynomial.real_roots), 197

interval_bernstein_polynomial_float (class in sage.rings.polynomial.real_roots), 197

interval_bernstein_polynomial_integer (class in sage.rings.polynomial.real_roots), 201

intervals_disjoint() (in module sage.rings.polynomial.invariants.invariant_theory.InvariantTheoryFactory), 221

int_list() (class in sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_modn_ntl_zz), 162

interval_roots() (in module sage.rings.polynomial.invariants.invariant_theory.InvariantTheoryFactory), 221

int_list() (class in sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_modn_zz), 168

intvec_to_doublevec() (in module sage.rings.polynomial.real_roots), 204

intersection() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal, 356

interreduction() (sage.rings.polynomial.symmetric_ideal.SymmetricIdeal, 572

interreduced_basis() (sage.rings.polynomial.infinite_polynomial_element), 560

interreduced_basis() (sage.rings.polynomial.symmetric_ideal.SymmetricIdeal, 572

interpolation_polynomial() (class in sage.rings.polynomial.invariants.invariant_theory.InvariantTheoryFactory, 454

interpolate_smallest_lex() (class in sage.rings.polynomial.infinite_polynomial_element), 637

interpolate() (class in sage.rings.polynomial.pbori.pbori.BooleSet), 589

inverse() (class in sage.rings.polynomial.invariants.invariant_theory.InvariantTheoryFactory, 454

inverse() (class in sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint), 433

inverse() (class in sage.rings.polynomial.symmetric_ideal.SymmetricIdeal_singular_repr), 315

inverse() (class in sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint), 433

inverse() (class in sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint), 572

inverse() (class in sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl), 572

inverse() (class in sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint), 572

inverse() (class in sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl), 572

inverse_of_unit() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal, 356

inverse_of_unit() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal, 356

irreducible_element() (sage.rings.polynomial.invariants.invariant_theory.InvariantTheoryFactory, 454
Index
is_polynomial_sequence() (in module sage.rings.polynomial.multi_polynomial_sequence), 28
is_prime() (sage.rings.polynomial.multi_polynomial_ideal_monoid), 366
is_primitive() (sage.rings.polynomial.polynomial_element), 299
is_singleton() (sage.rings.polynomial.pbori.pbori.BooleanPolynomial), 112
is_real_rooted() (sage.rings.polynomial.multi_polynomial_element), 272
is_square() (sage.rings.polynomial.polynomial_element), 62
is_square() (sage.rings.polynomial.polynomial_element), 15
is_square() (sage.rings.polynomial.multi_polynomial_libsingular), 168
is_square() (sage.rings.polynomial.polynomial_ring), 33
is_square() (sage.rings.polynomial.laurent_polynomial), 217
is_squarefree() (sage.rings.polynomial.multi_polynomial_element), 20
is_squarefree() (sage.rings.polynomial.polynomial_element), 65
is_surjective() (sage.rings.polynomial.multi_polynomial_element), 115
is_univariate() (sage.rings.polynomial.polynomial_element), 67
is_univariate() (sage.rings.polynomial.multi_polynomial_libsingular), 15
is_univariate() (sage.rings.polynomial.multi_polynomial_element_generic), 67
is_univariate() (sage.rings.polynomial.pbori.pbori.BooleanPolynomial), 112
is_univariate() (sage.rings.polynomial.pbori.pbori.BooleanPolynomial), 292
is_weighted_degree_order() (sage.rings.polynomial.polynomial_ring), 246
is_weighted_degree_order() (sage.rings.polynomial.multi_polynomial_element), 264
is_squarefree() (sage.rings.polynomial.multi_polynomial_element), 406
is_squarefree() (sage.rings.polynomial.polynomial_element), 406
is_zero() (sage.rings.polynomial.polynomial_ring), 188
is_zero() (sage.rings.polynomial.multi_polynomial_ideal), 124
is_zero() (sage.rings.polynomial.multi_polynomial_element), 586
is_zero() (sage.rings.polynomial.multi_polynomial_element), 133
is_zero() (sage.rings.polynomial.multi_polynomial_libsingular), 188
is_zero() (sage.rings.polynomial.polynomial_ring), 156
is_zero() (sage.rings.polynomial.polynomial_template), 188
island (class in sage.rings.polynomial.real_roots), 204

iter() (sage.rings.fraction_field_FpT.FpT method), 507

iterator_exp_coeff() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial method), 532

iterator_exp_coeff() (sage.rings.polynomial.multi_polynomial.MPolynomial method), 293

iterator_exp_coeff() (sage.rings.polynomial.multi_polynomial_element.MPolynomial method), 317

iterator_exp_coeff() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial method), 407

iterindex() (sage.rings.polynomial.polybori.polybori.BooleanMonomial method), 519

