# CONTENTS

1 Polynomial Rings 1
2 Univariate Polynomials 9
3 Multivariate Polynomials 281
4 Rational Functions 555
5 Laurent Polynomials 575
6 Infinite Polynomial Rings 601
7 Boolean Polynomials 639
8 Indices and Tables 701

Python Module Index 703
Index 705
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
# needs sage.rings.polynomial.pbori
sage: R.<x, y, z> = BooleanPolynomialRing(); R  # indirect doctest
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
sage: R = BooleanPolynomialRing(5, 'x', order='deglex(3),deglex(2)')  # needs sage.rings.polynomial.pbori
sage: R.term_order()  # needs sage.rings.polynomial.pbori
Block term order with blocks:
(Degree lexicographic term order of length 3,
 Degree lexicographic term order of length 2)
```
Polynomials, Release 10.1

sage: R = BooleanPolynomialRing(3, 'x', order='degneglex')  # needs sage.rings.polynomial.pbori
sage: R.term_order()  # needs sage.rings.polynomial.pbori
Degree negative lexicographic term order

sage: BooleanPolynomialRing(names=('x','y'))  # needs sage.rings.polynomial.pbori
Boolean PolynomialRing in x, y

sage: BooleanPolynomialRing(names='x,y')  # needs sage.rings.polynomial.pbori
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.\langle w \rangle = 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
sage: R.\langle abc \rangle = PolynomialRing(QQ, sparse=True); R
Sparse Univariate Polynomial Ring in abc over Rational Field
sage: R.\langle w \rangle = 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.\langle y \rangle = 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:
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:

sage: R is PolynomialRing(QQ, 'zz')
True

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']
False
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:

```python
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, ...)

```python
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
sage: T = PolynomialRing(QQ, ('a','b','c')); T
Multivariate Polynomial Ring in a, b, c over Rational Field
```

All three rings are identical:

```python
sage: R is S
True
sage: S is T
True
```

There is a unique polynomial ring with each term order:

```python
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.

```python
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:

```python
sage: PolynomialRing(QQ, "x", implementation="singular")
# needs sage.libs.singular
Multivariate Polynomial Ring in x over Rational Field
sage: P.<x> = PolynomialRing(QQ, implementation="singular"); P
# needs sage.libs.singular
Multivariate Polynomial Ring in x over Rational Field
```

3. PolynomialRing(base_ring, n, names, ...) (where the arguments n and names may be reversed)
Polynomials, Release 10.1

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
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
˓→needs sage.libs.pari
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

#␣

By calling the inject_variables() method, all those variable names are available for interactive use:
sage: R.inject_variables()
˓→needs sage.libs.pari
Defining 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
sage: (x2 + x41 + x71)^2
˓→needs sage.libs.pari
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:

6

Chapter 1. Polynomial Rings


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 x 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: # needs sage.rings.real_mpfr  
sage: RR['x']  
Univariate Polynomial Ring in x over Real Field with 53 bits of precision  
sage: RR['x,y']  
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
```
sage.rings.polynomial.polynomial_ring_constructor.polynomial_default_category(n_variables)
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()
True
sage: polynomial_default_category(Fields(),2) is UniqueFactorizationDomains() &→CommutativeAlgebras(Fields()).Infinite()
True
sage: QQ['t'].category() is EuclideanDomains() & CommutativeAlgebras(QQ.category()).Infinite()
True
sage: QQ['s','t'].category() is UniqueFactorizationDomains() &→CommutativeAlgebras(QQ.category()).Infinite()
True
sage: QQ['s']['t'].category() is UniqueFactorizationDomains() &→CommutativeAlgebras(QQ['s'].category()).Infinite()
True
```

sage.rings.polynomial.polynomial_ring_constructor.pickle_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 github issue #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 github issue #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: # needs sage.combinat sage.modules
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
```

github issue #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: S.0 + R.0
2*x
```

(continues on next page)
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')  # needs sage.libsntl
sage: S.<x> = PolynomialRing(ZZ, implementation='FLINT')  # needs sage.libs.flint sage.libs.ntl
sage: (S.0 + R.0).parent()  # needs sage.libs.flint sage.libs.ntl
Univariate Polynomial Ring in x over Finite Field of size 5
```

```python
element_class=None,
category=None)

Bases: PolynomialRing_cdvr, 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,
implementation=None,
element_class=None,
category=None)

Bases: 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,
implementation=None,
element_class=None,
category=None)

Bases: PolynomialRing_general, CommutativeAlgebra
Univariate polynomial ring over a commutative ring.

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

Univariate Quotient Polynomial Ring in xbar over Rational Field with modulus x^2 - 1

# needs sage.libs.pari

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

# needs sage.libs.pari

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

# needs sage.modules

sage: W.polynomial_ring() == R

True

class sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_finite_field(base_ring, name='x',
                                                                                   element_class=None,
                                                                                   implementation=None)

Bases: PolynomialRing_field

Univariate polynomial ring over a finite field.

EXAMPLES:

```python
sage: R = PolynomialRing(GF(27, 'a'), 'x')

# needs sage.rings.finite_rings

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' or None: 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 \( \text{self} \).

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: f = GF(5^3, 'a')['x'].irreducible_element(2)
sage: f.degree()
2
sage: f.is_irreducible()
True
sage: R = GF(19)[x]
sage: R.irreducible_element(21, algorithm="first_lexicographic")
x^21 + x + 5
sage: R = GF(5^2, 'a')['x']
sage: R.irreducible_element(17, algorithm="first_lexicographic")
x^17 + a*x + 4*a + 3
```

AUTHORS:
- Peter Bruin (June 2013)
- Jean-Pierre Flori (May 2014)

class \texttt{sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_mod_n}(base\_ring, name=None, element\_class=None, implementation=None, category=None)

Bases: \texttt{PolynomialRing\_commutative}

\texttt{modulus}()

EXAMPLES:

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

\texttt{residue\_field}(ideal, names=None)

Return the residue finite field at the given ideal.

EXAMPLES:

```
sage: # needs sage.libs.ntl
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: # needs sage.libs.ntl
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, element_class=None, category=None)

Bases: PolynomialRing_dense_finite_field, PolynomialRing_dense_mod_n,
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 pari:ffprimroot function from PARI.
  – 'first_lexicographic': return the lexicographically smallest irreducible polynomial of degree 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 \texttt{algorithm} is \texttt{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 \texttt{minimal_weight} is used if \( p = 2 \), and the algorithm \texttt{'adleman-lenstra'} if \( p > 2 \).

\textbf{OUTPUT:}

A monic irreducible polynomial of degree \( n \) in \texttt{self}.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: # needs sage.rings.finite_rings
sage: GF(5)['x'].irreducible_element(2)
x^2 + 4*x + 2
sage: GF(5)['x'].irreducible_element(2, algorithm="adleman-lenstra")
x^2 + x + 1
sage: GF(5)['x'].irreducible_element(2, algorithm="primitive")
x^2 + 4*x + 2
sage: GF(5)['x'].irreducible_element(32, algorithm="first_lexicographic")
x^32 + 2
sage: GF(5)['x'].irreducible_element(32, algorithm="conway")
Traceback (most recent call last):
 ... RuntimeError: requested Conway polynomial not in database.
sage: GF(5)['x'].irreducible_element(32, algorithm="primitive")
x^32 + ...
\end{verbatim}

In characteristic 2:

\begin{verbatim}
sage: GF(2)['x'].irreducible_element(33)  # needs sage.rings.finite_rings
x^33 + x^13 + x^12 + x^11 + x^10 + x^8 + x^6 + x^3 + 1
sage: GF(2)['x'].irreducible_element(33, algorithm="minimal_weight")  # needs sage.rings.finite_rings
x^33 + x^10 + 1
\end{verbatim}

In degree 1:

\begin{verbatim}
sage: GF(97)['x'].irreducible_element(1)  # needs sage.rings.finite_rings
x + 96
sage: GF(97)['x'].irreducible_element(1, algorithm="conway")  # needs sage.rings.finite_rings
x + 92
sage: GF(97)['x'].irreducible_element(1, algorithm="adleman-lenstra")  # needs sage.rings.finite_rings
x
\end{verbatim}

\textbf{AUTHORS:}

- Peter Bruin (June 2013)
- Jeroen Demeyer (September 2014): add “ffprimroot” algorithm, see github issue #8373.
A class for dense polynomial ring over p-adic fields
class sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_padic_ring_capped_relative(base_ring, name=None, implementation=None, element_class=None, category=None):

Bases: PolynomialRing_dense_padic_ring_generic

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

Bases: PolynomialRing_dense_padic_ring_generic

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

Bases: PolynomialRing_cdvr

A class for dense polynomial ring over p-adic rings

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

Bases: PolynomialRing_integral_domain, 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 \(\text{self}\), 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 \(\text{self}\).

• **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 \(\text{points}\) (see \(\text{lantanle_polynomial()}\)). These are the coefficients \(F_{0,0}, F_{1,1}, \ldots, F_{n,n}\) in the base ring of \(\text{self}\) 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 \(F_{i,i}\). This example is taken from Example 1, page 121 of [BF2005]:

```
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: 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: 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]
sage: R.divided_difference(points, full_table=True)
[[-3],
 [0, 3],
 [15, 15, 6],
 [48, 33, 9, 1],
```

(continues on next page)
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.
  – '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

(continues on next page)
Polynomials, Release 10.1

sage: # needs sage.rings.finite_rings
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: # needs sage.rings.finite_rings
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: # needs sage.rings.finite_rings
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

class sage.rings.polynomial.polynomial_ring.PolynomialRing_general(base_ring, name=None, sparse=False, implementation=None, element_class=None, category=None)

Bases: Algebra
Univariate polynomial ring over a ring.

**base_extend**(*R*)

Return the base extension of this polynomial ring to *R*.

**EXAMPLES:**

```
sage: # needs sage.rings.real_mpfr
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
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:**

```
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:**

```
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()
#...
```

(continues on next page)
needs sage.rings.finite_rings

completion\((p=None, \text{prec}=20, \text{extras}=None)\)
Return the completion of self with respect to the irreducible polynomial \(p\).
Currently only implemented for \(p=\text{self.gen()}\) (the default), i.e. you can only complete \(R[x]\) with respect to \(x\), the result being a ring of power series in \(x\). The \text{prec} variable controls the precision used in the power series ring. If \text{prec} is \(\infty\), then this returns a LazyPowerSeriesRing.

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)
sage: g = 1 / PP(f); g
1 + 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 + O(x^20)
sage: 1 / g
1 - x + O(x^20)
sage: # needs sage.combinat
sage: PP = P.completion(x, \text{prec}=\infty); PP
Lazy Taylor Series Ring in x over Rational Field
sage: g = 1 / PP(f); g
1 + x + x^2 + O(x^3)
sage: 1 / g == f
True
```

cyclotomic_polynomial\((n)\)
Return the \(n\)th cyclotomic polynomial as a polynomial in this polynomial ring. For details of the implementation, see the documentation for \texttt{sage.rings.polynomial.cyclotomic.cyclotomic_coeffs()}.

EXAMPLES:

```python
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)
```

construction()
Return the construction functor.

cyclotomic_polynomial\((n)\)
Return the \(n\)th cyclotomic polynomial as a polynomial in this polynomial ring. For details of the implementation, see the documentation for \texttt{sage.rings.polynomial.cyclotomic.cyclotomic_coeffs()}.
\[ x^4 + 6x^2 + 1 \]
\[
\textbf{sage: } \text{S.cyclotomic_polynomial(1)}
\]
\[ x + 6 \]

\textbf{extend_variables(added_names, order='degrevlex')} 

Return a multivariate polynomial ring with the same base ring but with \texttt{added_names} as additional variables.

**EXAMPLES:**

\[
\textbf{sage: } \text{R.<x> = ZZ[]; R}
\]
Univariate Polynomial Ring in \(x\) over Integer Ring

\[
\textbf{sage: } \text{R.extend_variables('y, z')}
\]
Multivariate Polynomial Ring in \(x, y, z\) over Integer Ring

\[
\textbf{sage: } \text{R.extend_variables(('y', 'z'))}
\]
Multivariate Polynomial Ring in \(x, y, z\) over Integer Ring

\textbf{flattening_morphism()}

Return the flattening morphism of this polynomial ring

**EXAMPLES:**

\[
\textbf{sage: } \text{QQ['a', 'b'][x].flattening_morphism()}
\]
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

\[
\textbf{sage: } \text{QQ['x'].flattening_morphism()}
\]
Identity endomorphism of Univariate Polynomial Ring in \(x\) over Rational Field

\textbf{gen(n=0)}

Return the indeterminate generator of this polynomial ring.

**EXAMPLES:**

\[
\textbf{sage: } \text{R.<abc> = Integers(8)[]; R}
\]
Univariate Polynomial Ring in \(abc\) over Ring of integers modulo 8

\[
\textbf{sage: } \text{t = R.gen(); t}
\]
abc

\[
\textbf{sage: } \text{t.is_gen()}
\]
True

An identical generator is always returned.

\[
\textbf{sage: } \text{t is R.gen()}
\]
True

\textbf{gens_dict()}

Return a dictionary whose entries are \{name:variable,...\}, where \texttt{name} stands for the variable names of this object (as strings) and \texttt{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: # needs sage.libs.ntl
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
sage: R.<W> = PolynomialRing(QQ, sparse=True); R
Sparse Univariate Polynomial Ring in W over Rational Field
(continues on next page)
is_unique_factorization_domain(proof=True)

EXAMPLES:

sage: ZZ[x].is_unique_factorization_domain()
True
sage: Integer(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:

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:

sage: R.<x> = QQ[]
sage: R.krull_dimension()
1

sage: S.<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

OUTPUT: an iterator

EXAMPLES:

```sage
sage: # needs sage.rings.finite_rings
sage: P = PolynomialRing(GF(4, 'a'), 'y')
sage: for p in P.monics(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 + y
y^2 + y + a
y^2 + y + a + 1
y^2 + y + 1
sage: for p in P.monics(max_degree=1): print(p)
1
y
y + a
y + a + 1
y + 1
sage: for p in P.monics(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

```sage
monomial(exponent)
```

Return the monomial with the exponent.

INPUT:

• exponent – nonnegative integer

EXAMPLES:

```sage
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)
```

(continues on next page)
\texttt{sage}: \texttt{R.monomial(m\textunderscore degree()) == m}
\texttt{True}

\texttt{ngens()}

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

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage}: \texttt{R.<z> = Integers(8)[]}; R
Univariate Polynomial Ring in z over Ring of integers modulo 8
\texttt{sage}: \texttt{R.ngens()}
1
\end{verbatim}

\texttt{parameter()}

Return the generator of this polynomial ring.

This is the same as \texttt{self.gen()}.

\texttt{polynomials}(\textit{of\_degree=\textit{None}}, \textit{max\_degree=\textit{None}})

Return an iterator over the polynomials of specified degree.

\textbf{INPUT:} Pass exactly one of:

\begin{itemize}
  \item \texttt{max\_degree} - an int; the iterator will generate all polynomials which have degree less than or equal to \texttt{max\_degree}
  \item \texttt{of\_degree} - an int; the iterator will generate all polynomials which have degree \texttt{of\_degree}
\end{itemize}

\textbf{OUTPUT:} an iterator

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage}: \texttt{P = PolynomialRing(GF(3), 'y')}\n\texttt{sage}: \texttt{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
\texttt{sage}: \texttt{for p in P.polynomials(max\_degree=1): print(p)}
0
1
2
\end{verbatim}
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 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:

sage: R.<x> = ZZ[]
sage: f = R.random_element(10, 5, 10)
sage: f.degree()
10
sage: f.parent() is R
True
sage: all(a in range(5, 10) for a in f.coefficients())
True
sage: R.random_element(6).degree()
6

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:

sage: R.random_element(degree=(0, 8)).degree() in range(0, 9)
True
sage: found = [False]*9
sage: while not all(found):
    ....:     found[R.random_element(degree=(0, 8)).degree()] = True

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

sage: while R.random_element(degree=(-1,2), x=-1, y=1) != R.zero():
    ....:     pass

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.

EXAMPLES:

```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`

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

**Bases:** `PolynomialRing_commutative, PolynomialRing_singular_repr, IntegralDomain`

**construction()**

Return the construction functor.

**EXAMPLES:**
weil_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

$$T^d f\left(\frac{q}{T}\right) = sq^{d/2} f(T)$$

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
- $sign$ – integer (default 1), the sign $s$ of the functional equation
- $lead$ – 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 $a \equiv a \pmod{b}$; 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 `is_weil_polynomial()` to test whether or not the given polynomial is a Weil polynomial.

**EXAMPLES:**

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

We do not require Weil polynomials to be monic. This example generates Weil polynomials associated to
K3 surfaces over $\mathbb{F}_2$ of Picard number at least 12:

sage: R.<T> = QQ[]
sage: l = R.weil_polynomials(10, 1, lead=2)       # needs sage.libs.flint
sage: len(l)                                        # needs sage.libs.flint
4865
sage: l[len(l)//2]                                  # needs sage.libs.flint
2*T^10 + T^8 + T^6 + 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:

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: type(R)                                        # needs sage.libs.singular
<class 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomialRing_libsingular'>

This polynomial ring is not univariate.

sage: is_PolynomialRing(ZZ['x,y,z'])
False
sage: is_MPolynomialRing(ZZ['x,y,z'])
True
sage: is_PolynomialRing(ZZ['\omega'])
True

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: # needs sage.libs.singular
sage: R.<\omega> = PolynomialRing(ZZ, implementation="singular"); R
Multivariate Polynomial Ring in \omega over Integer Ring
sage: is_PolynomialRing(R)
False
sage: type(R)
<class 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomialRing_libsingular'>

2.1. Univariate Polynomials and Polynomial Rings
sage.rings.polynomial.polynomial_ring.polygen(ring_or_element, name='x')
Return a polynomial indeterminate.

INPUT:

• polygen(base_ring, name="x")
• polygen(ring_element, name="x")

If the first input is a ring, return a polynomial generator over that ring. If it is a ring element, return a polynomial generator over the parent of the element.

EXAMPLES:

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.rings.polynomial.polynomial_ring.polygens(base_ring, names='x', *args)
Return indeterminates over the given base ring with the given names.

EXAMPLES:

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:

sage: polysgens(QQ, 'x', 4)
(x0, x1, x2, x3)

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: RingHomomorphism_from_base

The canonical ring homomorphism from \( R[x] \) to \( S[x] \) induced by a ring homomorphism from \( R \) to \( S \).

EXAMPLES:

```
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:

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

is_surjective()

Return whether this morphism is surjective.

EXAMPLES:

```
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: moved Polynomial_generic_dense to Cython
- Miguel Marco: implemented resultant in the case where PARI fails
- Simon King: used a faster way of conversion from the base ring
- Kwankyu Lee (2013-06-02): enhanced quo_rem()
- 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
- Simon King (2013-10): implemented copying of PolynomialBasingInjection
- Bruno Grenet (2014-07-13): enhanced quo_rem()
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 github issue #26421)

class sage.rings.polynomial.polynomial_element.ConstantPolynomialSection
Bases: Map
This class is used for conversion from a polynomial ring to its base ring.
Since github issue #9944, it calls the constant_coefficient() method, which can be optimized for a particular polynomial type.

```sage
sage: P0.<y_1> = GF(3)[]
sage: P1.<y_2,y_1,y_0> = GF(3)[]
sage: P0(-y_1)
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)
<class '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: CommutativePolynomial
A polynomial.

```sage
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)
<class 'sage.rings.polynomial.polynomial_element.Polynomial_generic_dense'>
sage: p = (y+1)^10; p(1)
1024
```

Chapter 2. Univariate Polynomials
_add_(right)
Add two polynomials.

EXAMPLES:

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

_sub_(other)
Default implementation of subtraction using addition and negation.

_lmul_(left)
Multiply self on the left by a scalar.

EXAMPLES:

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

_rmul_(right)
Multiply self on the right by a scalar.

EXAMPLES:

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

_mul_(right)

EXAMPLES:

```
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: D.<s> = PolynomialRing(C)
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)
```
\[(t^4 + O(t^5))s^4 + (2t^3 + O(t^4))s^3 + (3t^2 + O(t^3))s^2 + (2t + O(t^2))s + 1\]

__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:

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:

sage: # needs sage.libs.pari
sage: f = cyclotomic_polynomial(30)
sage: f.adams_operator(7)==f
True
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)  # needs sage.libs.singular
\[ x^2 + 1024 \]

When `self` is monic, the output will have leading coefficient ±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()  # needs sage.libs.singular
\[-1 \times (x - c^3) \times (x - b^3) \times (x - a^3)\]
sage: f.adams_operator(3, monic=True).factor()  # needs sage.libs.singular
\[(x - c^3) \times (x - b^3) \times (x - a^3)\]

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: # needs sage.libs.pari
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
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 \( \text{abs(degree)} \) over the ground field. If negative, also assumes that all factors of this polynomial are of degree \( \text{abs(degree)} \). 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:

```python
sage: # needs sage.rings.finite_rings
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)
```

```python
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 + a, 1),
 (a^5 + a^4 + 7*a^3 + 2*a^2 + 10*a, 1),
 (9*a^5 + 5*a^4 + 10*a^3 + 8*a^2 + 3*a + 1, 1),
 (2*a^5 + 8*a^4 + 3*a^3 + 6*a + 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 + a^2 + 10*a + 4, 1)]
```

```python
sage: # needs sage.rings.finite_rings
sage: g = (x-1)*(x^2 + 3*x + 9) * (x^5 + 5*x^4 + 8*x^3 + 5*x^2 + 3*x + 5)
```

```python
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
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:

```python
sage: R.<x> = QQ[]
sage: x.args()
(x,)
```

A constant polynomial has no variables, but still takes a single argument.

```python
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 the coefficient ring of `self` to \( R \).

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = x^3 - 17*x + 3
sage: f.base_extend(GF(7))
Traceback (most recent call last):
  ...  
TypeError: no such base extension
sage: 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[]
sage: x.base_ring()
Integer Ring
sage: (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)  # needs sage.rings.number_field
sage: f = K.defining_polynomial()  # needs sage.rings.number_field
sage: f.change_ring(GF(7))  # needs sage.rings.finite_rings sage.rings.number_field
x^2 + x + 1

sage: R.<x> = QQ[]
sage: f = x^2 + 1
sage: f.change_ring(QQ.embeddings(CC)[0])  # needs sage.rings.real_mpfr
x^2 + 1.00000000000000
```

---

2.1. Univariate Polynomials and Polynomial Rings
change_variable_name(var)

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

EXAMPLES:

```
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:

```
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 uses the appropriate precision from the input coefficients.