LaurentPolynomial (class in sage.rings.polynomial.laurent_polynomial), 17

LaurentPolynomial (class in sage.rings.polynomial.laurent_polynomial_mpair), 525

LaurentPolynomial_univariate (class in sage.rings.polynomial.laurent_polynomial_univariate), 526

LaurentPolynomialRing() (in module sage.rings.polynomial.laurent_polynomial_ring), 517

LaurentPolynomialRing_generic (class in sage.rings.polynomial.laurent_polynomial_ring), 519

J

j_covariant() (sage.rings.invariants.invariant_theory.BinaryQuintic method), 451

J_c covariant() (sage.rings.invariants.invariant_theory.TernaryCubic method), 467

J_c covariant() (sage.rings.invariants.invariant_theory.TwoTernaryQuadratics method), 473

J_c covariant() (sage.rings.invariants.invariant_theory.TwoQuaternaryQuadratics method), 476

jacobian_ideal() (sage.rings.polynomial.multi_polynomial_ideal_libsingular.MPolynomialRing_libsingular_ideal method), 293

karatsuba_threshold() (sage.rings.polynomial.laurent_polynomial_ring.PolynomialRing_general method), 23

kbase_libsingular() (in module sage.rings.polynomial.multi_polynomial_libsingular), 419

krull_dimension() (sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing_sparse method), 557

krull_dimension() (sage.rings.polynomial.laurent_polynomial_ring.PolynomialRing_ltc method), 522

krull_dimension() (sage.rings.polynomial.multi_polynomial_ring_generic.PolynomialRing_generic method), 276

krull_dimension() (sage.rings.polynomial.multi_polynomial_ring_sparse.PolynomialRing_sparse method), 238

krull_dimension() (sage.rings.polynomial.multi_polynomial_ring.PolynomialRing_general method), 23

l

lagrange_polynomial() (sage.rings.polynomial.lagrange_polynomial.Polynomial method), 17

latex() (sage.rings.polynomial.polydict.PolyDict method), 426

lead() (sage.rings.polynomial.polybori.polybori.BooleanMonomial method), 605

lead() (sage.rings.polynomial.polybori.polybori.BooleanPolynomial method), 632

lead_deg() (sage.rings.polynomial.polybori.polybori.BooleanPolynomial method), 605

lead_divisors() (sage.rings.polynomial.polybori.polybori.BooleanPolynomial method), 606

lead_divisors() (sage.rings.polynomial.polybori.polybori.BooleanPolynomial method), 606
leading_coefficient()
    (sage.rings.polynomial.polynomial_element.Polynomial_element
    method), 189
lex_lead()
    (sage.rings.polynomial.polynomial_element.Polynomial_element
    method), 68
less_bits()
    (sage.rings.polynomial.real_roots.island
    method), 205
lex_lead()
    (sage.rings.polynomial.pboring.BoothenPolynomial
    method), 606
lex_lead_deg()
    (sage.rings.polynomial.pboring.BoothenPolynomial
    method), 606
lift()
    (sage.rings.polynomial.multi_polynomial.MPolynomial
    method), 293
lift()
    (sage.rings.polynomial.multi_polynomial_element.MPolynomial
    method), 317
lift()
    (sage.rings.polynomial_multi_polynomial_libsingular.MPolynomial_libsingular
    method), 408
lift()
    (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense
    method), 180
lift()
    (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense
    method), 238
lift()
    (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense
    method), 246
linear_map
    (class in sage.rings.polynomial.real_roots), 206
linear_representation()
    (in module sage.rings.polynomial.real_roots), 69
list()
    (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense
    method), 427
list()
    (sage.rings.polynomial.multi_polynomial_element.Polynomial
    method), 486
list()
    (in module sage.rings.polynomial.toy_buchberger), 491
list()
    (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense
    method), 181
list()
    (sage.rings.polynomial.multi_polynomial_element.Polynomial
    method), 191
list()
    (sage.rings.polynomial.real_roots.bernstein_polynomial_factory
    method), 563
list()
    (sage.rings.polynomial.real_roots.bernstein_polynomial_factory
    method), 563
list()
    (sage.rings.polynomial.multi_polynomial_element.Polynomial
    method), 607
list()
    (sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial Sparse
    method), 69
list()
    (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense
    method), 181
list()
    (sage.rings.polynomial.real_roots.bernstein_polynomial_factory
    method), 191
list()
    (sage.rings.polynomial.real_roots.bernstein_polynomial_factory
    method), 191
list()
    (sage.rings.polynomial.toy_d_basis
    method), 276
list()
    (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base
    method), 118
list()
    (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense
    method), 133
list()
    (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense
    method), 133
list()
    (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense
    method), 133
list()
    (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense
    method), 133
magma_str() (sage.rings.polynomial.term_order.TermOrder method), 265
main() (sage.rings.polynomial.pbori.pbori.FGLMStrategy method), 626
make_element() (in module sage.rings.fraction_field_element), 507
make_element() (in module sage.rings.polynomial.polynomial_gf2x), 126
make_element() (in module sage.rings.polynomial.polynomial_modn_dense_flint), 169
make_element() (in module sage.rings.polynomial.polynomial_zmod_flint), 161
make_element() (in module sage.rings.polynomial.polynomial_zz_pex), 190
make_element_old() (in module sage.rings.fraction_field_element), 507
make_ETuple() (in module sage.rings.polynomial.polydict), 430
make_generic_polynomial() (in module sage.rings.polynomial.polynomial_element), 111
make_padic_poly() (in module sage.rings.polynomial.padic.polynomial_padic_capped_relative_dense), 185
make_PolyDict() (in module sage.rings.polynomial.polydict), 430
make_PolynomialReal Dense() (in module sage.rings.polynomial.polynomial_real_mpfr_dense), 175
map_coefficients() (in module sage.rings.polynomial.laurent_polynomial), 525
map_coefficients() (in module sage.rings.polynomial.multi_polynomial_quotient_ring_element.PolynomialQuotientRingElement), 70
map_coefficients() (in module sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic), 246
map every_x_to_x_plus_one() (in module sage.rings.polynomial.pbori.pbori), 639
map every_x_to_x_plus_one() (in module sage.rings.polynomial.pbori.BooleanPolynomial), 607
matrix() (in module sage.rings.invariants.invariant_theory.QuadraticForm), 464
matrix() (in module sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRingElement), 246
matrix() (in module sage.rings.polynomial.term_order.TermOrder), 265
max abs_doublevec() (in module sage.rings.polynomial.real_roots), 206
max_bitsize_intvec_doctest() (in module sage.rings.polynomial.real_roots), 206
maximal_degree() (in module sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic), 383
maximal_order() (in module sage.rings.fraction_field.FractionField_1poly_field), 499
maximum_root_first lambda() (in module sage.rings.polynomial.real_roots), 206
maximum_root_local_max() (in module sage.rings.polynomial.real_roots), 207
min exp() (in module sage.rings.polynomial.polydict.PolyDict), 427
min_max_delta_intvec() (in module sage.rings.polynomial.real_roots), 207
min_max_diff_doublevec() (in module sage.rings.polynomial.real_roots), 207
min_max_diff_intvec() (in module sage.rings.polynomial.real_roots), 207
minimal associated primes() (in module sage.rings.polynomial.multi_polynomial_element.Polynomial), 507
minimal elements() (in module sage.rings.polynomial.pbori.pbori.GroebnerStrategy), 589
minimize() (in module sage.rings.polynomial.pbori.pbori.GroebnerStrategy), 589
minimalize and tail reduce() (in module sage.rings.polynomial.pbori.pbori.GroebnerStrategy), 630
minpoly() (in module sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRingElement), 246
mk_context() (in module sage.rings.fraction_field.FractionField_1poly_field), 497
mk_ibpf() (in module sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic), 207
mk ibpi() (in module sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic), 207
mk Ibpi() (in module sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic), 207
mod() (in module sage.rings.polynomial.polynomial_element.Polynomial), 71
mod_mon_set() (in module sage.rings.polynomial.pbori.pbori), 639
mod var_set() (in module sage.rings.polynomial.pbori.pbori), 639
matrix() (in module sage.rings.invariants.invariant_theory.QuadraticForm), 464
matrix() (in module sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRingElement), 246
module

monomial_lcm()
(sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair)
method, 532

monomial_divides()
(sage.rings.polynomial.multi_polynomial_element.MPolynomial_libsingular)
method, 318

monomial_coefficient()
(sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair)
method, 468

monomial_reduction()
(sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair)
method, 471

monomial_lcm()
(sage.rings.polynomial.multi_polynomial_element.MPolynomial_libsingular)
method, 319

monomial_divides()
(sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair)
method, 410

monomial_divides()
(sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular)
method, 410

monomial_lcm()
(sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular)
method, 306

monomial_divides()
(sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair)
method, 608

monomial_divides()
(sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular)
method, 72

monomial_lcm()
(sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair)
method, 427

monomial_divides()
(sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular)
method, 394

monomial_lcm()
(sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair)
method, 306

monomial_pairwise_prime()
(sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair)
method, 307

monomial_pairwise_prime()
(sage.rings.polynomial.multi_polynomial_element.MPolynomial_libsingular)
method, 307

monomial_quotient()
(sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair)
method, 283

monomial_quotient()
(sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular)
method, 307

monomial_pairwise_prime()
(sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair)
method, 310

monomial_pairwise_prime()
(sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair)
method, 307

monomial_quotient()
(sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair)
method, 307

monomial_quotient()
(sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair)
method, 307

monomial_reduction()
(sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair)
method, 307