EXAMPLES:

```
sage: x = polygen(ZZ)
sage: (x^3 - 1).complex_roots()  # note: low order bits slightly different on ppc.  # needs sage.rings.real_mpfr
[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, $f \ast f \ast \cdots \ast f$ where this is $f$ and $f \ast f = f\text{.composed_op}(f, \text{operator.mul})$.

EXAMPLES:
sage: R.<a,b,c> = ZZ[

sage: x = polygen(R)

sage: f = (x - a) * (x - b) * (x - c)

sage: f.compose_power(2).factor()  # needs sage.libs.singular sage.modules
(x - c^2) * (x - b^2) * (x - a^2) * (x - b*c)^2 * (x - a*c)^2 * (x - a*b)^2

sage: # needs sage.libs.singular sage.modules

sage: x = polygen(QQ)

sage: f = x^2 - 2*x + 2

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:**

```
sage: Pol.<x> = CBF[]  # needs sage.libs.flint

sage: (1 + x + x^2/2 + x^3/6 + x^4/24 + x^5/120).compose_trunc(1 + x, 2)  # needs sage.libs.flint
([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 of polynomials
```

**composed_op** *(p1, p2, op=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:**

2.1. Univariate Polynomials and Polynomial Rings
• \(p_2\) – univariate polynomial belonging to the same polynomial ring as this polynomial

• \(op\) – \texttt{operator.OP} where \(OP=\text{add}\) or \(\text{sub}\) or \(\text{mul}\) or \(\text{truediv}\).

• \(algorithm\) – 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 \(\mathbb{Z}\) or \(\mathbb{Q}\). Otherwise the latter is used.

• \(monic\) – whether to return a monic polynomial. If \(\text{True}\) 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 \(\text{Res}_y(p_1(x-y), p_2(y))\). However, the method from [BFSS2006] using series expansions is asymptotically much faster. Note that the algorithm BFSS with polynomials with coefficients in \(\mathbb{Z}\) needs to perform operations over \(\mathbb{Q}\).

**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 \texttt{_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:**

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

This function works over any field. However for base rings other than \(\mathbb{Z}\) and \(\mathbb{Q}\) only the resultant algorithm is available:

```python
sage: # needs sage.rings.number_field
sage: x = polygen(QQbar)
sage: p1 = x**2 - AA(2).sqrt()
sage: p2 = x**3 - AA(3).sqrt()
sage: r1 = p1.roots(multiplicities=False)
sage: r2 = p2.roots(multiplicities=False)
sage: p = p1.composed_op(p2, operator.add); p
\rightarrow x^6 - 4.242640687192857*x^4 - 3.464101615137755*x^3 + 6*x^2
\rightarrow -14.69693845669907*x + 0.1715728752538099?
```

\(\text{all}(p(x+y).is_zero() \text{ for } x \text{ in } r1 \text{ for } y \text{ in } r2)\)

True

(continues on next page)
sage: x = polygen(GF(2))
sage: p1 = x**2 + x - 1
sage: p2 = x**3 + x - 1
sage: p_add = p1.composed_op(p2, operator.add); p_add  # needs sage.libs.singular
x^6 + x^5 + x^3 + x^2 + 1
sage: p_mul = p1.composed_op(p2, operator.mul); p_mul  # needs sage.libs.singular
x^6 + x^4 + x^2 + x + 1
sage: p_div = p1.composed_op(p2, operator.truediv); p_div  # needs sage.libs.singular
x^6 + x^5 + x^4 + x^2 + 1
sage: # needs sage.rings.finite_rings
sage: K = GF(2**6, 'a')
sage: r1 = p1.roots(K, multiplicities=False)
sage: r2 = p2.roots(K, multiplicities=False)
sage: all(p_add(x1+x2).is_zero() for x1 in r1 for x2 in r2)  # needs sage.libs.singular
True
sage: all(p_mul(x1*x2).is_zero() for x1 in r1 for x2 in r2)  # needs sage.libs.singular
True
sage: all(p_div(x1/x2).is_zero() for x1 in r1 for x2 in r2)  # needs sage.libs.singular
True

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

(continues on next page)
sage: f.content_ideal().gen()
2

\textbf{cyclotomic\_part()}
\begin{quote}
Return the product of the irreducible factors of this polynomial which are cyclotomic polynomials.
\end{quote}
The algorithm assumes that the polynomial has rational coefficients.

\textbf{See also:}
\begin{quote}
is\_cyclotomic() is\_cyclotomic\_product() has\_cyclotomic\_factor()
\end{quote}

\textbf{EXAMPLES:}
\begin{verbatim}
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)
sage: pol.cyclotomic_part()
x^8 + 2*x^4 + 1
sage: P.<x> = PolynomialRing(QQ)
sage: pol = (x^4 + 1)^2 * (x^4 + 2)
sage: pol.cyclotomic_part()
x^8 + 2*x^4 + 1
sage: pol = (x - 1) * x * (x + 2)
sage: pol.cyclotomic_part()
x - 1
\end{verbatim}

\textbf{degree}(\textit{gen=\textit{None}})
\begin{quote}
Return the degree of this polynomial. The zero polynomial has degree \(-1\).
\end{quote}

\textbf{EXAMPLES:}
\begin{verbatim}
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
sage: x = QQ['x'].0
sage: f = 2006*x^2006 - x^2 + 3
sage: f.degree()
2006
sage: f = 0*x
sage: f.degree()
0
\end{verbatim}

AUTHORS:

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

\textbf{denominator()}

Return a denominator of \texttt{self}.

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

Note that some subclasses may implement their own \texttt{denominator()} method. For example, see \texttt{sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint}

\textbf{Warning}: This is not the denominator of the rational function defined by \texttt{self}, which would always be 1 since \texttt{self} is a polynomial.

\textbf{EXAMPLES}:

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

\begin{verbatim}
sage: R.<x> = ZZ[]
sage: f = x^3 + 17*x + 1
sage: f.denominator()
1
\end{verbatim}

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

\begin{verbatim}
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
\end{verbatim}

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 \texttt{denominator()} method.

\begin{verbatim}
sage: R.<x> = RR[]
sage: f = x + RR('0.3'); f
x + 0.300000000000000
sage: f.denominator()
1.00000000000000
\end{verbatim}

Check that the denominator is an element over the base whenever the base has no \texttt{denominator()} method. This closes github issue #9063.

\begin{verbatim}
sage: R.<a> = GF(5)[]
sage: x = R(0)
sage: x.denominator()
1
\end{verbatim}
sage: type(x.denominator())
<class '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: 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
dict()
Return a sparse dictionary representation of this univariate polynomial.

EXAMPLES:

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.
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
```

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
```

(continues on next page)
Polynomials, Release 10.1

(continued from previous page)

\[3 \cdot t^4 \cdot x^2 + 2 \cdot t^3 \cdot x\]

```
sage: f.derivative(t)
4 \cdot t^3 \cdot x^3 + 3 \cdot t^2 \cdot x^2
```

\textbf{discriminant()}

Return the discriminant of self.

The discriminant is

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

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

\textbf{OUTPUT}: An element of the base ring of the polynomial ring.

\textbf{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 \texttt{resultant()}.

\textbf{EXAMPLES}:

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

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

```
sage: R.<x> = QQ[]
sage: f = 2 \cdot x^3 + x + 1
sage: d = f.discriminant(); d
# needs sage.libs.pari
-116
```

We can compute discriminants over univariate and multivariate polynomial rings:

```
sage: R.<a> = QQ[]
sage: S.<x> = R[]
sage: f = a \cdot x + x + a + 1
sage: d = f.discriminant(); d
# needs sage.libs.pari
1
sage: d.parent() is R
# needs sage.libs.pari
True
```
Polynomials, Release 10.1

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

dispersion\(\text{(other=None)}\)

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 \text{other} is \text{None}, compute the auto-dispersion of \text{self}, i.e., its dispersion with itself.

See also:

dispersion_set()\

EXAMPLES:

sage: Pol.<x> = QQ[]
sage: x.dispersion(x + 1)
# needs sage.libs.pari
1
sage: (x + 1).dispersion(x)
# needs sage.libs.pari
-Infinity
sage: # needs sage.libs.pari sage.rings.number_field sage.symbolic
sage: Pol.<x> = QQbar[]
sage: pol = Pol([sqrt(5), 1, 3/2])
sage: pol.dispersion()
0
sage: (pol*pol(x+3)).dispersion()
3

dispersion_set\(\text{(other=None)}\)

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 \text{other} is \text{None}, compute the auto-dispersion set of \text{self}, i.e., its dispersion set with itself.

ALGORITHM:

See Section 4 of Man & Wright [MW1994].

See also:

dispersion()\

EXAMPLES:
sage: Pol.<x> = QQ[]
sage: x.dispersion_set(x + 1)  # needs sage.libs.pari
[1]
sage: (x + 1).dispersion_set(x)  # needs sage.libs.pari
[]

\(sage: pol = x^3 + x - 7\)
\(sage: (pol*pol(x+3)^2).dispersion_set()\)  # needs sage.libs.pari
\[0, 3\]

\(\textbf{divides}(p)\)

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

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

\(\textbf{EXAMPLES:}\)

\(\textbf{sage: }R.<x> = ZZ[]\)
\(\textbf{sage: } (2*x + 1).divides(4*x**2 - 1)\)
\(\text{True}\)
\(\textbf{sage: } (2*x + 1).divides(4*x**2 + 1)\)
\(\text{False}\)
\(\textbf{sage: } (2*x + 1).divides(R(0))\)
\(\text{True}\)
\(\textbf{sage: } R(0).divides(2*x + 1)\)
\(\text{False}\)
\(\textbf{sage: } R(0).divides(R(0))\)
\(\text{True}\)
\(\textbf{sage: } S.<y> = R[]\)
\(\textbf{sage: } p = x * y**2 + (2*x + 1) * y + x + 1\)
\(\textbf{sage: } q = (x + 1) * y + (3*x + 2)\)
\(\textbf{sage: } q.divides(p)\)
\(\text{False}\)
\(\textbf{sage: } q.divides(p * q)\)
\(\text{True}\)
\(\textbf{sage: } R.<x> = Zmod(6)[]\)
\(\textbf{sage: } p = 4*x + 3\)
\(\textbf{sage: } q = 5*x**2 + x + 2\)
\(\textbf{sage: } p.divides(q)\)
\(\text{Traceback (most recent call last):}\)
\(\text{...}\)
\(\text{NotImplementedError: divisibility test only implemented for polynomials over an,}\)
\(\text{→integral domain}\)

\(\textbf{euclidean_degree()}\)

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 \textit{degree()}.

\(\textbf{EXAMPLES:}\)
```
sage: R.<x> = QQ[]
sage: x.euclidean_degree()
1
sage: R.<x> = ZZ[]
sage: x.euclidean_degree()
Traceback (most recent call last):
  ... Not ImplementedError
```

### 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 \(\mathbb{Q}\):

```
sage: x = QQ['x'].0
sage: f = (x^3 - 1)^2
sage: f.factor()  # needs sage.libs.pari
(x - 1)^2 * (x^2 + x + 1)^2
```

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

```
sage: f = 10*x^5 - 1
sage: f.factor()  # needs sage.libs.pari
(10) * (x^5 - 1/10)
sage: f = 10*x^5 - 10
sage: f.factor()  # needs sage.libs.pari
(10) * (x - 1) * (x^4 + x^3 + x^2 + x + 1)
```

Over \(\mathbb{Z}\) the irreducible factors need not be monic:
We factor a non-monic polynomial over a finite field of 25 elements:

\[
\text{sage: } \text{# needs sage.rings.finite_rings} \\
\text{sage: } k.<a> = GF(25) \\
\text{sage: } R.<x> = k[] \\
\text{sage: } f = 2^x^{10} + 2^x + 2^a \\
\text{sage: } F = f.factor(); F \\
\text{(2) * (x + a + 2) * (x^2 + 3^x + 4^a + 4) * (x^2 + (a + 1)^x + a + 2) * (x^5 + (3^a + 4)^x^4 + (3^a + 3)^x^3 + 2^a^x^2 + (3^a + 1)^x + 3^a + 1)}
\]

Notice that the unit factor is included when we multiply \( F \) back out:

\[
\text{sage: } \text{expand(F)} \\
\text{# needs sage.rings.finite_rings sage.symbolic} \\
2^x^{10} + 2^x + 2^a
\]

A new ring. In the example below, we set the special method \(_\text{factor_univariate_polynomial}()\) 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:

\[
\text{sage: } \text{# needs sage.libs.ntl} \\
\text{sage: } R.<x> = PolynomialRing(IntegerModRing(4), implementation="NTL") \\
\text{sage: } (x^2).factor() \\
\text{Traceback (most recent call last):} \\
\text{...} \\
\text{NotImplementedError: factorization of polynomials over rings with composite characteristic is not implemented} \\
\text{sage: } \text{def my_factor(f):} \\
\text{......: } \text{return f.change_ring(ZZ).factor()} \\
\text{sage: } R.base_ring()._factor_univariate_polynomial = my_factor \\
\text{sage: } (x^2).factor() \\
\text{# needs sage.libs.pari} \\
x^2 \\
\text{sage: } \text{del } R.base_ring()._factor_univariate_polynomial \text{ # clean up}
\]

Arbitrary precision real and complex factorization:

\[
\text{sage: } \text{# needs sage.libs.pari sage.rings.real_mpfr} \\
\text{sage: } R.<x> = RealField(100)[] \\
\text{sage: } F = factor(x^2 - 3); F \\
(x - 1.7320508075688772935274463415) * (x + 1.7320508075688772935274463415) \\
\text{sage: } \text{expand(F)} \\
x^2 - 3.0000000000000000000000000000 \\
\text{sage: } \text{factor(x^2 + 1)} \\
x^2 + 1.0000000000000000000000000000 \\
\text{sage: } \text{# needs sage.libs.pari sage.rings.real_mpfr}
\]

(continues on next page)
sage: R.<x> = ComplexField(100)[]
sage: F = factor(x^2 + 3); F
(x - 1.732050807568877293527463415*I) * (x + 1.732050807568877293527463415*I)
sage: expand(F)
x^2 + 3.0000000000000000000000000000
sage: factor(x^2 + 1)
(x - I) * (x + I)
sage: f = R(I) * (x^2 + 1); f
I*x^2 + I
sage: F = factor(f); F
(1.0000000000000000000000000000*I) * (x - I) * (x + I)
sage: expand(F)
I*x^2 + I

Over a number field:

sage: # needs sage.rings.number_field
sage: K.<z> = CyclotomicField(15)
sage: x = polygen(K)
sage: ((x^3 + z*x + 1)^3 * (x - z)).factor()
(x - z) * (x^3 + z*x + 1)^3
sage: cyclotomic_polynomial(12).change_ring(K).factor()
(x^2 - z^5 - 1) * (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)
* (x - 1/3*z - 2/3) * (x^3 + z*x + 1)^3

Over a relative number field:

sage: # needs sage.rings.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) * (t^5 + t + z)

Over the real double field:

sage: # needs numpy
sage: R.<x> = RDF[]
sage: (-2*x^2 - 1).factor()
(-2.0) * (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) * (x^2 - 1.9999934280563585*x + 0.9999934280995487)

The above output is incorrect because it relies on the \texttt{roots()} method, which does not detect that all the
roots are real:

```python
sage: f.roots() # abs tol 2e-5
← needs numpy
[(1.0000065719436413, 1)]
```

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

```python
sage: # needs numpy sage.rings.complex_double
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 - 0.9999942196864997 + 0.999987300803959*I)
* (x - 0.999918923847313 + 1.0000113554909125*I)
* (x + 0.999908759550227 - 1.0000069659624138*I)
* (x + 0.999985293216753 - 0.999986153831807*I)
* (x + 1.0000105947233 - 1.000004418654053*I)
```

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

```python
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 github issue #8544):

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

Factoring polynomials over the algebraic reals (see github issue #8544):

```python
sage: R.<x> = AA[]
← needs sage.rings.number_field
sage: (x^8 + 1).factor() # needs sage.rings.number_field
(x^2 - 1.8477590505022574?*x + 1.000000000000000?)
* (x^2 - 0.7653668647301795?*x + 1.000000000000000?)
* (x^2 + 0.7653668647301795?*x + 1.000000000000000?)
* (x^2 + 1.8477590505022574?*x + 1.000000000000000?)
```
**gcd**

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:**

```
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`:

```
sage: # needs sage.rings.number_field sage.symbolic
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: R._gcd_univariate_polynomial = lambda f, g: R(S(f).gcd(S(g)))

sage: p.gcd(q)
x + a
sage: del R._gcd_univariate_polynomial
```

Use multivariate implementation for polynomials over polynomial rings:

```
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)
```

(continues on next page)
→ needs sage.libs.singular
z + 2*x*y

sage: R.<x> = QQ[]
sage: S.<y> = R[]
sage: r = 2*x*y + 1
sage: p = r * (x - 1/2 * y)
sage: q = r * (x*y^2 - x + 1/3)
sage: p.gcd(q) #→ needs sage.libs.singular
2*x*y + 1

```python
global_height(prec=None)
```

Return the (projective) global height of the polynomial.

This returns the absolute logarithmic height of the coefficients thought of as a projective point.

**INPUT:**

* prec – desired floating point precision (default: default RealField precision).

**OUTPUT:** a real number.

**EXAMPLES:**

```
sage: R.<x> = PolynomialRing(QQ)
sage: f = 3*x^3 + 2*x^2 + x
sage: exp(f.global_height()) #→ needs sage.symbolic
3.00000000000000
```

Scaling should not change the result:

```
sage: R.<x> = PolynomialRing(QQ)
sage: f = 1/25*x^2 + 25/3*x + 1
sage: f.global_height() #→ needs sage.symbolic
6.43775164973640
sage: g = 100 * f
sage: g.global_height() #→ needs sage.symbolic
6.43775164973640
```

```
sage: R.<x> = PolynomialRing(QQbar) #→ needs sage.rings.number_field
sage: f = QQbar(i)*x^2 + 3*x
sage: f.global_height() #→ needs sage.rings.number_field
1.09861228866811
```

```
sage: # needs sage.rings.number_field
sage: R.<x> = PolynomialRing(QQ)
sage: K.<k> = NumberField(x^2 + 5)
```

(continues on next page)
sage: T.<t> = PolynomialRing(K)
sage: f = 1/1331 * t^2 + 5 * t + 7
sage: f.global_height()
9.13959596745043

sage: R.<x> = QQ[]
sage: f = 1/123*x^2 + 12
sage: f.global_height(prec=2)
# needs sage.symbolic
8.0

sage: R.<x> = QQ[]
sage: f = 0*x
sage: f.global_height()
# needs sage.rings.real_mpfr
0.000000000000000

**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:**

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

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:**

```python
sage: pol.<x> = PolynomialRing(Rationals())
sage: u = x^5 - 1; u.has_cyclotomic_factor()
True
sage: u = x^5 - 2; u.has_cyclotomic_factor()
False
sage: u = pol(cyclotomic_polynomial(7)) * pol.random_element()  # random
sage: u.has_cyclotomic_factor()  # random
True
```

`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:
Polynomials, Release 10.1

```sage
def 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:

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

In positive characteristic, the degree can drop in this case:

```sage
R.<x> = GF(2)[]
sage: f = x + 1
sage: f.homogenize(x)
0
```

For compatibility with the multivariate case, the parameter var can also be 0 to specify the variable in the polynomial ring:

```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
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: 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: \( g = f \text{.integral}() \) ; \( g \)

\[ \frac{1}{4}x^4 + \frac{1}{2}x^2 - 2x \]

sage: \( g \text{.parent}() \)

Univariate Polynomial Ring in \( x \) over Rational Field

This shows that the issue at github issue #7711 is resolved:

sage: \# needs sage.rings.finite_rings

sage: \( P \ll x, z \rr = \text{PolynomialRing}(GF(2147483647)) \)

sage: \( Q \ll y \rr = \text{PolynomialRing}(P) \)

sage: \( p = x + y + z \)

sage: \( p \text{.integral}() \)

\[ -1073741823y^2 + (x + z)y \]

sage: \# needs sage.rings.finite_rings

sage: \( P \ll x, z \rr = \text{PolynomialRing}(GF(\text{next_prime}(2147483647))) \)

sage: \( Q \ll y \rr = \text{PolynomialRing}(P) \)

sage: \( p = x + y + z \)

sage: \( p \text{.integral}() \)

\[ 1073741830y^2 + (x + z)y \]

A truly convoluted example:

sage: \( A \ll a1, a2 \rr = \text{PolynomialRing}(\ZZ) \)

sage: \( B \ll b \rr = \text{PolynomialRing}(A) \)

sage: \( C \ll c \rr = \text{PowerSeriesRing}(B) \)

sage: \( R \ll x \rr = \text{PolynomialRing}(C) \)

sage: \( f = a2x^2 + c^x - a1^b \)

sage: \( f \text{.parent}() \)

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: \( f \text{.integral}() \)

\[ \frac{1}{3}a2x^3 + \frac{1}{2}c^x^2 - a1^b^x \]

sage: \( f \text{.integral}() \).\( \text{parent}() \)

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: \( g = 3a2x^2 + 2c^x - a1^b \)

sage: \( g \text{.integral}() \)

\[ a2x^3 + c^x^2 - a1^b^x \]

sage: \( g \text{.integral}() \).\( \text{parent}() \)

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:

sage: \( R \ll x \rr = \QQ[] \)

sage: \( t = \text{PolynomialRing}(R, \text{''t''}) \).\( \text{gen}() \)

sage: \( f = x^t + 5^t^2 \)

(continues on next page)
inverse_mod($a, m$)

Invert the polynomial $a$ with respect to $m$, or raise 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:

```
sage: S.<t> = QQ[]
sage: f = inverse_mod(t^2 + 1, t^3 + 1); f
-1/2*t^2 - 1/2*t + 1/2
sage: f * (t^2 + 1) % (t^3 + 1)
1
sage: f = t.inverse_mod((t + 1)^7); f
-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: # needs sage.modules
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
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
sage: g = 5*x^3 + x - 7; m = x^4 - 12*x + 13; f = inverse_mod(g, m); f
-0.0319636125...*x^3 - 0.0383269759...*x^2 - 0.0463050900...*x + 0.346479687...
sage: poly = f*g % m
sage: # Remove noisy zero terms:
sage: parent(poly)((0.0 if abs(c) <= epsilon else c # abs tol 1e-14
....: for c in poly.coefficients(sparse=False))
1.0000000000000004
```

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:

```python
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 \( \text{prec} \) 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:

```python
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:

```python
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:

```python
sage: R = Zmod(4)
sage: Rx.<x> = R[]
sage: Rxy.<y> = Rx[]
sage: p = 1 + (1+2*x)*y + x**2*y**4
```
Polynomials, Release 10.1

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: # needs sage.modules
sage: M = MatrixSpace(ZZ, 2)
sage: x = polygen(M)
sage: p = M([1,2,3,4])*x^3 + M([-1,0,0,1])*x^2 + M([1,3,-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:

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 \texttt{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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sage: (x - 1).is_cyclotomic()</code></td>
<td>True</td>
</tr>
<tr>
<td><code>sage: (x + 1).is_cyclotomic()</code></td>
<td>True</td>
</tr>
<tr>
<td><code>sage: (x^2 - 1).is_cyclotomic()</code></td>
<td>False</td>
</tr>
<tr>
<td><code>sage: (x^2 + x + 1).is_cyclotomic(certificate=True)</code></td>
<td>3</td>
</tr>
<tr>
<td><code>sage: (x^2 + 2^x + 1).is_cyclotomic(certificate=True)</code></td>
<td>0</td>
</tr>
</tbody>
</table>

Test first 100 cyclotomic polynomials:

<table>
<thead>
<tr>
<th>Code</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sage: all(cyclotomic_polynomial(i).is_cyclotomic() for i in range(1, 101))</code></td>
<td>True</td>
</tr>
</tbody>
</table>

Some more tests:

<table>
<thead>
<tr>
<th>Code</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sage: y = polygen(QQ)</code></td>
<td></td>
</tr>
<tr>
<td><code>sage: (y/2 - 1/2).is_cyclotomic()</code></td>
<td>False</td>
</tr>
<tr>
<td><code>sage: (2^y(y/2 - 1/2)).is_cyclotomic()</code></td>
<td>True</td>
</tr>
</tbody>
</table>

Invalid arguments:

<table>
<thead>
<tr>
<th>Code</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sage: (x - 3).is_cyclotomic(algorithm=&quot;sage&quot;, certificate=True)</code></td>
<td></td>
</tr>
</tbody>
</table>

Traceback (most recent call last):
...
ValueError: no implementation of the certificate within Sage
Test using other rings:

```python
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:**

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

**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.
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 \( \mathbb{Z}[x] \):

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

is_lorentzian(explain=False)  
Return True if this is a Lorentzian polynomial.
A univariate real polynomial is Lorentzian if and only if it is a monomial with positive coefficient, or zero. The definition is more involved for multivariate real polynomials.

INPUT:

• explain – boolean (default: False); if True return a tuple whose first element is the boolean result of the test, and the second element is a string describing the reason the test failed, or None if the test succeeded

EXAMPLES:

sage: P.<x> = QQ[]
sage: p1 = x^2
sage: p1.is_lorentzian()
True
sage: p2 = 1 + x^2
sage: p2.is_lorentzian()
False
sage: p3 = P.zero()
True
sage: p4 = -2*x^3
sage: p4.is_lorentzian()  # (False, 'inhomogeneous')
False

It is an error to check if a polynomial is Lorentzian if its base ring is not a subring of the real numbers, as the notion is not defined in this case:

sage: Q.<y> = CC[]
sage: q = y^2
sage: q.is_lorentzian()  # Traceback (most recent call last):
... NotImplementedError: is_lorentzian only implemented for real polynomials

The method can give a reason for a polynomial failing to be Lorentzian:

sage: p = x^2 + 2*x
sage: p.is_lorentzian(explain=True)
(False, 'inhomogeneous')

REFERENCES:

For full definitions and related discussion, see [BrHu2019] and [HMMS2019].

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
False
(continues on next page)
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():

```
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: 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
```

(continues on next page)
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:**

```python
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 is_primitive() 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.
Polynomials, Release 10.1

```python
sage: # needs sage.rings.finite_rings
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)[x]
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)[x]
sage: f = x^2 + x + 1
sage: f.is_primitive()
False
sage: f = x^2 - x + 2
sage: f.is_primitive()
True
sage: x = polygen(QQ); f = x^2 + 1
sage: f.is_primitive()
Traceback (most recent call last):
... NotImplementedError: isPrimitive() not defined for polynomials over infinite fields.
```

Ring semantics examples.

```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: # needs sage.rings.number_field
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))
```

(continues on next page)
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:

\begin{verbatim}
sage: # needs sage.libs.pari
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
\end{verbatim}

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:

\begin{verbatim}
sage: R.<x> = PolynomialRing(QQ)
sage: (x^2 + 2*x + 1).is_square()
True
sage: (x^4 + 2*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
\end{verbatim}
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 itself.