MonomialConstruct (class in sage.rings.polynomial.multi_polynomial)
345

MonomialFactory (class in sage.rings.polynomial.multi_polynomial)
345

monomials() (in module sage.rings.polynomials), 435

monomials() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair)
method, 273

monomials() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair)
method, 442

monomials() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair)
method, 451

monomials() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_mpair)
method, 464
polynomial() (sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial_sparse method), 564

polynomial() (sage.rings.polynomial.multi_polynomial.MPolynomial method), 298

polynomial() (sage.rings.polynomial.polynomial_element.Polynomial method), 79

Polynomial_absolute_number_field_dense (class in sage.rings.polynomial.polynomial_number_field), 127

polynomial_coefficient() (sage.rings.polynomial.polydict.PolyDict method), 428

polynomial_construction() (in module sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate), 542

polynomial_default_category() (in module sage.rings.polynomial.polynomial_ring_constructor), 7

Polynomial_dense_mod_n (class in sage.rings.polynomial.polynomial_modn_dense_ntl), 162

Polynomial_dense_mod_p (class in sage.rings.polynomial.polynomial_modn_dense_ntl), 164

Polynomial_dense_mod_nntl_ZZ (class in sage.rings.polynomial.polynomial_modn_dense_ntl), 165

Polynomial_dense_mod_ntl_zz (class in sage.rings.polynomial.polynomial_modn_dense_ntl), 167

Polynomial_generic_cdv (class in sage.rings.polynomial.polynomial_element_generic), 112

Polynomial_generic_cdvf (class in sage.rings.polynomial.polynomial_element_generic), 114

Polynomial_generic_cdvr (class in sage.rings.polynomial.polynomial_element_generic), 114

Polynomial_generic_dense (class in sage.rings.polynomial.polynomial_element_generic), 107

Polynomial_generic_dense_cdv (class in sage.rings.polynomial.polynomial_element.generic), 115

Polynomial_generic_dense_cdvr (class in sage.rings.polynomial.polynomial_element_generic), 115

Polynomial_generic_dense_field (class in sage.rings.polynomial.polynomial_element_generic), 115

Polynomial_generic_dense_inexact (class in sage.rings.polynomial.polynomial_element), 564

Polynomial_generic_domain (class in sage.rings.polynomial.polynomial_element_generic), 115

Polynomial_generic_field (class in sage.rings.polynomial.polynomial_element_generic), 127

Polynomial_generic_sparse (class in sage.rings.polynomial.polynomial_element_generic), 116

Polynomial_generic_sparse_cdv (class in sage.rings.polynomial.polynomial_element_generic), 120

Polynomial_generic_sparse_cdvf (class in sage.rings.polynomial.polynomial_element_generic), 120

Polynomial_generic_sparse_cdvr (class in sage.rings.polynomial.polynomial_element_generic), 120

Polynomial_generic_sparse_field (class in sage.rings.polynomial.polynomial_element_generic), 121

Polynomial_GF2X (class in sage.rings.polynomial.polynomial_gf2x), 122

Polynomial_integer_dense_flint (class in sage.rings.polynomial.polynomial_integer_dense_flint), 128

Polynomial_integer_dense_ntl (class in sage.rings.polynomial.polynomial_integer_dense_ntl), 137

Polynomial_padic (class in sage.rings.polynomial.padics.polynomial_padic), 176

Polynomial_padic_capped_relative_dense (class in sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense), 179

Polynomial_padic_flat (class in sage.rings.polynomial.padics.polynomial_padic_flat), 186

polynomial_ring() (sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing_dense method), 554

polynomial_ring() (sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_generic method), 239

polynomial_ring() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic method), 239

Index 669
<table>
<thead>
<tr>
<th>Class Name</th>
<th>Module Location</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polynomial_singular_repr</td>
<td>sage.rings.polynomial.polynomial_singular_interface</td>
<td>175</td>
</tr>
<tr>
<td>Polynomial_template</td>
<td>sage.rings.polynomial.polynomial_gf2x</td>
<td>123</td>
</tr>
<tr>
<td>Polynomial_template</td>
<td>sage.rings.polynomial.polynomial_zmod_flint</td>
<td>155</td>
</tr>
<tr>
<td>Polynomial_template</td>
<td>sage.rings.polynomial.polynomial_zz_pex</td>
<td>187</td>
</tr>
<tr>
<td>Polynomial_zmod_flint</td>
<td>sage.rings.polynomial.polynomial_zmod_flint</td>
<td>157</td>
</tr>
<tr>
<td>Polynomial_ZZ_pEX</td>
<td>sage.rings.polynomial.polynomial_zz_pex</td>
<td>186</td>
</tr>
<tr>
<td>Polynomial_ZZ_pX</td>
<td>sage.rings.polynomial.polynomial_zz_pex</td>
<td>187</td>
</tr>
<tr>
<td>PolynomialBaseringInjection</td>
<td>sage.rings.polynomial.polynomial_element</td>
<td>105</td>
</tr>
<tr>
<td>PolynomialConstruct</td>
<td>sage.rings.polynomial.pbori, pbori</td>
<td>632</td>
</tr>
<tr>
<td>PolynomialFactory</td>
<td>sage.rings.polynomial.pbori, pbori</td>
<td>632</td>
</tr>
<tr>
<td>PolynomialQuotientRing_coercion</td>
<td>sage.rings.polynomial.polynomial_quotient_ring</td>
<td>226</td>
</tr>
<tr>
<td>PolynomialQuotientRing_domain</td>
<td>sage.rings.polynomial.polynomial_quotient_ring</td>
<td>227</td>
</tr>
<tr>
<td>PolynomialQuotientRing_field</td>
<td>sage.rings.polynomial.polynomial_quotient_ring</td>
<td>228</td>
</tr>
<tr>
<td>PolynomialQuotientRing_generic</td>
<td>sage.rings.polynomial.polynomial_quotient_ring</td>
<td>229</td>
</tr>
<tr>
<td>PolynomialQuotientRingElement</td>
<td>sage.rings.polynomial.polynomial_quotient_ring</td>
<td>244</td>
</tr>
<tr>
<td>PolynomialQuotientRingFactory</td>
<td>sage.rings.polynomial.polynomial_quotient_ring</td>
<td>223</td>
</tr>
<tr>
<td>PolynomialRealDense</td>
<td>sage.rings.polynomial.polynomial_real_mpfr_dense</td>
<td>172</td>
</tr>
<tr>
<td>polynomials()</td>
<td>sage.rings.polynomial.polynomial_ring</td>
<td>2</td>
</tr>
<tr>
<td>PolynomialRing()</td>
<td>sage.rings.polynomial.polynomial_ring_constructor, 2</td>
<td></td>
</tr>
<tr>
<td>PolynomialRing_cdvf</td>
<td>sage.rings.polynomial.polynomial_ring</td>
<td>11</td>
</tr>
<tr>
<td>PolynomialRing_cdvr</td>
<td>sage.rings.polynomial.polynomial_ring</td>
<td>11</td>
</tr>
<tr>
<td>PolynomialRing_commutative</td>
<td>sage.rings.polynomial.polynomial_ring</td>
<td>11</td>
</tr>
<tr>
<td>PolynomialRing_denseFiniteField</td>
<td>sage.rings.polynomial.polynomial_ring</td>
<td>12</td>
</tr>
<tr>
<td>PolynomialRing_denseMod_n</td>
<td>sage.rings.polynomial.polynomial_ring</td>
<td>13</td>
</tr>
<tr>
<td>PolynomialRing_denseMod_p</td>
<td>sage.rings.polynomial.polynomial_ring</td>
<td>13</td>
</tr>
<tr>
<td>PolynomialRing_densePadicFieldCappedRelative</td>
<td>sage.rings.polynomial.polynomial_ring</td>
<td>15</td>
</tr>
<tr>
<td>PolynomialRing_densePadicFieldGeneric</td>
<td>sage.rings.polynomial.polynomial_ring</td>
<td>15</td>
</tr>
<tr>
<td>PolynomialRing_densePadicRingCappedAbsolute</td>
<td>sage.rings.polynomial.polynomial_ring</td>
<td>15</td>
</tr>
<tr>
<td>PolynomialRing_densePadicRingCappedRelative</td>
<td>sage.rings.polynomial.polynomial_ring</td>
<td>15</td>
</tr>
<tr>
<td>PolynomialRing_densePadicRingFixedMod</td>
<td>sage.rings.polynomial.polynomial_ring</td>
<td>15</td>
</tr>
<tr>
<td>PolynomialRing_densePadicRingGeneric</td>
<td>sage.rings.polynomial.polynomial_ring</td>
<td>16</td>
</tr>
<tr>
<td>PolynomialRing_field</td>
<td>sage.rings.polynomial.polynomial_ring</td>
<td>16</td>
</tr>
<tr>
<td>PolynomialRing_general</td>
<td>sage.rings.polynomial.polynomial_ring</td>
<td>19</td>
</tr>
<tr>
<td>PolynomialRing_integaldomain</td>
<td>sage.rings.polynomial.polynomial_ring</td>
<td></td>
</tr>
<tr>
<td>PolynomialRing_singular_repr</td>
<td>sage.rings.polynomial.polynomial_singular_interface</td>
<td>175</td>
</tr>
<tr>
<td>PolynomialRing_homomorphismFromBase</td>
<td>sage.rings.polynomial.polynomial_ring_homo morphism</td>
<td></td>
</tr>
</tbody>
</table>