**Warning:** This method is not consistent with \texttt{squarefree\_decomposition()} since the latter does not factor the content of a polynomial. See the examples below.

**EXAMPLES:**

```sage
sage: R.<x> = QQ[]
sage: f = (x-1) * (x-2) * (x^2-5) * (x^17-3); f
x^21 - 3*x^20 + 15*x^18 - 10*x^17 + 9*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
sage: R.<x> = ZZ[]
sage: (2*x).is\_squarefree()      # needs sage.libs.pari
True
sage: (4*x).is\_squarefree()      # needs sage.libs.pari
False
sage: (2*x^2).is\_squarefree()    # needs sage.libs.pari
False
sage: R(0).is\_squarefree()      # needs sage.libs.pari
False
```

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

```sage
sage: K.<t> = FunctionField(GF(3))
sage: R.<x> = K[]
sage: (x^3 - x).is\_squarefree()       
```

(continues on next page)
In the following example, $t^2$ is a unit in the base field:

```
sage: R(t^2).is_squarefree()
True
```

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

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

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

```
sage: c = f.content()
sage: (f/c).is_squarefree()  # needs sage.libs.pari
True
```

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

```
sage: R.<x> = IntegerModRing(9)[]
sage: pol = (x + 3) * (x + 6); pol
x^2
sage: pol.is_squarefree()  # needs sage.libs.pari
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 \texttt{is_monomial()} instead:

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

\textbf{is_unit()}

Return \texttt{True} if this polynomial is a unit.

\textbf{EXAMPLES:}

sage: a = Integers(90384098234^3)
sage: b = a(2*191*236607587)
sage: b.is_nilpotent()
True

sage: # needs sage.libs.pari
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

\textbf{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.

\textbf{is_weil_polynomial} (\texttt{return_q=False})

Return \texttt{True} if this is a Weil polynomial.

This polynomial must have rational or integer coefficients.

\textbf{INPUT:}

\begin{itemize}
  \item \texttt{self} – polynomial with rational or integer coefficients
  \item \texttt{return\_q} – (default \texttt{False}) if \texttt{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.
\end{itemize}

\textbf{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
sage: P0.is_weil_polynomial(return_q=True) # needs sage.libs.pari
(True, 5)
sage: P0.is_weil_polynomial(return_q=False)
\texttt{# needs sage.libs.pari}
True
sage: P1.is_weil_polynomial(return_q=True)
(False, 0)
sage: P1.is_weil_polynomial(return_q=False)
False
sage: P2.is_weil_polynomial()
\texttt{# needs sage.libs.pari}
False

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.weil_polynomials.WeilPolynomials}.

AUTHORS:  
David Zureick-Brown (2017-10-01)

\texttt{is_zero()}
Test whether this polynomial is zero.

EXAMPLES:

\begin{verbatim}
    sage: R = GF(2)['x','y']
    sage: R([0,1]).is_zero()
    False
    sage: R([0]).is_zero()
    True
    sage: R([-1]).is_zero()
    False
\end{verbatim}

\texttt{lc()}
Return the leading coefficient of this polynomial.

OUTPUT: element of the base ring

This method is the same as \texttt{leading_coefficient()}.

EXAMPLES:

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

\texttt{lcm(other)}
Let $f$ and $g$ be two polynomials. Then this function returns the monic least common multiple of $f$ and $g$.

\texttt{leading_coefficient()}
Return the leading coefficient of this polynomial.

OUTPUT: element of the base ring

EXAMPLES:
sage: R.<x> = QQ[]
sage: f = (-2/5)*x^3 + 2*x - 1/3
sage: f.leading_coefficient()
-2/5

list(copy=True)

Return a new copy of the list of the underlying elements of self.

EXAMPLES:

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]

Note that v is a list, it is mutable, and each call to the list() method returns a new list:

sage: type(v)
<... 'list'>
sage: v[0] = 5
sage: f.list()
[-1/3, 2, 0, -2/5]

Here is an example with a generic polynomial ring:

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)
<class '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]

lm()

Return the leading monomial of this polynomial.

EXAMPLES:

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

local_height(v, prec=None)

Return the maximum of the local height of the coefficients of this polynomial.
INPUT:

- \( v \) – a prime or prime ideal of the base ring.
- \( \text{prec} \) – desired floating point precision (default: default `RealField` precision).

OUTPUT: a real number.

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(QQ)
sage: f = 1/1331*x^2 + 1/4000*x
sage: f.local_height(1331) # needs sage.rings.real_mpfr
7.19368581839511
```

```python
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: K.<k> = NumberField(x^2 - 5)
sage: T.<t> = K[]
sage: I = K.ideal(3)
sage: f = 1/3*t^2 + 3
sage: f.local_height(I)
1.09861228866811
```

```python
sage: R.<x> = QQ[]
sage: f = 1/2*x^2 + 2
sage: f.local_height(2, prec=2) # needs sage.rings.real_mpfr
0.75
```

\texttt{local height arch}(i, \text{prec=None})

Return the maximum of the local height at the \( i \)-th infinite place of the coefficients of this polynomial.

INPUT:

- \( i \) – an integer.
- \( \text{prec} \) – desired floating point precision (default: default `RealField` precision).

OUTPUT: a real number.

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(QQ)
sage: f = 210*x^2
sage: f.local_height_arch(0) # needs sage.rings.real_mpfr
5.34710753071747
```

```python
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: K.<k> = NumberField(x^2 - 5)
sage: T.<t> = K[]
sage: f = 1/2*t^2 + 3
sage: f.local_height_arch(1, prec=52)
1.09861228866811
```
Polynomials, Release 10.1

```python
sage: R.<x> = QQ[]
sage: f = 1/2*x^2 + 3
sage: f.local_height_arch(0, prec=2)  # needs sage.rings.real_mpfr
1.0
```

**lt()**

Return the leading term of this polynomial.

**EXAMPLES:**

```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> = ZZ[]
sage: f = x^2 + 2
sage: f.map_coefficients(lambda a: a + 42)
43*x^2 + 44
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
sage: # needs sage.symbolic
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> = 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
```
Examples with different base ring:

```
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
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)
```

\textbf{mod(\textit{other})}

Remainder of division of \textit{self} by \textit{other}.

EXAMPLES:

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

\textbf{monic()}

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

EXAMPLES:

```
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 \textit{f} = 0 cannot be made into a monic polynomial

```
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.
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 \( \text{self} \), where \( m \) must have the same parent as \( \text{self} \).

INPUT:
• \( m \) - a monomial

OUTPUT: Coefficient in base ring.

EXAMPLES:

sage: P.<x> = QQ[]

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

monomials()
Return the list of the monomials in \( \text{self} \) 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]

Polynomials, Release 10.1

# 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: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<rho> = NumberField(x**2 + 1)
sage: R.<y> = QQ[]
sage: p = rho * y
sage: p.monomials()
[y]

**multiplication_trunc**(other, n)

Truncated multiplication

EXAMPLES:

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

```python
sage: (x^10 + 5*x^5 + x^2 - 3).multiplication_trunc(x^7 - 3*x^3 + 1, 11)
```

```python
x^10 + x^9 - 15*x^8 - 3*x^7 + 2*x^5 + 9*x^3 + x^2 - 3
```

Check that coercion is working:

```python
sage: R2 = QQ['x']
sage: x2 = R2.gen()
sage: p1 = (x^3 + 1).multiplication_trunc(x2^3 - 2, 5); p1
```

```python
-x^3 - 2
```

```python
sage: p2 = (x2^3 + 1).multiplication_trunc(x^3 - 2, 5); p2
```

```python
-x^3 - 2
```

```python
sage: parent(p1) == parent(p2) == R2
```

True

**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.

**INPUT:**

- \( n \) - an integer (the number of iterations),
- \( x0 \) - an initial guess \( x_0 \).

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

If one of the iterates is a critical point of \( f \), a `ZeroDivisionError` exception is raised.

**EXAMPLES:**
Polynomials, Release 10.1

```python
sage: x = PolynomialRing(RealField(), 'x').gen()  # needs sage.rings.real_mpfr
sage: f = x^2 - 2  # needs sage.rings.real_mpfr
sage: f.newton_raphson(4, 1)  # needs sage.rings.real_mpfr
[1.50000000000000, 1.4166666666667, 1.41421568627451, 1.41421356237469]
```

**AUTHORS:**
- David Joyner and William Stein (2005-11-28)

### newton_slopes(p, lengths=False)

Return the $p$-adic slopes of the Newton polygon of `self`, when this makes sense.

**OUTPUT:**
If `lengths` is False, a list of rational numbers. If `lengths` is True, a list of couples $(s, l)$ where $s$ is the slope and $l$ the length of the corresponding segment in the Newton polygon.

**EXAMPLES:**

```python
sage: x = QQ['x'].0
sage: f = x^3 + 2
sage: f.newton_slopes(2)  # needs sage.libs.pari
[1/3, 1/3, 1/3]
```

```python
sage: R.<x> = PolynomialRing(ZZ, sparse=True)
```

```python
sage: p = x^5 + 6*x^2 + 4
sage: p.newton_slopes(2)  # needs sage.libs.pari
[1/2, 1/2, 1/3, 1/3, 1/3]
```

```python
sage: p.newton_slopes(2, lengths=True)  # needs sage.libs.pari
[(1/2, 2), (1/3, 3)]
```

```python
sage: (x^2^100 + 27).newton_slopes(3, lengths=True)  # needs sage.libs.pari
[(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:**

```python
sage: # needs sage.rings.real_mpfr
sage: R.<x> = RR[]
sage: f = x^6 + x^2 + -x^4 - 2*x^3
sage: f.norm(2)
2.64575131106459
```

```python
sage: (sqrt(1^2 + 1^2 + (-1)^2 + (-2)^2)).n()  # needs sage.rings.real_mpfr
1.41421356237310
```

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

→ needs sage.symbolic
2.64575131106459

sage: f.norm(1)  # needs sage.rings.real_mpfr
5.00000000000000
sage: f.norm(infinity)  # needs sage.rings.real_mpfr
2.00000000000000
sage: f.norm(-1)  # needs sage.rings.real_mpfr
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 \texttt{nth_root()} implemented.

EXAMPLES:

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: R(0).nth_root(3)
0
sage: R.<x> = 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: # needs sage.rings.number_field
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

(continues on next page)
Here we consider a base ring without \texttt{nth\_root} method. The third example with a non-trivial coefficient of lowest degree raises an error:

\begin{verbatim}
sage: R1.<x> = QQ[]
sage: R2.<y> = R1[]
sage: R3.<z> = R2[]
sage: (((y**2+x)*z^2 + x*y*z + 2*x)**3).nth_root(3)
(y^2 + x)*z^2 + x*y*z + 2*x
sage: ((x+y+z)**5).nth_root(5)
z + y + x
\end{verbatim}

(continues on next page)
\begin{verbatim}
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'
\end{verbatim}

**number_of_real_roots()**

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

**EXAMPLES:**

\begin{verbatim}
sage: # needs sage.libs.pari
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: # needs sage.libs.pari sage.rings.real_mpfr
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
\end{verbatim}

**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 \(-\text{Infinity}, \text{Infinity}\) (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). pari:polsturm requires a polynomial with real coefficients; in case PARI returns an error, we try again after taking the GCD of \texttt{self} with its complex conjugate.

**EXAMPLES:**

\begin{verbatim}
sage: # needs sage.libs.pari
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**)
\end{verbatim}
Polynomials, Release 10.1

3 sage: pol.number_of_roots_in_interval()
3 sage: pol = (x - 1) * (x - 2) * (x - 3)

sage: # needs sage.libs.pari sage.rings.real_mpfr
sage: pol2 = pol.change_ring(CC)

sage: pol2.number_of_roots_in_interval()
3
sage: R.<x> = PolynomialRing(CC)

sage: pol = (x - 1) * (x - CC(I))

sage: pol.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:

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)[x]
sage: S(f).number_of_terms()
5

sage: cyclotomic_polynomial(105).number_of_terms()
33

The method hamming_weight() is an alias:

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

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.
Next we compute the numerator 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.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.

```python
sage: R.<x> = RR[]
sage: f = x + RR('0.3'); f
x + 0.300000000000000
sage: f.numerator()
0
sage: f.denominator()
1
```

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

```python
sage: # needs sage.rings.number_field sage.symbolic
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 `valuation()`.

**EXAMPLES:**

```python
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`.

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, 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:

• xmin - float
• 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))
# needs sage.plot
Graphics object consisting of 1 graphics primitive
sage: x = polygen(QQ)
sage: plot(x^2 + 1, rgbcolor=(1,0,0))
# needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
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.

For univariate polynomials, if 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
• \( \text{prec} \) – (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); 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); 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 = {0}".format(i)
```

\( \text{prec}() \)

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
```

\( \text{pseudo_quo_rem}(\text{other}) \)

Compute the pseudo-division of two polynomials.

INPUT:

• \( \text{other} \) – a nonzero polynomial

OUTPUT:

2.1. Univariate Polynomials and Polynomial Rings
Q and R such that \( l^{m-n+1} \text{self} = Q \cdot \text{other} + R \) 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:**

```sage
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
```

\[\text{radical()}\]

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

**EXAMPLES:**

```sage
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
```

If \text{self} has a factor of multiplicity divisible by the characteristic (see github issue #8736):

```sage
sage: P.<x> = GF(2)[]
sage: (x^3 + x^2).radical()  # needs sage.rings.finite_rings
x^2 + x
```

\[\text{rational_reconstruct(*args, **kwds)}\]

Deprecated: Use \text{rational_reconstruction()} instead. See github issue #12696 for details.

\[\text{rational_reconstruction}(m, n_{\deg}=None, d_{\deg}=None)\]
Return a tuple of two polynomials \((n, d)\) where \text{self} * \text{d} is congruent to \text{n} modulo \text{m} and \text{n}.\degree() <= \text{n}.\deg and \text{d}.\degree() <= \text{d}.\deg.

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

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

EXAMPLES:
Over \( \mathbb{Q}[z] \):
\[
sage: z = PolynomialRing(QQ, 'z').gen()
sage: p = -z**16 - z**15 - z**14 + z**13 + z**12 + z**11 - z**5 - z**4 - z**3 + z**2 + z + 1
sage: m = z**21
sage: n, d = p.rational_reconstruction(m); n, d
(z**4 + 2*z**3 + 3*z**2 + 2*z + 1, z**10 + z**9 + z**8 + z**7 + z**6 + z**5 + z**4 + z**3 + z**2 + z + 1)
sage: ((p*d - n) % m).is_zero()
True
\]
Over \( \mathbb{Z}[z] \):
\[
sage: z = PolynomialRing(ZZ, 'z').gen()
sage: p = -z**16 - z**15 - z**14 + z**13 + z**12 + z**11 - z**5 - z**4 - z**3 + z**2 + z + 1
sage: m = z**21
sage: n, d = p.rational_reconstruction(m); n, d
(z**4 + 2*z**3 + 3*z**2 + 2*z + 1, z**10 + z**9 + z**8 + z**7 + z**6 + z**5 + z**4 + z**3 + z**2 + z + 1)
sage: ((p*d - n) % m).is_zero()
True
\]
Over an integral domain, \( d \) might not be monic:
\[
sage: P = PolynomialRing(ZZ, 'x')
sage: x = P.gen()
sage: p = 7*x**5 - 10*x**4 + 16*x**3 - 32*x**2 + 128*x + 256
sage: m = x**5
sage: n, d = p.rational_reconstruction(m, 3, 2); n, d
(-32*x**3 + 384*x**2 + 2304*x + 2048, 5*x + 8)
sage: ((p*d - n) % m).is_zero()
True
sage: n, d = p.rational_reconstruction(m, 4, 0); n, d
(-10*x**4 + 16*x**3 - 32*x**2 + 128*x + 256, 1)
sage: ((p*d - n) % m).is_zero()
True
\]
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).inverse_mod(z^9)
sage: m = z^9
sage: n, d = p.rational_reconstruction(m); n, d
(-1/t*z^4 - t*z - 1/t, z - 1/t)
sage: ((p*d - n) % m).is_zero()
True
sage: w = PowerSeriesRing(P.fraction_field(), 'w').gen()
sage: n = -10*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: x = n.subs(z=w)/d.subs(z=w) + O(w^prec)
sage: nc, dc = Pz(x.list()).rational_reconstruction(z^prec)
sage: (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) 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_reconstruction(m); n, d
(-z^4 - t^2*z - 1, t*z - 1)
sage: ((p*d - n) % m).is_zero()
True

Over $\mathbb{Q}_5$:

sage: # needs sage.rings.padics
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_reconstruction(m, 3, 2)
sage: ((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(z.exp().list())
sage: m = x^5
sage: n, d = p.rational_reconstruction(m, 4, 0); n, d
(1/24*x^4 + 1/6*x^3 + 1/2*x^2 + x + 1, 1)
sage: ((p*d - n) % m).is_zero()
True
sage: m = x^3
sage: n, d = p.rational_reconstruction(m, 1, 1); n, d
(-x - 2, x - 2)
sage: ((p*d - n) % m).is_zero()
True
sage: p = P(log(1-z).list())
    m = x^9
sage: n, d = p.rational_reconstruction(m, 4, 4); 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: ((p*d - n) % m).is_zero()
True
sage: p = P(sqrt(1+z).list())
    m = x^6
sage: n, d = p.rational_reconstruction(m, 3, 2); n, d
    (1/6*x^3 + 3*x^2 + 8*x + 16/3, x^2 + 16/3*x + 16/3)
sage: ((p*d - n) % m).is_zero()
True
sage: p = P((2*z).exp().list())
    m = x^7
sage: n, d = p.rational_reconstruction(m, 3, 3); n, d
    (-x^3 - 6*x^2 - 15*x - 15, x^3 - 6*x^2 + 15*x - 15)
sage: ((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((2*z).exp().list())
    m = x^7
sage: n, d = p.rational_reconstruction(m, 3, 3); 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_reconstruction`

```python
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:**

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

```python
reciprocal_transform(R=1, q=1)
```

Transform a general polynomial into a self-reciprocal polynomial.
The input \( Q \) and output \( P \) satisfy the relation

\[
P(x) = Q(x + q/x) x^{\text{deg}(Q)} R(x).
\]

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 \texttt{trace_polynomial()}.

INPUT:

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

EXAMPLES:

\begin{verbatim}
sage: R.<x> = PolynomialRing(Qrations())
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
\end{verbatim}

\texttt{resultant()} \( \textit{other} \)

Return the resultant of \texttt{self} and \texttt{other}.

INPUT:

- \( \texttt{other} \) – a polynomial

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

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

EXAMPLES:

\begin{verbatim}
sage: R.<x> = QQ[]
sage: f = x^3 + x + 1; g = x^3 - x - 1
sage: r = f.resultant(g); r
\end{verbatim}

We can compute resultants over univariate and multivariate polynomial rings:

\begin{verbatim}
sage: R.<a> = QQ[]
sage: S.<x> = R[]
sage: f = x^2 + a; g = x^3 + a
sage: r = f.resultant(g); r
\end{verbatim}
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[]
# needs sage.libs.flint
sage: (x + x^3/6 + x^5/120).revert_series(6)
([0.075000000000000 +/- ...e-17])*x^5 + ([0.166666666666667 +/- ...e-16])*x^3
# needs sage.libs.flint
sage: Pol.<x> = SR[]
sage: x.revert_series(6)
```

(continues on next page)
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.

EXAMPLES:

```sage
sage: R.<x> = QQ['x']
sage: f = x^3 + x + 17
sage: f.root_field('a')
    # needs sage.rings.number_field
    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')
    # needs sage.rings.number_field
    Rational Field

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

sage: # needs sage.rings.number_field
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: # needs sage.rings.finite_rings
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 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()  # needs sage.libs.pari
```

(continues on next page)
Traceback (most recent call last):
...
ValueError: polynomial must be irreducible

roots(ring=None, multiplicities=True, algorithm=None, **kwds)

Return the roots of this polynomial (by default, in the base ring of this polynomial).

INPUT:

• ring - the ring to find roots in
• multiplicities - bool (default: True) if True return list of pairs \((r, n)\), where \(r\) is the root and \(n\) is the multiplicity. If False, just return the unique roots, with no information about multiplicities.
• 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 \texttt{RealIntervalField} or \texttt{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 the \texttt{algorithm} keyword, for the cases where multiple algorithms exist.

EXAMPLES:

```
sage: # needs sage.libs.pari
sage: x = QQ['x'].0
sage: f = x^3 - 1
sage: f.roots()
[(1, 1)]
sage: f.roots(ring=CC)  # ... - 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)]
```

```
sage: # needs sage.rings.number_field
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:

```
sage: R.<x> = QQ[]
sage: (x^2 + 1).roots()  
˓→ needs sage.libs.pari
[]
sage: def my_roots(f, *args, **kwds):
    ....:     return f.change_ring(CDF).roots()
sage: QQ._roots_univariate_polynomial = my_roots
sage: (x^2 + 1).roots()  
˓→ needs numpy
[(2.775557615628914e-17 - 1.0*I, 1), (0.99999999999997*I, 1)]
sage: del QQ._roots_univariate_polynomial
```

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

```
sage: # needs numpy sage.rings.real_mpfr
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()  
˓→ needs numpy
[(1.25992104989487, 1), (-0.62996052494743... - 1.0911236597172*I, 1),
(-0.62996052494743... + 1.0911236597172*I, 1)]
sage: f.roots(algorithm='pari')
```
(continues on next page)
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()
needs sage.libs.pari
[(1, 1), (-1, 1)]
sage: f.roots(ring=QQ)
needs sage.libs.pari
[(3/2, 1), (1, 1), (-1, 1)]
```

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

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

An example involving large numbers:

```python
sage: # needs numpy sage.rings.real_mpfr
sage: x = RR['x'].0
sage: f = x^2 - 1e100
sage: f.roots()
([(-1.00000000000000e50, 1), (1.00000000000000e50, 1)],)
sage: f.roots(multiplicities=False)
[-1.6772670339941..., 0.19995479628..., 0.20004530611..., 1.5763035161844...]
```

```python
sage: # needs numpy sage.rings.real_mpfr
sage: x = CC['x'].0
sage: i = CC.0
sage: f = (x - 1) * (x - i)
sage: f.roots(multiplicities=False)
[1.00000000000000 - 1.00000000000000*I]
sage: g = (x - 1.33 + 1.33*i) * (x - 2.66 - 2.66*i)
sage: g.roots(multiplicities=False)
[1.33000000000000 - 1.33000000000000*I, 2.66000000000000 + 2.66000000000000*I]
```

Describing roots using radical expressions:

```python
sage: x = QQ['x'].0
sage: f = x^2 + 2
sage: f.roots(SR)
needs sage.symbolic
[(-I*sqrt(2), 1), (I*sqrt(2), 1)]
```

(continues on next page)
The roots of some polynomials cannot be described using radical expressions:

```
sage: (x^5 - x + 1).roots(SR)  # needs sage.symbolic
[]
```

For some other polynomials, no roots can be found at the moment due to the way roots are computed. [github issue #17516](https://github.com/sagemath/sage/issues/17516) addresses these defects. Until that gets implemented, one such example is the following:

```
sage: f = x^6 - 300*x^5 + 30361*x^4 - 1061610*x^3 + 1141893*x^2 - 915320*x + 101724
sage: f.roots()  # needs sage.libs.pari
[]
```

A purely symbolic roots example:

```
sage: X = var('X')
sage: f = 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 + 3)*X^3 - (3*I - 3)*sqrt(2)*X^2 - I*X^2 - (I + 3)*sqrt(2)*X + I*sqrt(2)
sage: f.roots()  # needs sage.libs.pari
[(I, 3), (-2^(1/4), 1), (2^(1/4), 1), (1, 1)]
```

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

```
sage: X = SR['X'].0
sage: f = (X - 1) * (X - I)^3 * (X^2 - sqrt(2)); f
X^6 + (-3*I - 1)*X^5 + (-sqrt(2) + 3*I - 3)*X^4 + ((3*I + 1)*sqrt(2) + I + 3)*X^3 + (-3*I - 3)*sqrt(2)*X^2 - I*X^2 - (-I + 3)*sqrt(2)*X + I*sqrt(2)
sage: f.roots()  # needs sage.libs.pari
[(I, 3), (-2^(1/4), 1), (2^(1/4), 1), (1, 1)]
```

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)
```
Polynomials, Release 10.1

(continued from previous page)

```
sage: p.roots(multiplicities=False) # needs sage.libs.pari
[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) # needs sage.libs.pari
[1, 0, 3, 6]
```

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

```
sage: # needs numpy sage.rings.complex_double
cdf = 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.
  ˓→5877852522924724*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.
  ˓→962854959127759e-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'].0
sage: ((x^3 - 1)).roots() # abs tol 4e-16 #
needs numpy
[[1.0000000000000002, 1]]
sage: ((x^3 - 1)).roots(multiplicities=False) # abs tol 4e-16 #
needs numpy
[1.0000000000000002]
```

More examples involving the complex double field:

```
sage: # needs numpy sage.rings.complex_double sage.rings.real_mpfr
cdf = CDF()
sage: i = CDF.0
sage: f = x^3 + 2*i; f
x^3 + 2.0*i
sage: f.roots() # abs tol 4e-16
[(-1.09112363597172... - 0.62996052494743...*I, 1),
  (1.25992104989487...*I, 1),
   (1.25992104989487...*I, 1)]
```

(continues on next page)
Polynomials, Release 10.1

Examples using real root isolation:

```python
sage: x = polygen(ZZ)
sage: f = x^2 - x - 1
sage: f.roots()
# needs sage.libs.pari
[]
sage: f.roots(ring=AA)
# needs sage.rings.number_field
[(-0.618033988749895?, 1), (1.618033988749895?, 1)]
sage: # needs sage.rings.real_interval_field
sage: f.roots(ring=RIF)
[(-0.6180339887498948482045868343657?, 1), (1.6180339887498948482045868343657?, 1)]
sage: f.roots(ring=RIF, multiplicities=False)
[(-0.6180339887498948482045868343657?, 1),
 (1.6180339887498948482045868343657?, 1)]
sage: f = f^2 * (x - 1)
sage: f.roots(ring=RIF)
[(-0.6180339887498948482045868343657?, 2),
 (1.0000000000000000000000000000000?, 1),
 (1.6180339887498948482045868343657?, 2)]
sage: f.roots(ring=RIF, multiplicities=False)
[(-0.6180339887498948482045868343657?,
 1.0000000000000000000000000000000?,
 1.6180339887498948482045868343657?]
```

Examples using complex root isolation:

```python
sage: x = polygen(ZZ)
sage: p = x^5 - x - 1
sage: p.roots()
# needs sage.libs.pari
[]
```

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

```python
sage: # needs sage.libs.flint
sage: Pol.<x> = CBF[]
```
```python
sage: (x^2 + 2).roots(multiplicities=False)
```
```python
[[+/- ...e-19] + [-1.414213562373095 +/- ...e-17]*I,
 [+/- ...e-19] + [1.414213562373095 +/- ...e-17]*I]
```
```python
sage: (x^3 - 1/2).roots(RBF, multiplicities=False)
```
```python
[[0.7937005259840997 +/- ...e-17]
```

In some cases, it is possible to isolate the roots of polynomials over complex ball fields:
UserWarning: roots may have been lost...
[[1.00000000000 +/- ...e-12] + [+/- ...e-11]*I,
 [1.00000000000 +/- ...e-12] + [+/- ...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.

```python
sage: # needs sage.rings.number_field
sage: K.<im> = QuadraticField(-1)
```

```python
data:
    y = polygen(K)
    p = y^4 - 2 - im
    p.roots(ring=CC)
```

```python
[(-1.2146389322441... - 0.14142505258239...*I, 1),
 (-0.14142505258239... + 1.2146389322441...*I, 1),
 (0.14142505258239... - 1.2146389322441...*I, 1),
 (1.2146389322441... + 0.14142505258239...*I, 1)]
```

```python
data:
    p = p^2 * (y^2 - 2)
    p.roots(ring=CIF)
```

```python
[(-1.4142135623709..., 1),
 (1.4142135623709..., 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:

```python
sage: # needs numpy sage.rings.real_mpfr sage.symbolic
sage: R.<x> = RealField(200)
```

```python
data:
    f = x^2 - R(pi)
    f.roots()
    f.roots(algorithm='numpy')
```

```python
[(-1.7724538509055160272981674833411451827975494561223871282138, 1),
 (1.7724538509055160272981674833411451827975494561223871282138, 1)]
```

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

We can also find roots over number fields:

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

```python
[(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.)

```python
sage: # needs sage.rings.complex_double sage.rings.real_mpfr
sage: rflds = (RR, RDF, RealField(100))
```

(continues on next page)
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'):
    ....:         rts = f.roots(ring=fld_out, multiplicities=False)
    ....:         if fld_in == fld_out and algo is None:
    ....:             print('{} {}'.format(fld_in, rts))
    ....:         for rt in rts:
    ....:             assert(abs(f2(rt)) <= 1e-10)
    ....:             assert(rt.parent() == fld_out)

Real Field with 53 bits of precision [1.25992104989487]
Real Double Field [1.25999210498948731647672106073]
Complex Field with 53 bits of precision [1.2599210498948731647672106073,...
    → 1.0911236359712*I, -0.62996052494743... + 1.0911236359712*I]
Complex Double Field [1.259921049898731647672106073,...
    → *I, -0.62996052494743... + 1.0911236359712*I]
Complex Field with 100 bits of precision [1.2599210498948731647672106073,...
    → 1.0911236359712*I, -0.62996052494743... + 1.0911236359712*I]

Note that we can find the roots of a polynomial with algebraic coefficients:

sage: # needs sage.rings.number_field
sage: rt2 = sqrt(AA(2))
sage: rt3 = sqrt(AA(3))
sage: x = polygen(AA)
sage: f = (x - rt2) * (x - rt3); f
x^2 - 3.146264369941973?*x + 2.449489742783178?
sage: rts = f.roots(); rts
[(1.41421356237095?, 1), (1.7320508075668878?, 1)]
sage: rts[0][0] == rt2
True
sage: f.roots(ring=RealIntervalField(150))
[(1.4142135623730950488016887242096980785696718753769480731766797387?, 1),
 (1.7320508075668877293527414505872366942805253810380628055806980?, 1)]

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)      #∞
∞
needs sage.rings.complex_double
+infinity
sage: CC(bigc)      #∞
∞
(continues on next page)
Polynomials using such large coefficients can’t be handled by numpy, but pari can deal with them:

```sage
sage: x = polygen(QQ)
sage: p = x + bigc
sage: p.roots(ring=RR, algorithm='numpy') # needs numpy
Traceback (most recent call last):
  ... LinAlgError: Array must not contain infs or NaNs
sage: p.roots(ring=RR, algorithm='pari') # needs sage.libs.pari
[(-3.50746621104340e451, 1)]
sage: p.roots(ring=AA) # needs sage.rings.number_field
[(-3.5074662110434039?e451, 1)]
sage: p.roots(ring=QQbar) # needs sage.rings.number_field
[(-3.5074662110434039?e451, 1)]
sage: p = bigc*x + 1
sage: p.roots(ring=RR) # needs numpy
[(-2.85106096489671e-452, 1)]
sage: p.roots(ring=AA) # needs sage.rings.number_field
[(-2.8510609648967059?e-452, 1)]
sage: p.roots(ring=QQbar) # needs sage.rings.number_field
[(-2.8510609648967059?e-452, 1)]
sage: p = x^2 - bigc
sage: p.roots(ring=RR) # needs numpy
[(-5.9223865215328558?e225, 1), (5.9223865215328558?e225, 1)]
sage: p.roots(ring=QQbar) # needs sage.rings.number_field
[(-5.9223865215328558?e225, 1), (5.9223865215328558?e225, 1)]
```

Check that github issue #30522 is fixed:

```sage
sage: PolynomialRing(SR, names="x")("x^2").roots() # needs sage.symbolic
[(0, 2)]
```

Check that github issue #30523 is fixed:

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

ALGORITHM:

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 $\text{roots()}$; otherwise, we use PARI's function $\text{pari:polroots}$. This choice can be overridden with $\text{algorithm='pari' or algorithm='numpy'}. If the algorithm is unspecified and NumPy's $\text{roots()}$ algorithm fails, then we fall back to PARI (NumPy will fail if some coefficient is infinite, for instance).

If $L$ is SR (or one of its subrings), then the roots will be radical expressions, computed as the solutions of a symbolic polynomial expression. At the moment this delegates to $\text{sage.symbolic.expression. Expression.solve()}$ which in turn uses Maxima to find radical solutions. Some solutions may be lost in this approach. Once github issue #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 $\text{sage.rings.polynomial. real_roots.real_roots()}$ is used. (You can call $\text{real_roots()}$ 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 $\text{sage.rings.polynomial.complex_roots.complex_roots()}$ is used. (You can call $\text{complex_roots()}$ directly to get more control than this method gives.)

If $L$ is AA and $K$ is QQbar or the Gaussian rationals, then $\text{complex_roots()}$ 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 $\text{change_ring()}$) (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 from $L$, we attempt to use $\text{change_ring(L)}$. 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 $ax + b$, 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 cube_root(j).
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: 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: 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: 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
```

(continues on next page)
splitting_field(names=None, map=False, **kwds)

Compute the absolute splitting field of a given polynomial.

INPUT:

- names – (default: None) a variable name for the splitting field.
- map – (default: False) also return an embedding of self into the resulting field.
- kwds – additional keywords depending on the type. Currently, only number fields are implemented. See `sage.rings.number_field.splitting_field.splitting_field()` for the documentation of these keywords.

OUTPUT:

If map is `False`, the splitting field as an absolute field. If map is `True`, a tuple `(K, phi)` where phi is an embedding of the base field of self in K.

EXAMPLES:

```
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
```

Relative situation:

```
sage: # needs sage.rings.number_field
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
```

With map=True, we also get the embedding of the base field into the splitting field:

```
sage: L.<b>, phi = (t^2 - a).splitting_field(map=True)
```

An example over a finite field:
sage: P.<x> = PolynomialRing(GF(7))
sage: t = x^2 + 1
sage: t.splitting_field('b')  # needs sage.rings.finite_rings
Finite Field in b of size 7^2

sage: P.<x> = PolynomialRing(GF(7^3, 'a'))  # needs sage.rings.finite_rings
sage: t = x^2 + 1
sage: t.splitting_field('b', map=True)  # needs sage.rings.finite_rings
(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)  # needs sage.rings.finite_rings
(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)  # needs sage.rings.finite_rings
(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

\texttt{squarefree\_decomposition()}

Return the square-free decomposition of this polynomial. This is a partial factorization into square-free, coprime polynomials.

EXAMPLES:

\begin{verbatim}
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) # needs sage.rings.number_field
sage: f.squarefree_decomposition() # needs sage.rings.number_field
1
\end{verbatim}

\texttt{subresultants(other)}

Return the nonzero subresultant polynomials of \texttt{self} and \texttt{other}.

INPUT:

\begin{itemize}
  \item \texttt{other} – a polynomial
\end{itemize}

OUTPUT: a list of polynomials in the same ring as \texttt{self}

EXAMPLES:

\begin{verbatim}
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^8 + 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]
\end{verbatim}

ALGORITHM:
We use the schoolbook algorithm with Lazard’s optimization described in [Duc1998]

REFERENCES:
Polynomials, Release 10.1

Wikipedia article Polynomial_greatest_common_divisor#Subresultants

**subs(**x, **kwds)**

Identical to `self(*x)`.

See the docstring for `__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
```

**substitute(**x, **kwds)**

Identical to `self(*x)`.

See the docstring for `__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:**
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)

If the polynomials share a non-constant common factor then the determinant of the Sylvester matrix will be zero:

sage: M.determinant()

If \(f\) and \(g\) 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)

The rank of the Sylvester matrix is related to the degree of the \(\gcd\) of \(f\) and \(g\):

sage: f.gcd(g).degree() == f.degree() + g.degree() - M.rank()
sage: h1.gcd(h2).degree() == h1.degree() + h2.degree() - M1.rank()

\textbf{symmetric\_power}(k, \text{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)]

(continues on next page)
\[\begin{align*}
x - 1, \\
x^5 - 2x + 2, \\
x^{10} + 2x^8 - 4x^6 - 8x^5 - 8x^4 + 16, \\
x^{10} + 4x^7 - 8x^6 + 16x^5 - 16x^4 + 32x^2 + 64, \\
x^5 + 2x^4 - 16, \\
x + 2
\end{align*}\]

\begin{vbcode}
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)]

\[\begin{align*}
x - 1, \\
(-x + d) * (-x + c) * (-x + b) * (-x + a), \\
(x - c*d) * (x - b*d) * (x - a*d) * (x - b*c) * (x - a*c) * (x - a*b), \\
(x - b*c*d) * (x - a*c*d) * (x - a*b*d) * (x - a*b*c), \\
x - a*b*c*d
\end{align*}\]

\textbf{trace\_polynomial()}\hfill

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:}\n
The inverse operation is \texttt{reciprocal\_transform()}.

\textbf{OUTPUT:}\n
- $Q$ – trace polynomial
- $R$ – cofactor
- $q$ – scaling factor

\textbf{EXAMPLES:}\n
\begin{vbcode}
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)
\end{vbcode}

We check that this function works for rings that have a coercion to the reals:

\begin{vbcode}
sage: # needs sage.rings.number_field
sage: K.<a> = NumberField(x^2 - 2, embedding=1.4)
sage: u = x^4 + a*x^3 + 3*a*x^2 + 2*a*x + 4
sage: u.trace_polynomial()
\end{vbcode}
\[
(x^2 + a^2x - 1, x^2 - 2, 2)
\]
sage: (u*(x^2-2)).trace_polynomial()
\[
(x^2 + a^2x - 1, x^2 - 2, 2)
\]
sage: (u*(x^2-2)^2).trace_polynomial()
\[
(x^4 + a^4x^3 - 9*a^2x^2 - 8*a^2x + 8, 1, 2)
\]
sage: (u*(x^2-2)^3).trace_polynomial()
\[
(x^4 + a^4x^3 - 9*a^2x^2 - 8*a^2x + 8, x^2 - 2, 2)
\]
sage: u = x^4 + a^4x^3 + 3*x^2 + 4*a^2x + 16
sage: u.trace_polynomial()
\[
(x^2 + a^2x - 5, 1, 4)
\]
sage: (u*(x-2)).trace_polynomial()
\[
(x^2 + a^2x - 5, x - 2, 4)
\]
sage: (u*(x+2)).trace_polynomial()
\[
(x^2 + a^2x - 5, x + 2, 4)
\]

**truncate** \((n)\)

Return the polynomial of degree \(< n\) 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 \(f = a_rx^r + a_{r+1}x^{r+1} + \cdots\), 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:

```
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:

```
sage: R.<x> = QQ[]
sage: x.variables()
(x,)
```

A constant polynomial has no variables.

```
sage: R(2).variables()
()```

**xgcd(other)**

Return an extended gcd of this polynomial and other.

**INPUT:**

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

**OUTPUT:**

A tuple \((r, s, t)\) where \(r\) is a greatest common divisor of this polynomial and \(other\), and \(s\) and \(t\) are such that \(r = s*\text{self} + t*\text{other}\) holds.

**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 `_xgcd_univariate_polynomial()`, then this method will be called (see examples below).

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQbar[]
sage: (2*x^2).gcd(2*x)
x
```

One can easily add xgcd functionality to new rings by providing a method `_xgcd_univariate_polynomial()`:

```
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
```

```python
class sage.rings.polynomial.polynomial_element.PolynomialBaseringInjection

Bases: Morphism

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 poly-
nomial 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 `github issue #9944`.

```
sage: # needs sage.rings.padics
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[]
sage: R.coerce_map_from(R.base_ring())
Polynomial base injection morphism:
  From: Rational Field
  To: Multivariate Polynomial Ring in x, y over Rational Field
sage: R.<x> = QQ[]
sage: R.coerce_map_from(R.base_ring())
```
```
Polynomial base injection morphism:
  From: Rational Field
  To:  Univariate Polynomial Ring in x over Rational Field

By [GitHub issue #9944](https://github.com/sagemath/sage/issues/9944), there are now only very few exceptions:

```python
sage: PolynomialRing(QQ, names=[]).coerce_map_from(QQ)
Call morphism:
  From: Rational Field
  To:  Multivariate Polynomial Ring in no variables over Rational Field

is_injective()
  Return whether this morphism is injective.

  EXAMPLES:

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

Check that [GitHub issue #23203](https://github.com/sagemath/sage/issues/23203) has been resolved:

```python
sage: R.is_subring(S)  # indirect doctest
True
```

is_surjective()
  Return whether this morphism is surjective.

  EXAMPLES:

  ```python
  sage: R.<x> = ZZ[]
sage: R.coerce_map_from(ZZ).is_surjective()
  False
  ```

section()

class `sage.rings.polynomial.polynomial_element.Polynomial_generic_dense`
  Bases: `Polynomial`

  A generic dense polynomial.

  EXAMPLES:

  ```python
  sage: f = QQ['x','y'].random_element()
sage: loads(f.dumps()) == f
  True
  ```

class `sage.rings.polynomial.polynomial_element.Polynomial_generic_dense`
  Bases: `Polynomial`

  A generic dense polynomial.

  EXAMPLES:

  ```python
  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:
Polynomials, Release 10.1

```python
sage: R.<t> = QQ[]
sage: S.<x> = R[]
sage: f = x*t + x + t
sage: f.constant_coefficient()
t
degree(gen=None)

EXAMPLES:

```python
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:

```python
sage: # needs sage.symbolic
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.

EXAMPLES:

```python
sage: P.<x> = QQ[]
sage: R.<y> = P[]
sage: f = y^10 + R.random_element(9)
sage: g = y^5 + R.random_element(4)
sage: q, r = f.quo_rem(g)
sage: f == q*g + r
(continues on next page)
```
Polynomials, Release 10.1

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

Polynomials over noncommutative rings are also allowed (after github issue #34733):

sage: # needs sage.combinat sage.modules
sage: HH = QuaternionAlgebra(QQ, -1, -1)
sage: P.<x> = HH[]
sage: f = P.random_element(5)
sage: g = P.random_element((0, 5))
sage: q, r = f.quo_rem(g)
sage: f == q*g + r
True

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:

sage: R.<x> = PolynomialRing(PolynomialRing(QQ,'y','x'))
sage: p = x^2 + 2*x + 4
sage: type(p)
<class 'sage.rings.polynomial.polynomial_element.Polynomial_generic_dense'>
sage: p.shift(0)
x^2 + 2*x + 4
sage: p.shift(-1)
x + 2
sage: p.shift(2)
x^4 + 2*x^3 + 4*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:

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

(continues on next page)
\[
q^{10} + 1
\]

```sage
sage: f.degree()
10
```

```sage
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: 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:

```sage
sage: # needs sage.rings.padics
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: # needs sage.rings.padics
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
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
```

2.1. Univariate Polynomials and Polynomial Rings
AUTHOR:

- Xavier Caruso (2013-03)

**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: # needs sage.rings.padics
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() # needs sage.rings.padics
0

sage: g.prec_degree() # needs sage.rings.padics
1
```

AUTHOR:

- Xavier Caruso (2013-03)

**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)

**sage.rings.polynomial.polynomial_element.is_Polynomial\(f\)**

Return True if \(f\) is of type univariate polynomial.

This function is deprecated.

**INPUT:**

- \(f\) - an object

**EXAMPLES:**

```sage
sage: from sage.rings.polynomial.polynomial_element import is_Polynomial
sage: R.<x> = ZZ[

sage: is_Polynomial(x^3 + x + 1)
```

(continues on next page)
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: # needs sage.symbolic
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.polynomial_is_variable(x)`

Test whether the given polynomial is a variable of its parent ring.

Implemented for instances of `Polynomial` and `MPolynomial`.

See also:

- `sage.rings.polynomial.polynomial_element.Polynomial.is_gen()`
- `sage.rings.polynomial.multi_polynomial.MPolynomial.is_generator()`

EXAMPLES:

```
sage: from sage.rings.polynomial.polynomial_element import polynomial_is_variable
sage: R.<x> = QQ[]
sage: polynomial_is_variable(x)
True
sage: polynomial_is_variable(R([0,1]))
True
sage: polynomial_is_variable(x^2)
False
sage: polynomial_is_variable(R(42))
False
```
sage: R.<y,z> = QQ[]
sage: polynomial_is_variable(y)
True
sage: polynomial_is_variable(z)
True
sage: polynomial_is_variable(y^2)
False
sage: polynomial_is_variable(y+z)
False
sage: polynomial_is_variable(R(42))
False
sage: polynomial_is_variable(42)
False

sage.rings.polynomial.polynomial_element.universal_discriminant()

Return the discriminant of the ‘universal’ univariate polynomial $a_n x^n + \cdots + a_1 x + a_0$ 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: # needs sage.libs.pari
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*a1^3*a3 + 18*a0*a1*a2*a3 - 27*a0^2*a3^2
sage: universal_discriminant(4).degrees()
(3, 4, 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)
class sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_cdv(parent, is_gen=False, construct=False)

Bases: Polynomial_generic_domain

A generic class for polynomials over complete discrete valuation domains and fields.

AUTHOR:
• Xavier Caruso (2013-03)

factor_of_slope(slope=None)

INPUT:
• slope – a rational number (default: the first slope in the Newton polygon of self)

OUTPUT:
The factor of self corresponding to the slope slope (i.e. the unique monic divisor of self whose slope is slope and degree is the length of slope in the Newton polygon).

EXAMPLES:

sage: # needs sage.rings.padics
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, -1/3, -1/3, -1/3, -1/3, -1/3, -1/3, -1/3]

sage: g = f.factor_of_slope(0)
# needs sage.rings.padics
sage: g.newton_slopes()
# needs sage.rings.padics
[0, 0, 0]

sage: (f % g).is_zero()
# needs sage.rings.padics
True

sage: h = f.factor_of_slope()
# needs sage.rings.padics
sage: h.newton_slopes()
# needs sage.rings.padics
[1]

sage: (f % h).is_zero()
# needs sage.rings.padics
True

If slope is not a slope of self, the corresponding factor is 1:

sage: f.factor_of_slope(-1)
# needs sage.rings.padics
1 + O(5^20)

AUTHOR:
**hensel_lift(a)**

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: # needs sage.rings.padics
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)
```

```python
sage: g = (x^2 + 1) * (x - 7); g
(x^2 + 1)*x - 7*(x^2 + 1)
```

**newton_polygon()**

Return 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: # needs sage.rings.padics
sage: K = Qp(5)
```

```python
sage: R.<t> = K[]
```

```python
sage: f = 5 + 3*t + t^4 + 25*t^10
```

```python
sage: f.newton_polygon()
Finite Newton polygon with 4 vertices: (0, 1), (1, 0), (4, 0), (10, 2)
```

```python
sage: g = f + K(0,0)*t^4; g
```

```python
Traceback (most recent call last):
... PrecisionError: The coefficient of \( t^4 \) has not enough precision
```

**AUTHOR:**

- Xavier Caruso (2013-03-20)
The `newton_slopes` function

The `newton_slopes` function takes an integer argument `repetition`, default value `True`, and returns a list of the Newton slopes of the 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 once.

**Examples:**

```python
sage: # needs sage.rings.padics
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]
```

**Author:**

- Xavier Caruso (2013-03-20)

The `slope_factorization` function

The `slope_factorization` function returns a factorization of the polynomial into a product of factors corresponding to each slope in the Newton polygon.

**Examples:**

```python
sage: # needs sage.rings.padics
sage: K = Qp(5)
sage: R.<x> = 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()
```

**Author:**

- Xavier Caruso (2013-03-20)
**Polynomials, Release 10.1**

```python
class sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_cdvf:
    (parent, is_gen=False, construct=False)
Bases: Polynomial_generic_cdv, Polynomial_generic_field

class sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_cdvr:
    (parent, is_gen=False, construct=False)
Bases: Polynomial_generic_cdv

class sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_dense_cdv:
    Bases: Polynomial_generic_dense_inexact, Polynomial_generic_cdv

class sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_dense_cdvf:
    Bases: Polynomial_generic_dense_cdv, Polynomial_generic_cdvf

class sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_dense_cdvr:
    Bases: Polynomial_generic_dense_cdv, Polynomial_generic_cdvr

class sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_dense_field:
    (parent, x=None, check=True, is_gen=False, construct=False)
Bases: Polynomial_generic_dense, Polynomial_generic_field

class sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_domain:
    (parent, is_gen=False, construct=False)
Bases: Polynomial, 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:**

```python
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: Polynomial_singular_repr, Polynomial_generic_domain, EuclideanDomainElement

quo_rem(other)
Return a tuple (quotient, remainder) where self = quotient * other + remainder.

EXAMPLES:

```python
sage: # needs sage.rings.number_field
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: 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:

```python
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:

```python
sage: # needs sage.libs.pari
sage: A.<T> = PolynomialRing(Integers(5), sparse=True)
sage: 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)
coefficients(sparse=True)
Return the coefficients of the monomials appearing in self.