379 PolynomialSequence_gf2 (class in sage.rings.polynomial.multi_polynomial_sequence), 386

Polyring_FpT_coerce (class in sage.rings.fraction_field_FpT), 513

power_trunc() (sage.rings.polynomial.multi_polynomial_element.Polynomial method), 79

prec() (sage.rings.polynomial.multi_polynomial_element.Polynomial method), 80

prec_degree() (sage.rings.polynomial.pads.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense method), 182

precision_absolute() (sage.rings.polynomial.multi_polynomial_element.Polynomial method), 182

precision_relative() (sage.rings.polynomial.multi_polynomial_element.Polynomial method), 183

PrecisionError, 190

precompute_degree_reduction_cache() (in module sage.rings.polynomial.real_roots), 210

primary_decomposition() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr method), 359

primary_decomposition_complete() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr method), 360

pseudo_divrem() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 134

pseudo_quo_rem() (sage.rings.polynomial.multi_polynomial_element_generic method), 80

pseudoinverse() (in module sage.rings.polynomial.real_roots), 211

Q

quadratic_form() (sage.rings.invariants.invariant_theory.BinaryQuinticFactory method), 458

QuadraticForm (class in sage.rings.invariants.invariant_theory), 462

quaternary_biquadratic() (sage.rings.invariants.invariant_theory.InvariantTheoryFactory method), 459

quaternary_quadratic() (sage.rings.invariants.invariant_theory.InvariantTheoryFactory method), 459

quo_rem() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_1var_dense_ntl.Polynomial_1var_dense_ntl method), 533

R

R_invariant() (sage.rings.invariants.invariant_theory.BinaryQuinticFactory method), 458

radical() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal method), 362

radical() (sage.rings.polynomial.multi_polynomial_element.Polynomial method), 81

random_element() (sage.rings.fraction_field.FractionField_generic method), 501

random_element() (sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRingFactory method), 522

random_element() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal method), 11
S_units() (sage.rings.polynomial.polynomial_quotient_ring \( \text{method} \)), 232
sage.rings.fraction_field module, 497
sage.rings.fraction_field_element module, 503
sage.rings.fraction_field_FpT module, 507
sage.rings.invariants.invariant_theory module, 435
sage.rings.invariants.reconstruction module, 480
sage.rings.monomials module, 435
sage.rings.polynomial.complex_roots module, 219
sage.rings.polynomial.convolution module, 248
sage.rings.polynomial.cyclotomic module, 249
sage.rings.polynomial.flatten module, 432
sage.rings.polynomial.hilbert module, 431
sage.rings.polynomial.ideal module, 222
sage.rings.polynomial.infinite_polynomial_element module, 559
sage.rings.polynomial.infinite_polynomial_rings module, 549
sage.rings.polynomial.laurent_polynomial module, 525
sage.rings.polynomial.laurent_polynomial_ring module, 517
sage.rings.polynomial.multi_polynomial module, 283
sage.rings.polynomial.multi_polynomial_ideal module, 309
sage.rings.polynomial.module, 324
sage.rings.polynomial.multi_polynomial_ideal.libsingular module, 418
sage.rings.polynomial.multi_polynomial_libsingular module, 390
sage.rings.polynomial.multi_polynomial_element module, 305
sage.rings.polynomial.multi_polynomial_ring module, 273
sage.rings.polynomial.multi_polynomial_sequence module, 375
sage.rings.polynomial.omega module, 544
sage.rings.polynomial.padics.polynomial_padic module, 176
sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense module, 179
sage.rings.polynomial.padics.polynomial_padic_capped_absolute module, 186
sage.rings.polynomial.padics.polynomial_padic_capped_absolute_dense module, 583
sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense module, 179
sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense module, 186
sage.rings.polynomial.padics.polynomial_padic_flat module, 186
sage.rings.polynomial.polydict module, 419
sage.rings.polynomial.pbori.pbori module, 583
sage.rings.polynomial.polynomial_compiled module, 247
sage.rings.polynomial.polynomial_element module, 31
sage.rings.polynomial.polynomial_element_generic module, 112
sage.rings.polynomial.polynomial_integer_dense_flint module, 128
sage.rings.polynomial.polynomial_integer_dense_ntl module, 137
sage.rings.polynomial.polynomial_modn_dense_ntl module, 162
sage.rings.polynomial.polynomial_number_field module, 126
sage.rings.polynomial.polynomial_quotient_ring module, 223
sage.rings.polynomial.polynomial_quotient_ring_element module, 243
sage.rings.polynomial.polynomial_real_mpfr_dense module, 190
sage.rings.polynomial.real_roots module, 172
sage.rings.polynomial.real_roots module, 172
sage.rings.polynomial.refine_root module, 222
sage.rings.polynomial.symmetric_ideal module, 567
sage.rings.polynomial.symmetric_reduction module, 576
sage.rings.polynomial.term_order module, 253
sage.rings.polynomial.toy_buchberger module, 483
sage.rings.polynomial.toy_d_basis module, 492
sage.rings.polynomial.toyvariety module, 489