EXAMPLES:

```python
sage: R.<w> = PolynomialRing(Integers(8), sparse=True)
sage: f = 5 + w^1997 - w^10000; f
7*w^10000 + w^1997 + 5
sage: f.coefficients()
[5, 1, 7]
```

degree(gen=None)
Return the degree of this sparse polynomial.

EXAMPLES:

```python
sage: R.<z> = PolynomialRing(ZZ, sparse=True)
sage: f = 13*z^50000 + 15*z^2 + 17*z
sage: f.degree()
50000
```

dict()
Return a new copy of the dict of the underlying elements of self.

EXAMPLES:

```python
sage: R.<w> = PolynomialRing(Integers(8), sparse=True)
sage: f = 5 + w^1997 - w^10000; f
7*w^10000 + w^1997 + 5
sage: d = f.dict(); d
{0: 5, 1997: 1, 10000: 7}
sage: d[0] = 10
sage: f.dict()
{0: 5, 1997: 1, 10000: 7}
```

exponents()
Return the exponents of the monomials appearing in self.

EXAMPLES:

```python
sage: R.<w> = PolynomialRing(Integers(8), sparse=True)
sage: f = 5 + w^1997 - w^10000; f
7*w^10000 + w^1997 + 5
sage: f.exponents()
[0, 1997, 10000]
```

gcd(other, algorithm=None)
Return the gcd of this polynomial and other

INPUT:

- 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 \( \mathbb{Z} \) 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
sage: p.number_of_terms()
3

\texttt{quo_rem}(\texttt{other})

Return the quotient and remainder of the Euclidean division of \texttt{self} and \texttt{other}.

Raises \texttt{ZeroDivisionError} if \texttt{other} is zero.

Raises \texttt{ArithmeticError} if \texttt{other} has a nonunit leading coefficient and this causes the Euclidean division to fail.

\texttt{EXAMPLES:}

sage: P.<x> = PolynomialRing(ZZ, sparse=True)
sage: R.<y> = PolynomialRing(P, sparse=True)
sage: f = R.random_element(10)
sage: while x.divides(f.leading_coefficient()):
    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
If the leading coefficient of \texttt{other} is not a unit, Euclidean division may still work:

sage: f = -x*y^10 + 2*x*y^7 + y^3 - 2*x^2*y^2 - y
sage: g = x*y^5
sage: f.quo_rem(g)
(-y^5 + 2*y^2, y^3 - 2*x^2*y^2 - y)

Polynomials over noncommutative rings are also allowed:

sage: # needs sage.combinat sage.modules
sage: HH = QuaternionAlgebra(QQ, -1, -1)
sage: P.<x> = PolynomialRing(HH, sparse=True)
sage: f = P.random_element(5)
sage: g = P.random_element((0, 5))
sage: q, r = f.quo_rem(g)
sage: f == q*g + r
True

\texttt{AUTHORS:}
reverse\[\text{degree=None}\]
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^100
sage: p.reverse()
x^1267650600228229401496703205372 + 2
sage: p.reverse(10)
x^6
```

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(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
```

AUTHOR: - David Harvey (2006-08-06)

truncate\(n\)
Return the polynomial of degree \(< n\) equal to self modulo \(x^n\).

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(ZZ, sparse=True)
sage: (x^11 + x^10 + 1).truncate(11)
x^10 + 1
sage: (x^2^500 + x^2^100 + 1).truncate(2^101)
x^1267650600228229401496703205376 + 1
```

valuation\(p=None\)
Return the valuation of self.

EXAMPLES:
```python
sage: # needs sage.rings.finite_rings
sage: R.<w> = PolynomialRing(GF(9, 'a'), sparse=True)
sage: f = w^1997 - w^10000
sage: f.valuation()
1997
sage: R(19).valuation()
0
sage: R(0).valuation()
+Infinity
```

```python
class sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_sparse_cdv(parent, x=None, check=True, is_gen=False, construct=False):
    Bases: Polynomial_generic_sparse, 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: Polynomial_generic_sparse_cdv, 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: Polynomial_generic_sparse_cdv, 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: Polynomial_generic_sparse, Polynomial_generic_field

EXAMPLES:
```
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:

```sage
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]
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:

```sage
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:

```sage
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: Polynomial_template
Univariate Polynomials over \( \mathbb{F}_2 \) via NTL’s GF2X.

EXAMPLES:

```sage
 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 \( F_2 \).

EXAMPLES:

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

Test that caching works:

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

```
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^21 + x^20 + x^16 + x^15 + x^14 + x^10 + x^9 +\x^8 + x^7 + x^6 + x^5 + x^2 + 1
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: 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 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
```
Polynomials, Release 10.1

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:
.. code-block::

    sage: f.truncate(10) is f
    True

If the precision is negative, the zero polynomial is returned:

.. code-block::

    sage: f.truncate(-1)
    0

\texttt{xgcd} (other)

Computes extended gcd of self and other.

\textbf{EXAMPLES:}

.. code-block::

    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{sage.rings.polynomial.polynomial\_gf2x.make\_element} (parent, args)

\section{2.1.6 Univariate polynomials over number fields}

\textbf{AUTHOR:}


\textbf{EXAMPLES:}

Define a polynomial over an absolute number field and perform basic operations with them:

.. code-block::

    sage: x = polygen(ZZ, 'x')
    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:

.. code-block::

    sage: # needs sage.rings.padics
    sage: x = polygen(ZZ, 'x')

(continues on next page)
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 + 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:

sage: # needs sage.symbolic
sage: N.<i, s2> = QQ[I, sqrt(2)]
sage: K.<x> = N[]
sage: f = x - s2
sage: f * (x + s2)
x^2 - 2
sage: f + g
x^3 - 2*i*x^2 + s2*x
sage: f % f
x^2 - 2
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: Polynomial_generic_dense_field

Class of dense univariate polynomials over an absolute number field.

gcd(other)

Compute the monic gcd of two univariate polynomials using PARI.

INPUT:

• other – a polynomial with the same parent as self.

OUTPUT: The monic gcd of self and other.

EXAMPLES:

sage: x = polygen(ZZ, 'x')
sage: N.<a> = NumberField(x^3 - 1/2, 'a')
sage: R.<r> = N['r']
sage: f = (5/4*a^2 - 2*a + 4)*r^2 + (5*a^2 - 81/5*a - 17/2)*r + 4/5*a^2 + 24*a
Polynomials, Release 10.1

(continued from previous page)

sage: g = (5/4*a^2 - 2*a + 4)*r^2 + (-11*a^2 + 79/5*a - 7/2)*r - 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

class sage.rings.polynomial.polynomial_number_field.Polynomial_relative_number_field_dense(parent, x=None, check=True, is_gen=False, construct=False)

Bases: 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:

sage: # needs sage.symbolic
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)

```python
class sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint
    Bases: Polynomial

A dense polynomial over the integers, implemented via FLINT.

_add_(right)

Return self plus right.

EXAMPLES:

```python
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:

```python
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)

Return self multiplied by right, where right is a scalar (integer).

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(ZZ)
sage: x^3
3*x
sage: (2*x^2 + 4)*x^3
6*x^2 + 12
```

_rmul_(right)

Return self multiplied by right, where right is a scalar (integer).

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(ZZ)
sage: 3*x
3*x
```
Polynomials, Release 10.1

(continued from previous page)

```
sage: 3*(2*x^2 + 4)
6*x^2 + 12

_mul_(right)

Return 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:

```
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:

...
\begin{verbatim}
sage: R.<x> = PolynomialRing(ZZ)
sage: x.degree()
1
sage: (x^2).degree()
2
sage: R(1).degree()
0
sage: R(0).degree()
-1
\end{verbatim}

\textbf{disc}(\textit{proof=\text{True}})

Return the discriminant of \texttt{self}, which is by definition
\[
(-1)^{m(m-1)/2} \frac{\text{resultant}(a, a')}{\text{lc}(a)},
\]
where \(m = \text{deg}(a)\), and \(\text{lc}(a)\) is the leading coefficient of \(a\). If \texttt{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()
-339
sage: f.discriminant(proof=False)
-339
\end{verbatim}

\textbf{discriminant}(\textit{proof=\text{True}})

Return the discriminant of \texttt{self}, which is by definition
\[
(-1)^{m(m-1)/2} \frac{\text{resultant}(a, a')}{\text{lc}(a)},
\]
where \(m = \text{deg}(a)\), and \(\text{lc}(a)\) is the leading coefficient of \(a\). If \texttt{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()
-339
sage: f.discriminant(proof=False)
-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
\end{verbatim}

(continues on next page)
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

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> = 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:
gcd(right)
Return the GCD of self and right. The leading coefficient need not be 1.

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: f.gcd(g)
sage: 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
sage: q5 = p.inverse_series_trunc(5)
sage: q5
-x^4 + 3*x^3 - x^2 - x + 1
sage: p*q5
-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)
sage: (q100 * p).truncate(100)
1
```

is_one()
Return True if self is equal to one.

EXAMPLES:

```python
sage: R.<x> = ZZ[]
sage: R(0).is_one()
False
sage: R(1).is_one()
True
sage: x.is_one()
False
```

is_zero()
Return True if self is equal to zero.

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

sage: 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.

**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)
```

(continues on next page)
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:

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)
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()
Return 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)
Return 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').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 \( f(\text{self}(x)) = \text{self}(f(x)) = x \mod x^n \).

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 \( \text{self} \). This is a partial factorization of \( \text{self} \) 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.sqfree_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.sqfree_decomposition()
(37) * (x - 4) * (x^2 - 3*x + 2)^2 * (x - 3)^3

xgcd(right)

Return a triple \((g, s, t)\) such that \(g = s \cdot \text{self} + t \cdot \text{right}\) and such that \(g\) is the gcd of \(\text{self}\) and \(\text{right}\) up to a divisor of the resultant of \(\text{self}\) and \(\text{other}\).

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 \(\text{self}\) and \(\text{right}\) are coprime as polynomials over the rationals, then \(g\) is guaranteed to be the resultant of \(\text{self}\) and \(\text{right}\), 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)
1

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, -432*x + 8208, 432*x^2 + 864*x + 14256)
sage: u*F + v*G
2985984

sage: (x, 0, 1)
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 `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:

```python
sage: K.<x> = PolynomialRing(ZZ, implementation='NTL')
sage: K
Univariate Polynomial Ring in x over Integer Ring (using NTL)
```

```python
class sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl
    Bases: Polynomial

    A dense polynomial over the integers, implemented via NTL.

    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, 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:

    ```python
    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

\[ (-1)^{m(m-1)/2} \text{resultant}(a, a')/lc(a), \]

where \( m = \text{deg}(a) \), and \( lc(a) \) 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:**

```
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()
(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
```

**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
```

(continues on next page)
\begin{verbatim}
sage: f = 2*x*(x-2)*(x-9)
sage: f.factor_mod(7)
(2) * x * (x + 5)^2
\end{verbatim}

\textbf{factor_padic}(p, \texttt{prec}=10)

Return \( p \)-adic factorization of \texttt{self} to given precision.

\textbf{INPUT:}

\begin{itemize}
  \item \( p \) – prime
  \item \( \texttt{prec} \) – integer; the precision
\end{itemize}

\textbf{OUTPUT:}

\begin{itemize}
  \item factorization of \texttt{self} over the completion at \( p \).
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
R.<x> = PolynomialRing(ZZ, \texttt{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 + 2*5^2 + 3*5^3 + O(5^4))
\end{verbatim}

A more difficult example:

\begin{verbatim}
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))*x + 1 + O(5^10))^2
\end{verbatim}

\textbf{gcd}(\texttt{right})

Return the GCD of \texttt{self} and \texttt{right}. The leading coefficient need not be 1.

\textbf{EXAMPLES:}

\begin{verbatim}
R.<x> = PolynomialRing(ZZ, \texttt{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
\end{verbatim}

\textbf{lcm}(\texttt{right})

Return the LCM of \texttt{self} and \texttt{right}.

\textbf{EXAMPLES:}

\begin{verbatim}
R.<x> = PolynomialRing(ZZ, \texttt{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)
6*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
\end{verbatim}
list($copy=True$)

Return a new copy of the list of the underlying elements of self.

EXAMPLES:

```sage
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($right$)

Attempt to divide self by right, and return a quotient and remainder.

If right is monic, then it returns \((q, r)\) where \(self = q \cdot right + r\) and \(\deg(r) < \deg(right)\).

If right is not monic, then it returns \((q, 0)\) where \(q = self/right\) if right exactly divides self, otherwise it raises an exception.

EXAMPLES:

```sage
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*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: 0//(2*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) * (2*x - 1)
sage: f.quo_rem(2*x - 1)
(x^2 + 3, 0)
sage: f = x^2
sage: f.quo_rem(2*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 $g = s*\text{self} + s*\text{right}$. 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))
(continues on next page)
Polynomials, Release 10.1

(continued from previous page)

(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: 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.

__add__(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
# indirect doctest
-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

(continues on next page)
Polynomials, Release 10.1

(continued from previous page)

```
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
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
```

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:
denominator()

Return the denominator of self.

EXAMPLES:

```python
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 $R_n(f) := (-1)^{(n(n-1)/2)} R(f, f') a_n^{(n-k-2)}$, 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 `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
```
**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 $R_n(f) := (-1)^{\frac{n(n-1)}{2}} R(f, f') a_n^{(n-k-2)}$, 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 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:

```
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 github issue #20631):

```
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 `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 + O(2^10)*x + 2 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 + 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 + O(3^10)*x + 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)
sage: f.factor_padic(5)
((1 + O(5^10))*x^2
 + 2 + 4*5 + 2*5^2 + 2*5^3 + 5^4 + 3*5^5 + 4*5^7 + 2*5^8 + 5^9 + O(5^10))
* ((1 + O(5^10))*x^2
 + (3 + 2*5^2 + 2*5^3 + 3*5^4 + 5^5 + 4*5^6 + 2*5^8 + 3*5^9 + O(5^10))*x
 + 4 + 5 + 2*5^2 + 4*5^3 + 4*5^4 + 3*5^5 + 3*5^6 + 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 github issue #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()  # precision error
Traceback (most recent call last):
...
PrecisionError: p-adic factorization not well-defined since the discriminant is zero up to the requestion p-adic precision
```

A more difficult example:
sage: \textbackslash f \textasciitilde 100 \ast (5\ast x + 1)^2 \ast (x + 5)^2 \\
sage: f.factor_padic(5, 10) \\
(4\ast 5^4 + 0(5^14)) \ast ((1 + 0(5^9))\ast x + 5^{-1} + 0(5^9))^2 \\
\ast ((1 + 0(5^10))\ast x + 5 + 0(5^10))^2

Try some bogus inputs:

sage: \textbackslash f.factor_padic(3, -1) \\
Traceback (most recent call last): \\
... \\
ValueError: prec_cap must be non-negative \\
sage: \textbackslash f.factor_padic(6, 10) \\
Traceback (most recent call last): \\
... \\
ValueError: p must be prime \\
sage: \textbackslash f.factor_padic('hello', 'world') \\
Traceback (most recent call last): \\
... \\
TypeError: unable to convert 'hello' to an integer

\textit{galois\_group}(pari\_group=False, algorithm='\textit{pari}')

Return the Galois group of this polynomial as a permutation group.

INPUT:

- \textit{self} - Irreducible polynomial
- \textit{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 \textit{P}, type \texttt{PermutationGroup(P)}.
- \textit{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.

\textbf{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 uses the “Transitive Groups Libraries” from the “TransGrp” GAP package which comes installed with the “gap” Sage package.

MAGMA does not return a provably correct result. Please see the MAGMA documentation for how to obtain a provably correct result.

EXAMPLES:

sage: R.\langle x \rangle = \text{QQ}[x] \\
sage: f = x^4 \minus{} 17\ast x^3 \minus{} 2\ast x + 1 \\
sage: G = f.galois\_group(); G \\
Transitive group number 5 of degree 4

(continues on next page)
Polynomials, Release 10.1

\begin{verbatim}
(continued from previous page)
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.)

\begin{verbatim}
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
\end{verbatim}

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.)

\begin{verbatim}
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')
Transitive group number 5 of degree 4
sage: f = x^13 - 17*x^3 - 2*x + 1
sage: f.galois_group(algorithm='gap')
Transitive group number 9 of degree 13
sage: f = x^12 - 2*x^8 - x^7 + 2*x^6 + 4*x^4 - 2*x^3 - x^2 - x + 1
sage: f.galois_group(algorithm='gap')
Transitive group number 183 of degree 12
sage: f.galois_group(algorithm='magma')  # optional - magma
Transitive group number 5 of degree 4
\end{verbatim}

\texttt{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 assume\_irreducible=True; this yields undefined results if $f$ is not irreducible.

A corresponding function in Magma is \texttt{IsEasySnAn}.

EXAMPLES:

\begin{verbatim}
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 + 3*x^2 - 1
sage: u.galois_group_davenport_smith_test()
2
\end{verbatim}
\end{verbatim}

(continues on next page)
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:

```
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.gcd(g)
1
sage: f = (-3*t + 1/2) * f
sage: g = (-3*t + 1/2) * (4*t^2/3 - 1) * g
sage: f.gcd(g)
t - 1/6
```

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 \( \text{prec} \) 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
```
$$\frac{151}{800}x^4 - \frac{17}{80}x^3 + \frac{11}{40}x^2 - \frac{1}{4}x + \frac{1}{2}$$

```python
sage: q5 * p
-\frac{453}{4000}x^6 + \frac{253}{800}x^5 + 1
```

```python
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:**

```python
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:**

```python
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:**

```python
sage: R.<t> = QQ[]
sage: f = 1 - t + \frac{1}{2}t^2 - \frac{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 \( \text{lcm}(a, b) \cdot \text{gcd}(a, b) = a \cdot b \) holds up to multiplication by rationals.

**EXAMPLES:**

```python
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
```

**list**(copy=True)

Return a list with the coefficients of *self*.

**EXAMPLES:**

```python
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*.

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:**

```python
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:**

```python
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:

```python
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:

```python
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=None)
Reverse the coefficients of this polynomial (thought of as a polynomial of degree degree).

INPUT:

- degree (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:
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:

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:

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(n)

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

EXAMPLES:

sage: R.<t> = QQ[]
sage: f = t - t^3/6 + t^5/120
sage: f.revert_series(6)
3/40*t^5 + 1/6*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^3/3 + t^5/5
sage: g.revert_series(6)
Traceback (most recent call last):
  ... ValueError: self must have constant coefficient 0 and a unit for coefficient t^1

truncate(n)

Return self truncated modulo \( t^n \).

INPUT:

• \( n \) - The power of \( t \) modulo which self is truncated

EXAMPLES:

sage: R.<t> = QQ[]
sage: f = 1 - t + 1/2*t^2 - 1/3*t^3
sage: f.truncate(0)
0

(continues on next page)
sage: f.truncate(2)
-t + 1

xgcd(right)

Return polynomials \(d, s,\) and \(t\) such that \(d == s * \text{self} + t * \text{right},\) 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:

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)

2.1.10 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 \(\text{sys.maxsize}\). 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:

sage: R.<a> = PolynomialRing(Integers(100))
sage: type(a)
<class 'sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint'>
sage: R.<a> = PolynomialRing(Integers(5*2^64))
sage: type(a)
<class '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 92233720368547758080

AUTHORS:

- Burcin Erocal (2008-11) initial implementation
- Martin Albrecht (2009-01) another initial implementation

class sage.rings.polynomial.polynomial_zmod_flint.Polynomial_template

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 \(\text{celement}\) functions (see \(\text{sage.libs.ntl.ntl_GF2X_linkage}\) for an example). Both parts are then plugged together by inclusion of the linkage file when inheriting from this class. See \(\text{sage.rings.polynomial.polynomial_gf2x}\) 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:

```plaintext
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:

```plaintext
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:

```plaintext
sage: P.<x> = GF(2)[]
sage: x.is_gen()
True
sage: (x+1).is_gen()
False
```

**is_one()**

EXAMPLES:

```plaintext
sage: P.<x> = GF(2)[]
sage: P(1).is_one()
True
```

**is_zero()**

EXAMPLES:

```plaintext
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
```

**xgcd** *(other)*

Computes extended gcd of self and other.

**EXAMPLES:**

```python
```
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)

class sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint
Bases: Polynomial_template
Polynomial on \(Z/nZ\) implemented via FLINT.

_add_(right)
EXAMPLES:

_sage: P.<x> = GF(2)[]
_sage: x + 1
x + 1

-sub_(right)
EXAMPLES:

-sage: P.<x> = GF(2)[]
-sage: x - 1
x + 1

-lmul_(left)
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

-sage: P.<x> = GF(2)[]
-sage: t = x^2 + x + 1
-sage: t*0
0
-sage: t*1
x^2 + x + 1

(continues on next page)
sage: R.<y> = GF(5)[]
sage: u = y^2 + y + 1
sage: u^3
3*y^2 + 3*y + 3
sage: u^5
0

_\texttt{rmul}_ (right)

Multiply self on the right by a scalar.

EXAMPLES:

sage: R.<x> = ZZ[]
sage: f = (x^3 + x + 5)
sage: f._\texttt{rmul}_ (7)
7*x^3 + 7*x + 35
sage: f*7
7*x^3 + 7*x + 35

_\texttt{mul}_ (right)

EXAMPLES:

sage: P.<a> = GF(7)[]
sage: a = P(range(10)); b = P(range(5, 15))
sage: a._\texttt{mul_trunc}_ (b, 5)
4*a^4 + 6*a^3 + 2*a^2 + 5*a

factor()

Return the factorization of the polynomial.

EXAMPLES:

sage: R.<x> = GF(5)[]
sage: (x^2 + 1).factor()
(x + 2) * (x + 3)

It also works for prime-power moduli:

sage: R.<x> = Zmod(23^5)[]
sage: (x^3 + 1).factor()
(x + 1) * (x^2 + 6436342*x + 1)
**is_irreducible()**

Return whether this polynomial is irreducible.

**EXAMPLES:**

```python
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:

```python
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
```

**minpoly_mod(other)**

Thin wrapper for `sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_mod_n.minpoly_mod()`.  

**EXAMPLES:**

```python
sage: R.<x> = GF(127)[]
sage: type(x)
<class 'sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint'>
sage: (x^5 - 3).minpoly_mod(x^3 + 5*x - 1)
x^3 + 34*x^2 + 125*x + 95
```

**monic()**

Return this polynomial divided by its leading coefficient.

 Raises `ValueError` if the leading coefficient is not invertible in the base ring.

**EXAMPLES:**

```python
sage: R.<x> = GF(5)[]
sage: (2*x^2 + 1).monic()
x^2 + 3
```

**rational_reconstruct(**args, **kwds)**

Deprecated: Use `rational_reconstruction()` instead.  See github issue #12696 for details.

**rational_reconstruction(m, n_deg=0, d_deg=0)**

Construct a rational function \( \frac{n}{d} \) such that \( p \cdot d \) is equivalent to \( n \mod m \) where \( p \) is this polynomial.