saturation() (sage.rings.polynomial.multi_polynomial_element.Polynomial_generic_sparse method), 451

scalar_lmult() (sage.rings.polynomial.pbori.pbori.BooleSet method), 591

set() (sage.rings.polynomial.pboration monetaryBooleSet method), 591

set_karatsuba_threshold() (sage.rings.polynomial.polynomial_ring.PolynomialRing_general method), 26

set_random_seed() (in module sage.rings.polynomial.pboration monetaryBooleSet), 640

scale_intvec_var() (in module sage.rings.polynomial.real_roots), 216

scaled_coeffs() (sage.rings.invariants.invariant_theory.Shift Generic sage.rings.polynomial.polynomial_element.Polynomial generic_dense method), 442

scaled_coeffs() (sage.rings.invariants.invariant_theory.Shift Generic sage.rings.polynomial.polynomial_element_generic.Polynomial generic method), 108

scaled_coeffs() (sage.rings.invariants.invariant_theory.Shift Generic sage.rings.polynomial.polynomial_element_generic.Polynomial generic method), 119

scaled_coeffs() (sage.rings.invariants.invariant_theory.Shift Generic sage.rings.polynomial.polynomial_element_generic.Polynomial generic method), 125

scaled_coeffs() (sage.rings.invariants.invariant_theory.Shift Generic sage.rings.polynomial.polynomial_element_generic.Polynomial generic method), 136

scaled_coeffs() (sage.rings.invariants.invariant_theory.Shift Generic sage.rings.polynomial.polynomial_element_generic.Polynomial generic method), 143

scaled_coeffs() (sage.rings.invariants.invariant_theory.Shift Generic sage.rings.polynomial.polynomial_element_generic.Polynomial generic method), 150

scaled_coeffs() (sage.rings.invariants.invariant_theory.Shift Generic sage.rings.polynomial.polynomial_element_generic.Polynomial generic method), 157

scaled_coeffs() (sage.rings.invariants.invariant_theory.Shift Generic sage.rings.polynomial.polynomial_element_generic.Polynomial generic method), 164

second() (sage.rings.invariants.invariant_theory.TwoAlgebraicForms method), 429

section() (sage.rings.fraction_field.FractionFieldEmbedding method), 498

section() (sage.rings.fraction_field.FpT.FpT coerce shift) (sage.rings.polynomial.polynomial_zz_pex.Polynomial_template method), 174

section() (sage.rings.fraction_field.FpT.Polyring_FpT coerce shift) (sage.rings.polynomial.polynomial_zz_pex.Polynomial_template method), 189

section() (sage.rings.fraction_field.FpT.ZZ_FpT coerce shift) (sage.rings.polynomial.polynomial_zz_pex.Polynomial_ZZ_pEX method), 187

section() (sage.rings.polynomial.flatten.FlatteningMorphism shrink_bp) (sage.rings.polynomial.real_roots.island method), 206

section() (sage.rings.polynomial.polynomial_element.Polynomial singular by empty blocks) (sage.rings.polynomial.term_order.TermOrder method), 265

select() (in module sage.rings.polynomial.toy_buchberger), 487

select() (in module sage.rings.polynomial.toy_d_basis), 494

select() (sage.rings.polynomial.pbori.pbori.GroebnerStrategy method), 630

selmer_generators() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic method), 240

slope_factorization() (sage.rings.polynomial.multi_polynomial_ideal.libsingular), 114

set() (sage.rings.polynomial.pbori.pbori.BooleMonomial slope range) (sage.rings.polynomial.real_roots.interval_benchmark polynomial method), 201

set() (sage.rings.polynomial.pbori.pbori.BoolePolynomial slope range) (sage.rings.polynomial.real_roots.interval_benchmark polynomial method), 201