**EXAMPLES:**

```python
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_reconstruction(x^9, 4, 4);
```

2.1. Univariate Polynomials and Polynomial Rings 173
**resultant**(other)

Return 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)['x']
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
```

The following example shows that github issue #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 the optional argument degree 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.

```python
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 \( f(self(x)) = self(f(x)) = x \) (mod \( x^n \)).

**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...
```

**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)
sage: f = x^3 + 10*x^2 + 5000*x - 222
sage: f.small_roots()
[4]
```

**squarefree_decomposition**

Return the squarefree decomposition of this polynomial.

**EXAMPLES:**

```python
```
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: `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
```

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(copy=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]
```

**minpoly_mod(other)**

Compute the minimal polynomial of this polynomial modulo another polynomial in the same ring.

**ALGORITHM:**

NTL's MinPolyMod(), which uses Shoup's algorithm [Sho1999].

**EXAMPLES:**

```
sage: R.<x> = PolynomialRing(GF(101), implementation='NTL')
sage: f = x^17 + x^2 - 1
sage: (x^2).minpoly_mod(f)
x^17 + 100*x^2 + 2*x + 100
```

**ntl_ZZ_pX()**

Return underlying NTL representation of this polynomial. Additional “bonus” functionality is available through this function.

**Warning:** You must call ntl.set_modulus(ntl.ZZ(n)) before doing arithmetic with this object!

**ntl_set_directly(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 ZZ_pX class. Use this function to set the value of this polynomial using the NTL constructor, which is potentially very fast. The input v is either a vector of ints or a string of the form [ n1 n2 n3 ... ] 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(100), implementation='NTL')
sage: from sage.rings.polynomial.polynomial_modn_dense_ntl import Polynomial_˓→dense_mod_n as poly_modn_dense
sage: poly_modn_dense(R, ([1,-2,3]))
3*x^2 + 98*x + 1
```
Polynomials, Release 10.1

(continued from previous page)

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

quo_rem(right)

Return a tuple (quotient, remainder) where self = quotient*other + remainder.

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:

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)
x + 2
sage: p.shift(-5)
0
sage: p.shift(2)
x^4 + 2*x^3 + 4*x^2

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:

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: `Polynomial_dense_mod_n`

A dense polynomial over the integers modulo \( p \), where \( p \) is prime.

discriminant()

EXAMPLES:
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)

Return 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)
(1, 0, 1)
sage: (x^2 - 1).xgcd(x - 1)
(1, 0, 1)
sage: R.zero().xgcd(R.one())
(1, 0, 1)
sage: (x^3 - 1).xgcd((x - 1)^2)
```

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

```
(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: Polynomial_dense_mod_n
degree()

EXAMPLES:
```
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()

degree()

EXAMPLES:
```
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:
```
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()
```
(continues on next page)
\[
x^2 + 1
\]
\[
sage: f.reverse(1)
1
\]
\[
sage: f.reverse(5)
\]
\[
x^4 + x^2
\]

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

Shift self to left by \(n\), which is multiplication by \(x^n\), truncating if \(n\) is negative.

EXAMPLES:

\[
sage: R.<x> = PolynomialRing(Integers(12^{30}), implementation='NTL')
sage: f = x^7 + x + 1
sage: f.shift(1)
x^8 + x^2 + x
sage: f.shift(-1)
x^6 + 1
sage: f.shift(10).shift(-10) == f
True
\]

**truncate** \((n)\)

Return this polynomial mod \(x^n\).

EXAMPLES:

\[
sage: R.<x> = PolynomialRing(Integers(15^{30}), implementation='NTL')
sage: f = sum(x^n for n in range(10)); f
x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1
sage: f.truncate(6)
x^5 + x^4 + x^3 + x^2 + x + 1
\]

**valuation**()

Return 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: \(Polynomial\_dense\_mod\_n\)

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()

Return 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)

Return $q$ and $r$, with the degree of $r$ less than the degree of $right$, such that $q \cdot right + r = self$.

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
```
**truncation**

Return this polynomial mod $x^n$.

**EXAMPLES:**

```python
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**

Return the valuation of `self`, that is, the power of the lowest non-zero monomial of `self`.

**EXAMPLES:**

```python
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
```

`sage.rings.polynomial.polynomial_modn_dense_ntl.make_element(parent, args)`

`sage.rings.polynomial.polynomial_modn_dense_ntl.small_roots(self, X=None, beta=1.0, epsilon=None, **kwds)`

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 $b > N^\beta$. 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 $X = \text{cei}(1/2N^{\beta^2/\beta-\epsilon})$. 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)
- $\beta$ – compute a root mod $b$ where $b$ is a factor of $N$ and $b \geq N^{2\beta}$. (Default: 1.0, so $b = N$.)
- $\epsilon$ – the parameter $\epsilon$ described above. (Default: $\beta/8$)
- **kwds** – passed through to method `Matrix_integer_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 $\mathbb{Z}$):
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:

```python
sage: p = 2^256 + 2^8 + 2^5 + 2^3 + 1
sage: q = 2^256 + 2^8 + 2^5 + 2^3 + 2^2 + 1
sage: N = p*q
sage: ZmodN = Zmod( N )
```

We choose a random key:

```python
sage: K = ZZ.random_element(0, 2^Kbits)
sage: Kdigits = K.digits(2)
```

and pad it with $512 - 56 = 456$ 1s:

```python
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:

```python
sage: C = ZmodN(M)^e
```

To recover $K$ we consider the following polynomial modulo $N$:

```python
sage: P.<x> = PolynomialRing(ZmodN, implementation='NTL')
sage: f = (2^Nbits - 2^Kbits + x)^e - C
```
and recover its small roots:

```python
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 $N$:

```python
sage: length = 512
sage: hidden = 110
sage: p = next_prime(2^int(round(length/2)))
sage: q = next_prime(round(pi.n()*p))
# needs sage.symbolic
sage: N = p*q
# needs sage.symbolic
```

Now we disturb the low 110 bits of $q$:

```python
sage: qbar = q + ZZ.random_element(0, 2^hidden - 1)
# needs sage.symbolic
```

And try to recover $q$ from it:

```python
sage: F.<x> = PolynomialRing(Zmod(N), implementation='NTL')
# needs sage.symbolic
sage: f = x - qbar
# needs sage.symbolic
```

We know that the error is $\leq 2^{\text{hidden}} - 1$ and that the modulus we are looking for is $\geq \sqrt{N}$:

```python
sage: from sage.misc.verbose import set_verbose
sage: set_verbose(2)
sage: d = f.small_roots(X=2^hidden-1, beta=0.5)[0] # time random #
# needs sage.symbolic
```

REFERENCES:


2.1.12 Dense univariate polynomials over \( \mathbb{R} \), implemented using MPFR

```python
class sage.rings.polynomial.polynomial_real_mpfr_dense.PolynomialRealDense
    Bases: Polynomial

    change_ring(R)
        EXAMPLES:

        sage: from sage.rings.polynomial.polynomial_real_mpfr_dense import PolynomialRealDense
        sage: f = PolynomialRealDense(RR['x'], [-2, 0, 1.5])
        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:

        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:

        sage: from sage.rings.polynomial.polynomial_real_mpfr_dense import PolynomialRealDense
        sage: f = PolynomialRealDense(RR['x'], [3, pi, 1])
        # needs sage.symbolic
        sage: f.integral()
        # needs sage.symbolic
        0.33333333333333*x^3 + 1.57079632679490*x^2 + 3.00000000000000*x

    list(copy=True)
        EXAMPLES:

        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:
```
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: # needs sage.symbolic
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: # needs sage.symbolic
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
**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** \( (\text{bound}) \)

Truncate all high order coefficients below \( \text{bound} \).

**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 + 0.000010*x^4 + 0.010*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 + 0.00010*x^3 + 0.010*x^2 + 0.10*x + 1.0
sage: f.truncate_abs(10.0)
0
sage: f.truncate_abs(1e-100) == f
True
```

`sage.rings.polynomial.polynomial_real_mpfr_dense.make_PolynomialRealDense` \( (\text{parent, data}) \)

**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)

```python
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.
```

```python
sage.rings.polynomial.polynomial_singular_interface.can_convert_to_singular(R):
    Return True if this ring's base field or ring can be represented in Singular, and the polynomial ring has at least one generator.
    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
    sage: can_convert_to_singular(PolynomialRing(QQ, names=[]))
    False
```

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.
Bases: *Polynomial*

**content()**

Compute the content of this polynomial.

**OUTPUT:**

If this is the zero polynomial, return the constant coefficient. Otherwise, since the content is only defined up to a unit, return the content as $\pi^k$ with maximal precision where $k$ is the minimal valuation of any of the coefficients.

**EXAMPLES:**

```
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
O(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 + O(2^10))*x + 2 + O(2^11)
sage: fp.content()
1 + O(2^10)
sage: (2*fp).content()
2 + O(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 = 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 + O(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:

```
sage: factor(t^2)
Traceback (most recent call last):
...
PrecisionError: p-adic factorization not well-defined since
the discriminant is zero up to the requestion p-adic precision
```

An example of factoring a constant polynomial (see github issue #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 github issue #4038:

sage: E = EllipticCurve('37a1')
sage: K = Qp(7, 10)
sage: EK = E.base_extend(K)
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 + O(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 + O(7^10)) * ((1 + O(7^10))*x^3 + (6 + 4*7 + 2*7^2 +
   4*7^3 + 3*7^4 + 7*6 + 3*7^7 + 7*8 + 3*7^9 + O(7^10))*x^2 + (6 + 3*7 +
   5*7^2 + 2*7^4 + 7*5 + 7^6 + 2*7^8 + 3*7^9 + O(7^10))*x + 2 + 5*7 + 4*7^2 +
   6*7^4 + 3*7^5 + 7^6 + 4*7^7 + O(7^10))

root_field(names, check_irreducible=True, **kwds)

Return the \( p \)-adic extension field generated by the roots of the irreducible polynomial \( \text{self} \).

INPUT:

- names – name of the generator of the extension
- check_irreducible – check whether the polynomial is irreducible
- kwds – see \sage\text\text{ring.padics.padic.generic.pAdicGeneric.extension()}

EXAMPLES:

sage: R.<x> = Qp(3, 5, 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

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 Polymo

Bases: Polynomial generic cdv, 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:

```
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

dsage: x = O(3^5)
sage: li = [3^i * x for i in range(0,5)]; li
[O(3^5), 0(3^6), 0(3^7), 0(3^8), 0(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:

```python
sage: K = Qp(5)
sage: R.<t> = K[]
sage: f = 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:

```python
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:

```python
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),
0,
2 + 0(13^7)]
```
Polynomials, Release 10.1

\textbf{lshift_coeffs}(shift, no_list=False)

Return a new polynomials whose coefficients are multiplied by p^shift.

\textbf{EXAMPLES:}

\begin{verbatim}
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)
\end{verbatim}

\textbf{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 \texttt{NewtonPolygon}

\textbf{EXAMPLES:}

\begin{verbatim}
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)
\end{verbatim}

Here is an example where the computation fails because precision is not sufficient:

\begin{verbatim}
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
\end{verbatim}

\textbf{AUTHOR:}

• Xavier Caruso (2013-03-20)

\textbf{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 \texttt{repetition} is True, each slope is repeated a number of times equal to its multiplicity. Otherwise it appears only one time.

\textbf{INPUT:}

196 Chapter 2. Univariate Polynomials
• repetition – boolean (default True)

OUTPUT:
• a list of rationals

EXAMPLES:

```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:

```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 $n$ is 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**\( (n=\text{None}) \)

Return relative precision information about \textit{self}.

INPUT:

- \textit{self} – a p-adic polynomial
- \textit{n} – None or an integer (default None).

OUTPUT:

If \textit{n} is None, returns a list of relative precisions of coefficients. Otherwise, returns the relative precision of the coefficient of \(x^n\).

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.precision_relative()
[10, 10]
```

**quo_rem**\( (\text{right}, \text{secure}=\text{False}) \)

Return the quotient and remainder in division of \textit{self} by \textit{right}.

EXAMPLES:

```python
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 + O(3^10))^T + 1 + 3 + 2*3^2 + 2*3^3 + O(3^10),
2 + 3 + O(3^3) + O(3^10))
```

**rescale**\( (a) \)

Return \(f(a \cdot x)\).

Todo: Need to write this function for integer polynomials before this works.

EXAMPLES:

```python
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 \textit{f} is a degree-\textit{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: 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(shift, no_list=False)

Return a new polynomial whose coefficients are p-adically shifted to the right by shift.

**Note:** Type `Qp(5)(0).__rshift__?` for more information.

EXAMPLES:

```python
sage: K = Zp(13, 4)
sage: R.<t> = K[]
sage: a = t^2 + K(13,3)*t + 169; a
(1 + O(13^4))*t^2 + (13 + O(13^3))*t + 13^2 + O(13^6)
sage: b = a.rshift_coeffs(1); b
0*(13^3)*t^2 + (1 + O(13^2))*t + 13 + O(13^5)
sage: b.list()
[13 + O(13^5), 1 + O(13^2), 0(13^3)]
sage: b = a.rshift_coeffs(2); b
0*(13^2)*t^2 + 0(13)^*t + 1 + 0(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` is `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:
```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.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 is None, returns a list of valuations of coefficients. Otherwise, returns the valuation of the coefficient of $x^n$.

**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.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:** `Polynomial_generic_dense, Polynomial_padic`
2.1.17 Univariate Polynomials over GF(p^e) via NTL's ZZ_pEX

AUTHOR:
• Yann Laigle-Chapuy (2010-01) initial implementation
• Lorenz Panny (2023-01): minpoly_mod()

class sage.rings.polynomial.polynomial_zz_pex.Polynomial_ZZ_pEX

Bases: Polynomial_template

Univariate Polynomials over F_{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)`

Return True precisely when `self` is irreducible over its base ring.

INPUT:

• 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 F_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")
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(5^3)
sage: P = PolynomialRing(K, 'x')
sage: f = P.random_element(100)
```

(continues on next page)
### minpoly_mod(other)
Compute the minimal polynomial of this polynomial modulo another polynomial in the same ring.

**ALGORITHM:**

NTL's `MinPolyMod()`, which uses Shoup’s algorithm [Sho1999].

**EXAMPLES:**

```python
sage: R.<x> = GF(101^2)[]
sage: f = x^17 + x^2 - 1
sage: (x^2).minpoly_mod(f)
x^17 + 100*x^2 + 2*x + 100
```

### resultant(other)
Return 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: 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:**

```python
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
```
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 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
```

gc_cparent()

is_gen()

EXAMPLES:

```
sage: P.<x> = GF(2)[]
sage: x.is_gen()
True
sage: (x+1).is_gen()
False
```

is_one()

EXAMPLES:
Polynomials, Release 10.1

```python
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
```

`xgcd(other)`

Computes extended gcd of self and other.

EXAMPLES:

```python
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_zz_pex.make_element(parent, args)

## 2.1.18 Isolate Real Roots of Real Polynomials

**AUTHOR:**

- Carl Witty (2007-09-19): initial version

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:

- Precision reduction: at certain points in the computation, we discard the low-order bits of the coefficients, widening the intervals.
- 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.
- 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.

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 https://wiki.sagemath.org/days4schedule.

**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.

EXAMPLES:
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
[ 452/441 -290/147 395/441 132/49 -724/441
[-172/245 338/245 -37/49 -348/245 612/245

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:

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, -2032, -2500, -3000), 1)

class sage.rings.polynomial.real_roots.bernstein_polynomial_factory

Bases: object

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.

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)

Bases: bernstein_polynomial_factory

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)
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>
```

```python
sage: bpf = bernstein_polynomial_factory_ar(p, True)
sage: bpf.bernstein_polynomial(-20)
<IBP: ((-2965821, -2181962, -1542880, -1048576) + [0 .. 1)) * 2^-20>
```

```python
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_intlist([10, -20, 30, -40]).coeffs_bitsize()
1
```

class sage.rings.polynomial.real_roots.bernstein_polynomial_factory_intlist(coeffs)

Bases: 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>
```

coeffs_bitsize()

Computes the approximate log2 of the maximum of the absolute values of the coefficients.

EXAMPLES:
**class** `sage.rings.polynomial.real_roots.bernstein_polynomial_factory_ratlist(coefs)`

Bases: `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.)

**coeffs_bitsize()**

Computes the approximate log2 of the maximum of the absolute values of the coefficients.

**sage.rings.polynomial.real_roots.bernstein_up(d1, d2, s=None)**

Given polynomial degrees d1 and d2, where d1 < d2, compute a matrix bu.

If you have a Bernstein polynomial of formal degree d1, 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
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>
```

```python
from sage.rings.polynomial.real_roots import *
sage: bernstein_polynomial_factory_ratlist([1, 2, 3, -60000]).coeffs_bitsize() 16
sage: bernstein_polynomial_factory_ratlist([65535/65536]).coeffs_bitsize() -1
sage: bernstein_polynomial_factory_ratlist([65536/65535]).coeffs_bitsize() 1

sage: from sage.rings.polynomial.real_roots import *
sage: bernstein_polynomial_factory_ratlist([1, 2, 3, -60000]).coeffs_bitsize() 16
sage: bernstein_polynomial_factory_ratlist([65535/65536]).coeffs_bitsize() -1
sage: bernstein_polynomial_factory_ratlist([65536/65535]).coeffs_bitsize() 1
```

```python
from sage.rings.polynomial.real_roots import *
sage: 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>
```

```python
from sage.rings.polynomial.real_roots import *
sage: 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>
```
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:**
- $c_1$ – coefficients of polynomial over range $[0 .. x]$
- $c_2$ – coefficients of polynomial over range $[x .. 1]$
- $\text{err\textunderscore inc}$ – 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, 0.47727272727272727, 0.3254132231404959, 0.2218265214124724, 0.15127680827812312, 0.10314327837144759), (0.10314327837144759, 0.0, 0.0, 0.0, 0.0, 0.0), → 0.0), 15)
```

**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 $x = a/(a + b)$, 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 $x = a/(a + b)$ and $y = c/(c + d)$, 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:**
- $c_1$ – coefficients of polynomial over range $[0 .. x]$
- $c_2$ – coefficients of polynomial over range $[x .. 1]$

Chapter 2. Univariate Polynomials
• **err_inc** – amount by which error intervals widened

**EXAMPLES:**

```python
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)
((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:**

```python
degree_reduction_next_size(1000)
30
degree_reduction_next_size(20)
15
degree_reduction_next_size(3)
2
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:**

```python
m = matrix(3, range(9))
dprod_imatrow_vec(m, vector(ZZ, [1, 0, 0, 0]), 1)
0
dprod_imatrow_vec(m, vector(ZZ, [0, 1, 0, 0]), 1)
3
dprod_imatrow_vec(m, vector(ZZ, [0, 0, 1, 0]), 1)
4
dprod_imatrow_vec(m, vector(ZZ, [0, 0, 0, 1]), 1)
5
dprod_imatrow_vec(m, vector(ZZ, [1, 0, 0]), 1)
3
dprod_imatrow_vec(m, vector(ZZ, [0, 1, 0]), 1)
4
dprod_imatrow_vec(m, vector(ZZ, [0, 0, 1]), 1)
5
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:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: get_realfield_rndu(20)
Real Field with 20 bits of precision and rounding RNDU
sage: get_realfield_rndu(53)
Real Field with 53 bits of precision and rounding RNDU
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 \([a .. b]\), there are no 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
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)
```

(continues on next page)
Try doing a de Casteljau split of this polynomial at 1/2, resulting in polynomials $p_1$ and $p_2$. If we see that the sign of this polynomial is determined at 1/2, then return $(p_1, p_2, 1/2)$; otherwise, return None.

**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_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.

```python
class sage.rings.polynomial.real_roots.interval_bernstein_polynomial_float
    Bases: 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 \( E_1 \) and \( E_2 \). These represent the coefficients \( (A+E_1)*2^n <= c <= (A+E_2)*2^n \).

Note that we always have \( E_1 <= 0 <= E_2 \). 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; \( 0 < \text{mid} < 1 \)
- `msign` – default 0 (unknown); the sign of this polynomial at `mid`

**OUTPUT:**

- `bp1, bp2` – the new interval Bernstein polynomials

2.1. Univariate Polynomials and Polynomial Rings 215
• 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: # rel tol 2e-16
sage: bp1
<IBP: (0.5, 0.30000000000000004, -0.2555555555555555, -0.5444444444444444, -0.32172839506172846) + [-0.100000000000000069 .. 0.010000000000000677] over [0 .. 2/3]>
sage: bp2
<IBP: (-0.32172839506172846, -0.21037037037037046, 0.0288888888888888797, 0.04266666666666666, 0.99) + [-0.100000000000000069 .. 0.010000000000000677] over [2/3 .. 1]>
sage: bp1, bp2, ok = bp.de_casteljau(ctx, 7/39)
sage: # rel tol
sage: bp1
<IBP: (0.5, 0.4461538461538461, 0.36653517422748183, 0.27328680523946786, 0.99) + [-0.100000000000000069 .. 0.010000000000000677] over [0 .. 7/39]>
sage: bp2
<IBP: (-0.9999999999999999, -0.100000000000000069, -0.9999999999999999, -0.100000000000000069 .. 0.010000000000000677] 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:

```python
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: interval_bernstein_polynomial

This is the subclass of interval_bernstein_polynomial where polynomial coefficients are represented using integers.

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:

```python
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, # needs sage.symbolic
.....: error=2, scale_log2=-3, level=2, slope_err=RR(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() # needs sage.symbolic
(3, 3)
```

as_float()

Compute an interval_bernstein_polynomial_float which contains (or bounds) all the polynomials this interval polynomial contains (or bounds).

EXAMPLES:

```python
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 .. 0.0195312500000017]) * 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:

```python
sage: from sage.rings.polynomial.real_roots import *
sage: bp = mk_ibpi([50, 20, -90, -70, 200], error=5)
(continues on next page)
```
down_degree\(\text{(ctx, max\_err, exp\_err\_shift)}\)

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 0.7e2>
```

down_degree\_iter\(\text{(ctx, max\_scale)}\)

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 \(1 \ll max\_scale\).

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
```
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 = [a1, ..., an], and an integer error bound E, returns a vector of floating-point numbers B = [b1, ..., bn], lower and upper error bounds F1 and F2, and a scaling factor d, such that

\[(bk + F1) * 2^d \leq ak\]

and

\[ak + E \leq (bk + F2) * 2^d\]

If bj is the element of B with largest absolute value, then 0.5 <= abs(bj) < 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), -1.127502593849246e-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 \([r0 .. r]\) 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.
  
  This is 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))
  6.00000000000000
  sage: maximum_root_first_lambda((x+1)*(x+2)*(x+3))
  0.000000000000000
  sage: maximum_root_first_lambda(x^2 - 1)
  1.00000000000000

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:
  
  sage: from sage.rings.polynomial.real_roots import *
  sage: x = polygen(ZZ)
  sage: maximum_root_local_max((x-1)*(x-2)*(x-3))
  12.00000000000000
  sage: maximum_root_local_max((x+1)*(x+2)*(x+3))
  0.000000000000000
  sage: maximum_root_local_max(x^2 - 1)
  1.414213562373093

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:
  
  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)
Given a floating-point vector \( b = (b_0, \ldots, b_n) \), 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)
```

Given an integer vector \( b = (b_0, \ldots, b_n) \), 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)
```

A simple wrapper for creating context objects with coercions, defaults, etc.

For use in doctests.

**EXAMPLES:**

```python
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
```

A simple wrapper for creating \texttt{intervalbernsteinpolynomialfloat} objects with coercions, defaults, etc.

For use in doctests.

**EXAMPLES:**

```python
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
```

A simple wrapper for creating \texttt{intervalbernsteinpolynomialinteger} objects with coercions, defaults, etc.

For use in doctests.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.real_roots import *
sage: print(mk_ibpi([50, 20, -90, -70, 200], error=5))
degree 4 IBP with 8-bit coefficients
```

**class** \texttt{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()
Return True iff all islands are 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.all_done()
False
sage: oc.find_roots()
sage: oc.all_done()
True
```

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()
```

(continues on next page)
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:

```sage
def 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().)

    # Apply the increase_precision function
    increase_precision()
```

refine_all()

Refine all islands which are not done (which are not known to contain exactly one root).

EXAMPLES:

```sage
def refine_all():
    # Refine all islands which are not done.
    # (Which are not known to contain exactly one root.)

    # Apply the refine_all function
    refine_all()
```

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:

```sage
def 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.

    # Apply the reset_root_width function
    reset_root_width(isle_num, target_width)
```

(continues on next page)
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:

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, 11108889/14, 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)]

precompute_degree_reduction_cache(n)
Compute and cache the matrices used for degree reduction, starting from degree n.

EXAMPLES:

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)
)

class 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:
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: from sage.rings.polynomial.real_roots import *
sage: x = polygen(ZZ)
sage: real_roots(x^3 - x^2 - x - 1)

[((7/4, 19/8), 1)]

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, 55/4), 1), ((165/8, 341/16), 1), ((22, 44), 1)]

sage: real_roots(x^5 * (x^2 - 9999)^2 - 1)

[((-292744963813111/9007199254740992, 419601125186091/2251799813685248), 1),
  ((2126664501458494539510616544153249597/21267647932558653966460912964485513216,
    425331690272133001885369639953061621799/4253529586511370932912825928971026432),
    1),
  ((1063329226287740282451317352558954186101/106338239662793269832304656822427256608,
    53166461435386566760144520163085465433/5316911983139663491615228241121378304), 1)]

sage: real_roots(x^5 * (x^2 - 9999)^2 - 1, seed=42)

[((-123196838480289/18014398509481984, 293964743458749/9007199254740992), 1),
  ((8387259573979551909878419638198637143/83876749736572420564878941267521536,
    16614151915068103378913794877845325503/166153499473114484112975882535043072), 1),
  ((-519280373526592617581013249797434335/5192296858534827628530496329220096, 1),
    ((-404432689240106860312183/40446290987098377530888), 1)]

sage: real_roots(x^5 * (x^2 - 9999)^2 - 1, wordsize=64)

[((-628656380382302151500003/193428131133834066795298816, 90018055412564177624143/4835703278458516698824794), 1),
  ((5444245632373373152149909879228090501157/5444517870735015413939718908291383296,
    1088894127096660194637118845654929064385439/10889055741470030830827987437816582766592), 1),
  ((-272212281929661439711063928866060007142141/272225893536750770770699685945145691648,
    136166141275823501959100399337685485662633/13611294676837585385349429727072845824), 1)]

sage: real_roots(x)

[((-47/256, 81/512), 1)]

sage: real_roots(x * (x - 1))

[((-47/256, 81/512), 1), ((1/2, 1201/1024), 1)]

sage: real_roots(x-1)

[[(299/256, 593/512), 1]]

sage: real_roots(x*(x-1)*(x-2), bounds=(0, 2))

(continues on next page)
2.1. Univariate Polynomials and Polynomial Rings

```python
sage: real_roots(x^2 - 2, bounds=(0, 2), retval='algebraic_real')
[(1.000000000000?, 1), (1.000000000000?, 1)]
sage: real_roots((x-1) * (x-(v+1)/v), retval='algebraic_real')

sage: real_roots(x^2 - 2, max_diameter=1/2^30)
[(-2.?, 1), (2.?, 1)]
sage: real_roots(x^2 - 2, max_diameter=1/2^500)
```

(continues on next page)
Polynomials, Release 10.1

sage: real_roots((x-1) * (x-(v+1)/v), retval='interval')
[(1.000000000000000000?, 1), (1.000000000000000000?, 1)]
sage: real_roots((x-1) * (x-2), strategy='warp')
[((499/525, 1173/875), 1), ((337/175, 849/175), 1)]
sage: real_roots((x+3)*(x+1)*x*(x-1)*(x-2), strategy='warp', retval='algebraic_real')
[(-3.000000000000000000?, 1), (-1.000000000000000000?, 1), (0, 1), (1.000000000000000000?, 1), (2.000000000000000000?, 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 - 3*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 + 9*x^28 + 9*x^27 + 3*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 + x^7 - 7*x^6 + 7*x^5 - 4*x^4 + 4 + 12*x^3 - 6*x^2 + 2*x - 6
sage: cc = real_roots(p, retval='algebraic_real')[2][0] # long time
RealField(180)(cc) # long time
1.303577269034296391257099112152551890730725046594049

Now we play with algebraic real coefficients.

sage: x = polygen(AA)
sage: p = (x - 1) * (x - sqrt(AA(2))) * (x - 2)
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
[(-3.000000000000000000?, 1), (-1.000000000000000000?, 1), (0, 1), (1.000000000000000000?, 1), (2.000000000000000000?, 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 - 8.828427124746197*x^5 + 31.970562748477141*x^4 - 60.779552621700472*x^3 + 63.98526732578017*x^2 - 35.376134905855957*x + 8.0282842712474627
sage: real_roots(p2, retval='interval')
[(1.000?, 1), (1.17, 1), (1.38?, 1), (1.57, 1), (2.00?, 1), (2.17, 1)]
sage: p = (x - 1) * (x - sqrt(AA(2)))^2 * (x - 2)^3 * sqrt(AA(3))
sage: real_roots(p, retval='interval')
[(1.00000000000000?, 1), (1.414213562373095?, 2), (2.00000000000000?, 3)]

sage.rings.polynomial.real_roots.relative_bounds(a, b)

INPUT:

• (al, ah) – pair of rationals
• (bl, bh) – pair of rationals

OUTPUT:

• (cl, ch) – pair of rationals

Computes the linear transformation that maps (al, ah) to (0, 1); then applies this transformation to (bl, bh) and returns the result.

EXAMPLES:

sage: from sage.rings.polynomial.real_roots import *
sage: relative_bounds((1/7, 1/4), (1/6, 1/5))
(2/9, 8/15)

sage.rings.polynomial.real_roots.reverse_intvec(c)

Given a vector of integers, reverse the vector (like the reverse() method on lists).

Modifies the input vector; has no return value.

EXAMPLES:

sage: from sage.rings.polynomial.real_roots import *
sage: v = vector(ZZ, [1, 2, 3, 4]); v
(1, 2, 3, 4)
sage: reverse_intvec(v)
sage: v
(4, 3, 2, 1)

sage.rings.polynomial.real_roots.root_bounds(p)

Given a polynomial with real coefficients, computes a lower and upper bound on its real roots. Uses algorithms of Akritas, Strzebo'nski, and Vigklas.

EXAMPLES:

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))
((-1.000000000000000, 0.000000000000000))
sage: root_bounds((x^2)*(x-3))
((-2.44948974278317, 3.46410161513776))
sage: root_bounds(x^995 * (x^2 - 9999) - 1)
((-99.9949998749937, 141.414284992713))

(continues on next page)
Polynomials, Release 10.1

sage: root_bounds(x^995 * (x^2 - 9999) + 1)
(-141.414284992712, 99.9949998749938)

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

Bases: object

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()

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 $[-\infty .. 0]$; or, $(u1, u2)$ can be replaced by None, meaning $[1 .. \infty]$.

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:

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, -232 Chapter 2. Univariate Polynomials

(continues on next page)
Given a vector of integers $c$ of length $d+1$, representing the coefficients of a degree-$(d-1)$ 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: 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)
(x^3 - 3*x^2 + 2*x, (0, 2, -3, 1))
```

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: to_bernstein(x)
([0, 1], 1)
sage: to_bernstein(x, degree=5)
([0, 1/5, 2/5, 3/5, 4/5, 1], 1)
sage: to_bernstein(x^3 + x^2 - x - 1, low=-3, high=3)
([-16, 24, -32, 32], 1)
sage: to_bernstein(x^3 + x^2 - x - 1, low=3, high=22/7)
([296352, 310464, 325206, 340605], 9261)
```

Given a polynomial $p$ with rational coefficients, compute the exact rational Bernstein coefficients of $p(x/(x+1))$.

**EXAMPLES:**

```python
sage: to_bernstein_warp(p)
```
```
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)`

Bases: `object`

A class to map between original coordinates and ocean coordinates. If `neg` is False, then the original->ocean transform is \( x \rightarrow \frac{x}{x+1} \), and the ocean->original transform is \( \frac{x}{1-x} \); this maps between \([0 .. infinity]\) and \([0 .. 1]\). If `neg` is True, then the original->ocean transform is \( x \rightarrow -\frac{x}{1-x} \), and the ocean->original transform is the same thing: \( -\frac{x}{1-x} \). 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 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:**

```
sage: from sage.rings.polynomial.real_roots import *
sage: wordsize_rational(1/5, 1/7, 32)
429496729/2147483648
sage: wordsize_rational(1/7, 1/5, 32)
306783379/2147483648
sage: wordsize_rational(1/5, 1/7, 64)
1844674407370955161/9223372036854775808
sage: wordsize_rational(1/7, 1/5, 64)
658812288346769701/4611686018427387904
sage: wordsize_rational(1/17, 1/19, 32)
252645135/4294967296
sage: wordsize_rational(1/17, 1/19, 64)
1085102592571150095/18446744073709551616
```

(continues on next page)
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)
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? - 1.083954101317711?*I, 1), (0.181232444469876? + 1.083954101317711?*I, 1)]
```

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), (0.181232444469876? - 1.083954101317711?*I, 1), (0.181232444469876? + 1.083954101317711?*I, 1)]
```

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:
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

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 \((x-1)^5\):

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.

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.

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
....:     if tiny(x.real()):
....:         return CIF(0, x.imag())

sage: rts = complex_roots(p); type(rts[0][0]), sorted(map(smash, rts))
(<class 'sage.rings.complex_interval.ComplexIntervalFieldElement'>, [-1.618033988749895?, -0.618033988749895?*I, 1.618033988749895?*I, 0.618033988749895?])

sage: rts = complex_roots(p, retval='algebraic'); type(rts[0][0]), rts
(<class 'sage.rings.qqbar.AlgebraicNumber'>, [-1.618033988749895?, -0.618033988749895?*I, 1.618033988749895?*I, 0.618033988749895?])

sage: rts = complex_roots(p, retval='algebraic_real'); type(rts[0][0]), rts
(<class 'sage.rings.qqbar.AlgebraicReal'>, [-1.618033988749895?, 1], (0.618033988749895?, 1))

sage.rings.polynomial.complex_roots.interval_roots(p, rts, prec)

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]]
```

```python
sage: from sage.rings.polynomial.complex_roots import intervals_disjoint
sage: intervals_disjoint([a,b,c,d])
False
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: ip, ipd, irt, fld
sage: refine_root(ip, ipd, irt, fld)
```

We are given a polynomial and its derivative (with complex interval coefficients), an estimated root, and a complex interval field to use in computations. We use interval arithmetic to refine the root and prove that we have in fact isolated a unique root.

If we succeed, we return the isolated root; if we fail, we return None.

EXAMPLES:

```python
sage: # needs sage.symbolic
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.

**class** sage.rings.polynomial.ideal.Ideal_1poly_field(ring, gen)

**Bases:** 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:**

```
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()**

Return 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 $P \subset F_p[t]$, return 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'); a = S.gen()
sage: S
Univariate Quotient Polynomial Ring in a
```

class sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRingFactory
Bases: UniqueFactory
Create a quotient of a polynomial ring.

INPUT:

• ring - a univariate polynomial ring
• 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

$$R = \left(\mathbb{F}_2[y]/(y^2 + y + 1)\right)[x]/(x^3 - 5).$$

```
sage: # needs sage.libs.ntl
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 $\mathbb{Q}$:

```
sage: R = PolynomialRing(RationalField(), 'x'); x = R.gen()
sage: S = R.quotient(x^3 + 2*x - 5, 'a'); S
Univariate Quotient Polynomial Ring in a over Rational Field with modulus x^3 + 2*x - 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 + 2*x - 5
sage: K.polynomial_quotient_ring()
Univariate Quotient Polynomial Ring in a over Rational Field with modulus x^3 + 2*x - 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 github issue #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:

sage: R.<x> = QQ[]
sage: PolynomialQuotientRing.create_object((8, 0, 0),
.....:               (R, x^2 - 1, ('xbar')))  
Univariate Quotient Polynomial Ring in xbar over Rational Field with modulus x^2 - 1

class sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_coercion
Bases: DefaultConvertMap_unique
A coercion map from a PolynomialQuotientRing to a PolynomialQuotientRing that restricts to the coercion map on the underlying ring of constants.

2.1. Univariate Polynomials and Polynomial Rings 241
EXAMPLES:

```sage
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
sage: f.is_injective()
        True
```

**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:

```sage
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:

```sage
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:

```sage
sage: # needs sage.rings.padics
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(characteristics, names, category=None)`

Bases: `PolynomialQuotientRing_generic`, `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)`

Take a polynomial quotient ring, and return a tuple with three elements: the \texttt{NumberField} defined by the same polynomial quotient ring, a homomorphism from its parent to the \texttt{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: # needs sage.rings.number_field
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>
   # needs sage.rings.finite_rings
sage: S.<a> = Rquo(x^3 - 2)
   # needs sage.rings.finite_rings
sage: F, g, h = S.field_extension('b')
   # needs sage.rings.finite_rings
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: # needs sage.rings.finite_rings
sage: R.<x> = GF(25, 'a')<x>
(continues on next page)
```
Polynomials, Release 10.1

We do an example involving a relative number field:

```sage
sage: # needs sage.rings.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
  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
sage: # needs sage.rings.number_field
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))
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: PolynomialQuotientRing_domain, Field

EXAMPLES:

```python
sage: # needs sage.rings.number_field
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
```

```python
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:

```python
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: f = x^5 + x + 17
sage: k = R.quotient(f)
sage: v = k.complex_embeddings(100)
sage: [phi(k.0^2) for phi in v]
[2.9757207403766761469671194565,
 -2.408899437163850098316292196 - 1.9025410530350528612407363802*I,
 -2.408899437163850098316292196 + 1.9025410530350528612407363802*I,
 0.92106969730469364806949137 - 3.07553311884577947326541806*I,
 0.92106969730469364806949137 + 3.07553311884577947326541806*I]
```

class sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic(ring, polynomial, name=None, category=None)

Bases: QuotientRing_generic

Quotient of a univariate polynomial ring by an ideal.

EXAMPLES:
We demonstrate object persistence.

\[
sage: \text{loads(S.dumps()) == S} \\
\text{True} \\
sage: \text{loads(xbar.dumps()) == xbar} \\
\text{True}
\]

We create some sample homomorphisms:

\[
sage: R.<x> = PolynomialRing(ZZ) \\
sage: S = R.quo(x^2 - 4) \\
sage: f = S.hom([2]) \\
sage: f \\
\text{Ring morphism:} \\
\text{From: Univariate Quotient Polynomial Ring in xbar over Integer Ring} \\
\text{with modulus x^2 - 4} \\
\text{To: Integer Ring} \\
\text{Defn: xbar |--> 2} \\
sage: f(x) \\
2 \\
sage: f(x^2 - 4) \\
0 \\
sage: f(x^2) \\
4
\]

**Element**

alias of `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:
When we include the prime \((2, -a + 1)\), the \(S\)-class group becomes trivial:

```sage
sage: S.S_class_group([K.ideal(2, -a+1)])
# needs sage.rings.number_field
[]
```

Here is an example where the base and the extension both contribute to the class group:

```sage
sage: # needs sage.rings.number_field
sage: K.<a> = QuadraticField(-5)

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)]
```

Now we take an example over a nontrivial base with two factors, each contributing to the class group:

```sage
sage: # needs sage.rings.number_field
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([[]])
# not tested
[((1/4*xbar^2 + 31/4, (-1/8*a + 1/8)*xbar^2 - 31/8*a + 31/8, -1/16*a^2+3 + 1/16*a^2+2 + 31/16*a^2+31/16, 6),
 ((-1/4*xbar^2 - 23/4, (1/8*a - 1/8)*xbar^2 + 23/8*a - 23/8, -1/16*a^2+3 - 1/16*a^2+2 - 23/16*a^2 - 23/16, 6),
 ((-5/4*xbar^2 - 115/4, 1/4*a^2+2 + 23/4*a, -1/16*a^2+3 - 7/16*a^2+2 - 23/16*a^2 + 161/16, 6),
 ((1/4*a^2+2 + 3/4*a, 1/16*a^2+3 - 1/16*a^2+2 - 23/16*a^2 - 23/16, 6),
 ((-1/8*a + 1/8)*xbar^2 + 31/8*a + 31/8, 1/16*a^2+3 + 1/16*a^2+2 + 31/16*a^2+31/16, -1/16*a^2+3 + (-1/16*a + 1/8)*xbar^2 - 31/16*a^2+31/16, 6),
 ((1/4*a^2+2 + 23/4*a, 1/16*a^2+3 - 1/16*a^2+2 - 23/16*a^2 - 23/16, 6),
 ((-1/8*a + 1/8)*xbar^2 + 31/8*a + 31/8, 1/16*a^2+3 + 1/16*a^2+2 + 31/16*a^2+31/16, -1/16*a^2+3 + (-1/16*a + 1/8)*xbar^2 - 31/16*a^2+31/16, 6),
 ((-5/4*xbar^2 - 115/4, 1/4*a^2+2 + 23/4*a, -1/16*a^2+3 - 7/16*a^2+2 - 23/16*a^2 + 161/16, 6),
 ((1/4*a^2+2 + 3/4*a, 1/16*a^2+3 - 1/16*a^2+2 - 23/16*a^2 - 23/16, 6)]
```

2.1. Univariate Polynomials and Polynomial Rings

247
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 github issue #14489):

```python
sage: S.S_class_group([K.ideal(a)])  # representation varies  # not tested,
 "needs sage.rings.number_field"
[[(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,  
  -1/16*a*xbar^3 + (1/16*a + 1/8)*xbar^2 - 31/16*a*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*xbar - 23/16*a - 23/8),  
  2)]
```

Note that all the returned values live where we expect them to:

```python
sage: # needs sage.rings.number_field
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])
<class '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 \((\text{gen}, \text{order})\), where \(\text{gen}\) is a unit of order \(\text{order}\).

**EXAMPLES:**

```python
sage: K.<a> = QuadraticField(-3)  # needs sage.rings.number_field
sage: K.unit_group()  # needs sage.rings.number_field
Unit group with structure C6 of Number Field in a with defining polynomial x^2 + 3 with a = 1.732050807568878?*I
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = QQ['x'].quotient(x^2 + 3)
sage: u, o = K.S_units([])[0]; o  
6
sage: 2*u - 1 in {a, -a}
True
sage: u^6  
1
sage: u^3
```

(continues on next page)
-1
\[
\text{sage: } 2 u^2 + 1 \text{ in } \{a, -a\}
\]
True

\[
\text{sage: } \# \text{ needs sage.rings.number_field}
\]
\[
\text{sage: } K.<a> = QuadraticField(-3)
\]
\[
\text{sage: } y = \text{polygen}(K)
\]
\[
\text{sage: } L.<b> = K[y].quotient(y^3 + 5); L
\]
Univariate Quotient Polynomial Ring in b over Number Field in a
\[
\text{with defining polynomial } x^2 + 3 \text{ with } a = 1.732050807568878?I
\]
\[
\text{with modulus } y^3 + 5
\]
\[
\text{sage: } [u \text{ for } u, o \text{ in } L.S_units([]) \text{ if } o \text{ is Infinity}]
\]
\[
[(-1/3*a - 1)*b^2 - 4/3*a*b - 5/6*a + 7/2,
2/3*a*b^2 + (2/3*a - 2)*b - 5/6*a - 7/2]
\]
\[
\text{sage: } [u \text{ for } u, o \text{ in } L.S_units([K.ideal(1/2*a - 3/2)])]
\]
\[
\text{....: if } o \text{ is Infinity}]
\[
[(-1/6*a - 1/2)*b^2 + (1/3*a - 1)*b + 4/3*a,
(-1/3*a - 1)*b^2 - 4/3*a*b - 5/6*a + 7/2,
2/3*a*b^2 + (2/3*a - 2)*b - 5/6*a - 7/2]
\]
\[
\text{sage: } [u \text{ for } u, o \text{ in } L.S_units([K.ideal(2)]) \text{ if } o \text{ is Infinity}]
\]
\[
[(1/2*a - 1/2)*b^2 + (a + 1)*b + 3,
(1/6*a + 1/2)*b^2 + (-1/3*a + 1)*b - 5/6*a + 1/2,
(1/6*a + 1/2)*b^2 + (-1/3*a + 1)*b - 5/6*a - 1/2,
(-1/3*a - 1)*b^2 - 4/3*a*b - 5/6*a + 7/2,
2/3*a*b^2 + (2/3*a - 2)*b - 5/6*a - 7/2]
\]

Note that all the returned values live where we expect them to:

\[
\text{sage: } \# \text{ needs sage.rings.number_field}
\]
\[
\text{sage: } U = L.S_units([])
\]
\[
\text{sage: } \text{type}(U[0][0])
\]
<class 'sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_field_with_category.element_class'>
\[
\text{sage: } \text{type}(U[0][1])
\]
<class 'sage.rings.integer.Integer'>
\[
\text{sage: } \text{type}(U[1][1])
\]
<class 'sage.rings.infinity.PlusInfinity'>

ambient()

base_ring()

Return the base ring of the polynomial ring, of which this ring is a quotient.

EXAMPLES:
The base ring of \(\mathbb{Z}[z]/(z^3 + z^2 + z + 1)\) is \(\mathbb{Z}\).

\[
\text{sage: } R.<z> = PolynomialRing(ZZ)
\]
\[
\text{sage: } S.<beta> = R.quo(z^3 + z^2 + z + 1)
\]
\[
\text{sage: } S.base_ring()
\]
Integer Ring

Next we make a polynomial quotient ring over \(S\) and ask for its base ring.

2.1. Univariate Polynomials and Polynomial Rings 249
Polynomials, Release 10.1

```python
sage: T.<t> = PolynomialRing(S)
sage: W = T.quotient(t^99 + 99)
sage: W.base_ring()
Univariate Quotient Polynomial Ring in beta
over Integer Ring with modulus z^3 + z^2 + z + 1
```

cardinality()

Return the number of elements of this quotient ring.

order is an alias of cardinality.

EXAMPLES:

```python
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
```

```python
sage: # needs sage.rings.finite_rings
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
```

class_group(proof=True)

If self is a quotient ring of a polynomial ring over a number field \( K \), by a polynomial of nonzero discriminant, return a list of generators of the class group.