Index 675
SymmetricIdeal (class in Sage.rings.polynomial.symmetric_ideal), 419
stretch() (sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial_sparse method), 566
SymmetricReductionStrategy (class in Sage.rings.polynomial.symmetric_reduction), 578
sub_m_mul_q() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 414
subresultants() (sage.rings.polynomial.symmetric_ideal.SymmetricIdeal method), 303
subresultants() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict method), 100
T_prime_covariant() (sage.rings.invariants.invariant_theory.BinaryQuintic method), 445
T_prime_covariant() (sage.rings.invariants.invariant_theory.TwoQuaternaryQuadratics method), 473
T_invariant() (sage.rings.invariants.invariant_theory.TernaryCubic method), 467
T_prime_invariant() (sage.rings.invariants.invariant_theory.TwoQuaternaryQuadratics method), 474
tail() (sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomialSparse method), 567
tailreduce() (sage.rings.polynomial.symmetric_reduction.SymmetricReductionStrategy method), 581
tau_covariant() (sage.rings.invariants.invariant_theory.BinaryQuintic method), 452
taylor_shift_intvec() (in module sage.rings.polynomial.real_roots), 217
tensor_with_ring() (sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomialElement method), 554
tensor_with_ring() (sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomialRing methods), 557
term_reduce() (sage.rings.polynomial.polydict.PolyDict method), 430
term_order() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial method), 523
term_order() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing method), 303

terminal_one() (sage.rings.polynomial.pbori.pbori.CCuddNavigator method), 535
termOrder (class in sage.rings.polynomial.term_order), 258
termOrder_from_pb_order() (in module sage.rings.polynomial.pbori.pbori), 634
termorder_from_singular() (in module sage.rings.polynomial.term_order), 272
terms() (sage.rings.polynomial.pbori.BooleanPolynomial method), 612
ternary_biquadratic() (sage.rings.invariants.invariant_theory.InvariantTheoryFactory method), 460
ternary_cubic() (sage.rings.invariants.invariant_theory.InvariantTheoryFactory method), 460
ternary_quadratic() (sage.rings.invariants.invariant_theory.InvariantTheoryFactory method), 461
TernaryCubic (class in sage.rings.invariants.invariant_theory), 466
ternary_quadratic() (sage.rings.invariants.invariant_theory.InvariantTheoryFactory method), 461
TernaryQuadratic (class in sage.rings.invariants.invariant_theory), 466
transvectant() (in module sage.rings.invariants.invariant_theory), 478
triangular_decomposition() (sage.rings.polynomial.multi_polynomial理想的Ideal method), 465
triangular_factorization() (in module sage.rings.polynomial.toy_variety), 491
transform() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial method), 543
transformed() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal method), 364
transformed_basis() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal method), 364
trace() (sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRingElement method), 430
trace_polynomial() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal method), 103
total_degree() (sage.rings.polynomial.multi_polynomial_element.MPolynomial method), 322
total_degree() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial method), 415
total_degree() (sage.rings.polynomial.pbori.pbori.BooleanPolynomial method), 612
then_branch() (sage.rings.polynomial.pbori.pbori.CCuddNavigator method), 365
tilde() (sage.rings.polynomial.polynomial_element.Polynomial method), 109
trunc() (sage.rings.polynomial.pbori.pbori.CCuddNavigator method), 365
trunc() (sage.rings.polynomial.polynomial_element.Polynomial method), 120
trunc() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial method), 125
trunc() (sage.rings.polynomial.real_roots.polynomial_element_generic.Polynomial method), 125
trunc() (sage.rings.polynomial.real_roots.polynomial_modn_dense_ntl.Polynomial method), 166
trunc() (sage.rings.polynomial.real_roots.polynomial_modn_dense_flint.Polynomial method), 169
trunc() (sage.rings.polynomial.real_roots.polynomial_rational_flint.Polynomial method), 154
trunc() (sage.rings.polynomial.real_roots.polynomial_mpfr_dense.Polynomial method), 174
trunc() (sage.rings.polynomial.real_roots.polynomial_zmod_flint.Polynomial method), 156
trunc() (sage.rings.polynomial.real_roots.polynomial_zz_pex.Polynomial method), 198
try_split() (sage.rings.polynomial.polydict.PolyDict method), 535
try_rand_split() (sage.rings.polynomial.polydict.PolyDict method), 535
try_random() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial method), 535
ZZ_FpT_coerce (class in sage.rings.fraction_field_FpT), 514