```python
sage: R.<z> = PolynomialRing(ZZ)
sage: S.<a> = R.quo(z - 19)
sage: S.characteristic()
0
sage: R.<x> = PolynomialRing(GF(9, 'a'))
# needs sage.rings.finite_rings
sage: S = R.quotient(x^3 + 1)
# needs sage.rings.finite_rings
sage: S.characteristic()
3
```
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 \((\text{gen}, \text{order})\), where \(\text{gen}\) is a tuple of generators of an ideal \(I\) and \(\text{order}\) is the order of \(I\) in the class group.

INPUT:
- \text{proof} - if False, assume the GRH in computing the class group

OUTPUT:
A list of pairs \((\text{gen}, \text{order})\), where \(\text{gen}\) is a tuple of elements generating a fractional ideal and \(\text{order}\) is the order of \(I\) in the class group.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: K.<a> = QuadraticField(-3)
sage: K.class_group()
Class group of order 1 of Number Field in a with defining polynomial x^2 + 3 with a = 1.732050807568878?*I
sage: x = polygen(QQ, 'x')
sage: K.<a> = QQ[x].quotient(x^2 + 3)
sage: K.class_group()
[]
```

A trivial algebra over \(\mathbb{Q}(\sqrt{-5})\) has the same class group as its base:

```
sage: # needs sage.rings.number_field
sage: K.<a> = QuadraticField(-5)
sage: R.<x> = K[]
sage: S.<xbar> = R.quotient(x)
sage: S.class_group()
[[(2, -a + 1), 2]]
```

The same algebra constructed in a different way:

```
sage: x = polygen(ZZ, 'x')
sage: K.<a> = QQ[x].quotient(x^2 + 5)
sage: K.class_group() # needs sage.rings.number_field
[[(2, a + 1), 2]]
```

Here is an example where the base and the extension both contribute to the class group:

```
sage: # needs sage.rings.number_field
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:
**Polynomials, Release 10.1**

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: S.<xbar> = R.quotient((x^2 + 23) * (x^2 + 47))
sage: S.class_group()
[((1/12*xbar^2 + 47/12,
  1/48*xbar^3 - 1/48*xbar^2 + 47/48*xbar - 47/48),
  3),
 ((-1/12*xbar^2 - 23/12,
  -1/48*xbar^3 - 1/48*xbar^2 - 23/48*xbar - 23/48),
  5)]
```

Now we take an example over a nontrivial base with two factors, each contributing to the class group:

```
sage: # needs sage.rings.number_field
sage: K.<a> = QuadraticField(-5)
sage: R.<x> = K[]
sage: S.<xbar> = R.quotient((x^2 + 23) * (x^2 + 31))
sage: S.class_group()
# not tested
[((1/4*xbar^2 + 31/4,
  (-1/8*a + 1/8)*xbar^2 - 31/8*a + 31/8,
  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*a*xbar^3 - 1/16*a*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*a*xbar^3 - 7/16*a*xbar^2 - 23/16*xbar - 161/16, 
  1/16*a*xbar^3 - 1/16*a*xbar^2 + 23/16*a^2*xbar - 23/16*a),
  2)]
```

Note that all the returned values live where we expect them to:

```
sage: # needs sage.rings.number_field
sage: CG = S.class_group()
```

```
sage: type(CG[0][0][1])
<class 'sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_element_class'>
sage: type(CG[0][1])
<class 'sage.rings.integer.Integer'>
```

**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
```

(continues on next page)
AUTHOR:
– Simon King (2010-05)

cover_ring()
Return the polynomial ring of which this ring is the quotient.

EXAMPLES:

```sage
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
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
sage: R.<x> = PolynomialRing(QQ)
sage: S = R.quotient((x + 1) * (x + 1))
sage: S.discriminant()
0
```
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
sage: S = R.quotient(x^2 - 8)
sage: S.number_field().discriminant() # needs sage.rings.number_field
8
sage: S.discriminant()
32
```
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:**

```python
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:**

```python
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:

```python
sage: # needs sage.rings.padics
sage: R1.<x> = Qp(2)[]
sage: F1 = R1.quotient_ring(x^2 + 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)
False
```

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')[]
# needs sage.rings.finite_rings
```

(continues on next page)
sage: R.quo(2*x^3 + x + 1).is_finite()  # needs sage.rings.finite_rings
True

sage: R.quo(2).is_finite()  # needs sage.rings.finite_rings
True

sage: P.<v> = GF(2)[]
sage: P.quotient(v^2 - v).is_finite()
True

is_integral_domain(proof=True)

Return whether or not this quotient ring is an integral domain.

EXAMPLES:

sage: R.<z> = PolynomialRing(ZZ)
sage: S = R.quotient(z^2 - z)
sage: S.is_integral_domain()
False
sage: T = R.quotient(z^2 + 1)
sage: T.is_integral_domain()
True
sage: U = R.quotient(-1)
sage: U.is_integral_domain()
False
sage: R2.<y> = PolynomialRing(R)
sage: S2 = R2.quotient(z^2 - y^3)
sage: S2.is_integral_domain()  # needs sage.libs.singular
True
sage: S3 = R2.quotient(z^2 - 2*y*z + y^2)
sage: S3.is_integral_domain()  # needs sage.libs.singular
False
sage: R.<z> = PolynomialRing(ZZ.quotient(4))
sage: S = R.quotient(z - 1)
sage: S.is_integral_domain()
False

krull_dimension()

Return the Krull dimension.

This is the Krull dimension of the base ring, unless the quotient is zero.

EXAMPLES:

sage: x = polygen(ZZ, 'x')
sage: R = PolynomialRing(ZZ, 'x').quotient(x**6 - 1)
sage: R.krull_dimension()
1
sage: R = PolynomialRing(ZZ, 'x').quotient(1)

lift(x)

Return an element of the ambient ring mapping to the given argument.

EXAMPLES:

```
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:

```
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: # needs sage.rings.number_field
sage: R.<x> = PolynomialRing(QQ)
sage: S.<alpha> = R.quotient(x^29 - 17*x - 1)
sage: K = S.number_field(); 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: # needs sage.rings.finite_rings  
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
```

classical_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: # needs sage.rings.finite_rings  
sage: F1.<a> = GF(2^7)
sage: P1.<x> = F1[]
sage: F2 = F1.extension(x^2 + x + 1, 'u')
sage: F2.random_element().parent() is F2
True
```
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

\textbf{selmer\_generators}(\(S, m, proof=True\))

If self is an \(\text{étale}\) 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 \(D(\sqrt[2]{a})/D\) is unramified at all primes of \(D\) lying above a place outside of \(S\).

\textbf{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

\textbf{OUTPUT}:

A list of generators of \(D(S, m)\).

\textbf{EXAMPLES}:

\begin{verbatim}
sage: # needs sage.rings.number_field
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)], 3)
[2, a + 1, -a]
sage: D.selmer_generators([K.ideal(2, -a + 1), K.ideal(3, a + 1)], 3)
[2, a + 1, -a]
\end{verbatim}

\textbf{selmer\_group}(\(S, m, proof=True\))

If self is an \(\text{étale}\) 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 \(D(\sqrt[2]{a})/D\) is unramified at all primes of \(D\) lying above a place outside of \(S\).

\textbf{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: # needs sage.rings.number_field
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)], 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 \((\text{gen}, \text{order})\), where \text{gen} is a unit of order \text{order}.

EXAMPLES:

```python
sage: # needs sage.rings.number_field
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: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = QQ[x].quotient(x^2 + 3)
```

(continues on next page)
sage: u = K.units()[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: x = polygen(ZZ, 'x')
sage: K.<a> = QQ['x'].quotient(x^2 + 5)
sage: K.units()
[(-1, 2)]

sage: # needs sage.rings.number_field
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: L.<b> = K.extension(y^3 + 5)
sage: L.unit_group()
Unit group with structure C6 x Z x Z of
Number Field in b with defining polynomial x^3 + 5 over its base field
sage: L.unit_group().gens()
# abstract generators
(u0, u1, u2)
sage: L.unit_group().gens_values()[1:]
[(-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]

Note that all the returned values live where we expect them to:

sage: # needs sage.rings.number_field
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])
<class 'sage.rings.integer.Integer'>
sage: type(U[1][1])
<class 'sage.rings.infinity.PlusInfinity'>

sage.rings.polynomial.polynomial_quotient_ring.is_PolynomialQuotientRing(x)
2.1.23 Elements of Quotients of Univariate Polynomial Rings

EXAMPLES: We create a quotient of a univariate polynomial ring over \( \mathbb{Z} \).

\[
\begin{align*}
sage: & \text{R.<x> = ZZ[]} \\
sage: & \text{S.<a> = R.quotient(x^3 + 3*x - 1)} \\
sage: & 2 * a^3 \\
& -6*a + 2
\end{align*}
\]

Next we make a univariate polynomial ring over \( \mathbb{Z}[x]/(x^3 + 3x - 1) \).

\[
\begin{align*}
sage: & \text{S1.<y> = S[]} \\
\end{align*}
\]

And, we quotient out that by \( y^2 + a \).

\[
\begin{align*}
sage: & \text{T.<z> = S1.quotient(y^2 + a)} \\
\end{align*}
\]

In the quotient \( z^2 \) is \( -a \).

\[
\begin{align*}
sage: & \text{z^2} \\
& -a
\end{align*}
\]

And since \( a^3 = -3x + 1 \), we have:

\[
\begin{align*}
sage: & \text{z^6} \\
& 3*a - 1
\end{align*}
\]

\[
\begin{align*}
sage: & \text{R.<x> = PolynomialRing(Integers(9))} \\
sage: & \text{S.<a> = R.quotient(x^4 + 2*x^3 + x + 2)} \\
sage: & a^100 \\
& 7*a^3 + 8*a + 7
\end{align*}
\]

\[
\begin{align*}
sage: & \text{R.<x> = PolynomialRing(QQ)} \\
sage: & \text{S.<a> = R.quotient(x^3 - 2)} \\
sage: & a \\
& a \\
sage: & a^3 \\
& 2
\end{align*}
\]

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 \( a^3 \).

\[
\begin{align*}
sage: & \text{a == x} \\
& True \\
sage: & \text{a^3 == x^3} \\
& True \\
sage: & \text{x^3} \\
& x^3 \\
sage: & \text{S(x^3)} \\
& 2
\end{align*}
\]

AUTHORS:

- William Stein
class sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRingElement(parent, polynomial, check=True)

Bases: Polynomial_singular_repr, CommutativeRingElement

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)  # needs sage.libs.singular
xi
sage: (singular(xi)*singular(xi)).NF('std(0)')  # needs sage.libs.singular
-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.quotient(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
```

`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: # needs sage.rings.number_field
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: # needs sage.rings.finite_rings
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: # needs sage.rings.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: # needs sage.rings.number_field
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
```

(continues on next page)
AUTHORS:
- Craig Citro (2006-08-06)
- William Stein (2006-08-06)

**is_unit()**
Return True if self is invertible.

**Warning:** Only implemented when the base ring is a field.

**EXAMPLES:**
```
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:**
```
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:**
```
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 the `matrix()` of this element.

ALGORITHM: Use `minpoly_mod()` if possible, otherwise compute the minimal polynomial of the `matrix()`.

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quotient(x^3 + 2*x - 5)
sage: (a + 123).minpoly()
x^3 - 369*x^2 + 45389*x - 1861118
sage: (a + 123).matrix().minpoly()
x^3 - 369*x^2 + 45389*x - 1861118
```

One useful application of this function is to compute a minimal polynomial of a finite-field element over an intermediate extension, rather than the absolute minimal polynomial over the prime field:

```python
sage: # needs sage.rings.finite_rings
sage: R.<x> = GF(65537)
```

```python
sage: F2.<i> = GF((431,2), modulus=[1,0,1])
sage: F6.<u> = F2.extension(3)
sage: (u + 1).minpoly()
x^6 + 425*x^5 + 19*x^4 + 125*x^3 + 189*x^2 + 239*x + 302
sage: ext = F6.over(F2)
sage: ext(u + 1).minpoly()  # indirect doctest
x^3 + (396*i + 428)*x^2 + (80*i + 39)*x + 9*i + 178
```

norm()

The norm of this element, which is the determinant of the matrix of right multiplication by this element.

EXAMPLES:

```python
sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quotient(x^3 - 389*x^2 + 2*x - 5)
sage: a.norm()
5
```

rational_reconstruction(*args, **kwargs)

Compute a rational reconstruction of this polynomial quotient ring element to its cover ring.

This method is a thin convenience wrapper around `Polynomial.rational_reconstruction()`.

EXAMPLES:

```python
sage: # needs sage.rings.finite_rings
sage: R.<x> = GF(65537)
```
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 CompiledPolynomialFunction 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: binary_pd
2.1.25 Polynomial multiplication by Kronecker substitution

2.1.26 Integer-valued polynomial rings

AUTHORS:
• Frédéric Chapoton (2023-03): Initial version

```python
class sage.rings.polynomial.integer_valued_polynomials.IntegerValuedPolynomialRing(R)
    Bases: UniqueRepresentation, Parent

The integer-valued polynomial ring over a base ring $R$.

Integer-valued polynomial rings are commutative and associative algebras, with a basis indexed by non-negative integers.

There are two natural bases, made of the sequence $\binom{x}{n}$ for $n \geq 0$ (the binomial basis) and of the other sequence $\binom{x+n}{n}$ for $n \geq 0$ (the shifted basis).

These two bases are available as follows:

```sage```
B = IntegerValuedPolynomialRing(QQ).Binomial()
sage: S = IntegerValuedPolynomialRing(QQ).Shifted()
```

or by using the shortcuts:
There is a conversion formula between the two bases:

\[
\binom{x}{i} = \sum_{k=0}^{i} (-1)^{i-k} \binom{i}{k} \binom{x+k}{k}
\]

with inverse:

\[
\binom{x+i}{i} = \sum_{k=0}^{i} \binom{i}{k} \binom{x}{k}.
\]

REFERENCES:
- Wikipedia article Integer-valued polynomial

**B**
alias of *Binomial*

class **Bases**(parent_with_realization)

Bases: Category_realization_of_parent
class **ElementMethods**

Bases: object

**content()**

Return the content of self.

This is the gcd of the coefficients.

**EXAMPLES:**

```sage
sage: F = IntegerValuedPolynomialRing(QQ).S()
sage: B = F.basis()
sage: (3*B[4]+6*B[7]).content()
sage: 3
```

**polynomial()**

Convert to a polynomial in \(x\).

**EXAMPLES:**

```sage
sage: F = IntegerValuedPolynomialRing(QZ).S()
sage: B = F.basis()
sage: (B+1).polynomial()
x + 2
sage: F = IntegerValuedPolynomialRing(QZ).B()
sage: B = F.basis()
sage: (B+1).polynomial()
x + 1
```

**shift**

Shift all indices by \(j\).

**INPUT:**
Polynomials, Release 10.1

• \( j \) – integer (default: 1)
In the binomial basis, the shift by 1 corresponds to a summation operator from 0 to \( x \).

EXAMPLES:

```python
sage: F = IntegerValuedPolynomialRing(ZZ).B()
sage: B = F.gen()
sage: (B+1).shift()
sage: (B+1).shift(3)
```

`sage: F = IntegerValuedPolynomialRing(ZZ).S()
sage: B = F.basis()
sage: (B[2]*B[4]).sum_of_coefficients()
1
```

```python
class ParentMethods
Bases: object

algebra_generators()
Return the generators of this algebra.

EXAMPLES:

```python
sage: A = IntegerValuedPolynomialRing(ZZ).S(); A
Integer-Valued Polynomial Ring over Integer Ring
in the shifted basis
sage: A.algebra_generators()
Family (S[1],)
```

```python
degree_on_basis(m)
Return the degree of the basis element indexed by \( m \).

EXAMPLES:

```python
sage: A = IntegerValuedPolynomialRing(QQ).S()
sage: A.degree_on_basis(4)
4
```

```
from_polynomial(p)
Convert a polynomial into the ring of integer-valued polynomials.

This raises a ValueError if this is not possible.

INPUT:
• \( p \) – a polynomial in one variable

EXAMPLES:

```python
```
sage: A = IntegerValuedPolynomialRing(ZZ).S()
sage: S = A.basis()
sage: S[5].polynomial()
1/120*x^5 + 1/8*x^4 + 17/24*x^3 + 15/8*x^2 + 137/60*x + 1
sage: A.from_polynomial(_)
S[5]
sage: x = polygen(QQ, 'x')
sage: A.from_polynomial(x)
-S[0] + S[1]

sage: A = IntegerValuedPolynomialRing(ZZ).B()
sage: B = A.basis()
sage: B[5].polynomial()
1/120*x^5 - 1/12*x^4 + 7/24*x^3 - 5/12*x^2 + 1/5*x
sage: A.from_polynomial(_)
B[5]
sage: x = polygen(QQ, 'x')
sage: A.from_polynomial(x)
B[1]

\texttt{gen}(i=0)

Return the generator of this algebra.

The optional argument is ignored.

EXAMPLES:

\begin{verbatim}
sage: F = IntegerValuedPolynomialRing(ZZ).B()
sage: F.gen()
B[1]
\end{verbatim}

\texttt{gens}()

Return the generators of this algebra.

EXAMPLES:

\begin{verbatim}
sage: A = IntegerValuedPolynomialRing(ZZ).S(); A
Integer-Valued Polynomial Ring over Integer Ring in the shifted basis
sage: A.algebra_generators()
Family (S[1],)
\end{verbatim}

\texttt{one_basis}()

Return the number 0, which index the unit of this algebra.

EXAMPLES:

\begin{verbatim}
sage: A = IntegerValuedPolynomialRing(QQ).S()
sage: A.one_basis()
0
sage: A.one()
S[0]
\end{verbatim}

\texttt{super_categories}()

Return the super-categories of \texttt{self}.  

\section*{Chapter 2. Univariate Polynomials}
EXAMPLES:

```python
sage: A = IntegerValuedPolynomialRing(QQ); A
Integer-Valued Polynomial Ring over Rational Field
sage: C = A.Bases(); C
Category of bases of Integer-Valued Polynomial Ring over Rational Field
sage: C.super_categories()
[Category of realizations of Integer-Valued Polynomial Ring over Rational Field,
 Join of Category of algebras with basis over Rational Field and
 Category of filtered algebras over Rational Field and
 Category of commutative algebras over Rational Field and
 Category of realizations of unital magmas]
```

```python
class Binomial(A):
    Bases: CombinatorialFreeModule, BindableClass

The integer-valued polynomial ring in the binomial basis.

The basis used here is given by \( B[i] = \binom{n}{i} \) for \( i \in \mathbb{N} \).

Assuming \( n_1 \leq n_2 \), the product of two monomials \( B[n_1] \cdot B[n_2] \) is given by the sum

\[
\sum_{k=0}^{n_1} \binom{n_1}{k} \binom{n_1 + n_2 - k}{n_1} B[n_1 + n_2 - k].
\]

The product of two monomials is therefore a positive linear combination of monomials.

EXAMPLES:

```python
sage: F = IntegerValuedPolynomialRing(QQ).B(); F
Integer-Valued Polynomial Ring over Rational Field in the binomial basis
sage: F.gen()
B[1]

sage: S = IntegerValuedPolynomialRing(ZZ).B(); S
Integer-Valued Polynomial Ring over Integer Ring in the binomial basis
sage: S.base_ring()
Integer Ring

sage: G = IntegerValuedPolynomialRing(S).B(); G
Integer-Valued Polynomial Ring over Integer-Valued Polynomial Ring over Integer Ring in the binomial basis in the binomial basis
sage: G.base_ring()
Integer-Valued Polynomial Ring over Integer Ring in the binomial basis
```

Integer-valued polynomial rings commute with their base ring:

```python
sage: K = IntegerValuedPolynomialRing(QQ).B()
sage: a = K.gen()
sage: K.is_commutative()
```

(continues on next page)
Polynomials, Release 10.1

(continued from previous page)

True

```python
sage: L = IntegerValuedPolynomialRing(K).B()
sage: c = L.gen()
sage: L.is_commutative()
True
sage: s = a * c^3; s
sage: parent(s)
Integer-Valued Polynomial Ring over Integer-Valued Polynomial Ring over Rational Field in the binomial basis in the binomial basis
```

Integer-valued polynomial rings are commutative:

```python
sage: c^3 * a == c * a * c * c
True
```

We can also manipulate elements in the basis:

```python
sage: F = IntegerValuedPolynomialRing(QQ).B()
sage: B = F.basis()
```

and coerce elements from our base field:

```python
sage: F(4/3)
4/3*B[0]
```

class `Element`

Bases: `IndexedFreeModuleElement`

`product_on_basis(n1, n2)`

Return the product of basis elements n1 and n2.

INPUT:

• n1, n2 – integers

EXAMPLES:

```python
sage: A = IntegerValuedPolynomialRing(QQ).B()
sage: A.product_on_basis(0, 1)
B[1]
sage: A.product_on_basis(1, 2)
```

S

alias of `Shifted`

class `Shifted`(`A`)

Bases: `CombinatorialFreeModule`, `BindableClass`

The integer-valued polynomial ring in the shifted basis.

The basis used here is given by $S[i] = (i+x)^{i}$ for $i \in \mathbb{N}$. 272 Chapter 2. Univariate Polynomials
Assuming \( n_1 \leq n_2 \), the product of two monomials \( S[n_1] \cdot S[n_2] \) is given by the sum

\[
\sum_{k=0}^{n_1} (-1)^k \binom{n_1}{k} \binom{n_1 + n_2 - k}{n_1} S[n_1 + n_2 - k].
\]

**EXAMPLES:**

```python
sage: F = IntegerValuedPolynomialRing(QQ).S(); F
Integer-Valued Polynomial Ring over Rational Field
in the shifted basis

sage: F.parent()
Integer-Valued Polynomial Ring over Rational Field
in the shifted basis

sage: F.basis()
(S[1],)

sage: G = IntegerValuedPolynomialRing(S).S(); G
Integer-Valued Polynomial Ring over Integer-Valued Polynomial
Ring over Integer-Valued Polynomial Ring over Integer Ring
in the shifted basis in the shifted basis

sage: G.base_ring()
Integer-Valued Polynomial Ring over Integer Ring
in the shifted basis

Integer-valued polynomial rings commute with their base ring:

```python
sage: K = IntegerValuedPolynomialRing(QQ).S();
```

```python
sage: a = K.gen()
```

```python
sage: K.is_commutative()
True
```

```python
sage: L = IntegerValuedPolynomialRing(K).S();
```

```python
sage: c = L.gen()
```

```python
sage: L.is_commutative()
True
```

```python
sage: s = a * c^3; s
```

```python
sage: parent(s)
Integer-Valued Polynomial Ring over Integer-Valued Polynomial
Ring over Rational Field in the shifted basis in the shifted basis
```

Integer-valued polynomial rings are commutative:

```python
sage: c^3 * a == c * a * c * c
True
```

We can also manipulate elements in the basis and coerce elements from our base field:

```python
sage: F = IntegerValuedPolynomialRing(QQ).S();
```

```python
sage: S = F.basis();
```

```python
```

(continues on next page)
class Element

    Bases: IndexedFreeModuleElement

    delta()

    Return the image by the difference operator \( \Delta \).
    The operator \( \Delta \) is defined on polynomials by
    \[
    f \mapsto f(x + 1) - f(x).
    \]

    EXAMPLES:

    sage: F = IntegerValuedPolynomialRing(ZZ).S()
    sage: S = F.basis()
    sage: S[5].delta()

    derivative_at_minus_one()

    Return the derivative at \(-1\).
    This is sometimes useful when \(-1\) is a root.

    See also:
    umbra()

    EXAMPLES:

    sage: F = IntegerValuedPolynomialRing(ZZ).S()
    sage: B = F.gen()
    sage: (B+1).derivative_at_minus_one()
    1

    h_polynomial()

    Return the \(h\)-vector as a polynomial.

    See also:
    h_vector()

    EXAMPLES:

    sage: x = polygen(QQ,'x')
    sage: A = IntegerValuedPolynomialRing(ZZ).S()
    sage: ex = A.from_polynomial((1+x)**3)
    sage: ex.h_polynomial()
    z^3 + 4*z^2 + z

    h_vector()

    Return the numerator of the generating series of values.
    If self is an Ehrhart polynomial, this is the \(h\)-vector.

    See also:
    h_polynomial()
EXAMPLES:

```
sage: x = polygen(QQ, 'x')
sage: A = IntegerValuedPolynomialRing(ZZ).S()
sage: ex = A.from_polynomial((1+x)**3)
sage: ex.h_vector()
(0, 1, 4, 1)
```

**umbra()**

Return the Bernoulli umbra.

This is the derivative at $-1$ of the shift by one.

See also:

`derivative_at_minus_one()`

EXAMPLES:

```
sage: F = IntegerValuedPolynomialRing(ZZ).S()
sage: B = F.gen()
sage: (B+1).umbra()
3/2
```

**variable_shift($k=1$)**

Return the image by the shift of variables.

On polynomials, the action is the shift on variables $x \mapsto x + k$.

INPUT:

- $k$ – integer (default: 1)

EXAMPLES:

```
sage: A = IntegerValuedPolynomialRing(ZZ).S()
sage: S = A.basis()
sage: S[5].variable_shift()
sage: S[5].variable_shift(-1)
```

**from_h_vector($h$)**

Convert from some $h$-vector.

INPUT:

- $h$ – a tuple or vector

See also:

`Element.h_vector()`

EXAMPLES:

```
sage: A = IntegerValuedPolynomialRing(ZZ).S()
sage: S = A.basis()
sage: A.from_h_vector(ex.h_vector())
```
product_on_basis(n1, n2)

Return the product of basis elements n1 and n2.

INPUT:
  • n1, n2 – integers

EXAMPLES:

```python
sage: A = IntegerValuedPolynomialRing(QQ).S()
sage: A.product_on_basis(0, 1)
S[1]
sage: A.product_on_basis(1, 2)
```

a_realization()

Return a default realization.

The Binomial realization is chosen.

EXAMPLES:

```python
sage: IntegerValuedPolynomialRing(QQ).a_realization()
Integer-Valued Polynomial Ring over Rational Field
in the binomial basis
```

## 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 multiplication 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.convolution(L1, L2)

Return convolution of non-empty lists L1 and L2.

L1 and L2 may have arbitrary lengths.

EXAMPLES:

```
sage: convolution([1, 2, 3], [4, 5, 6, 7])
[4, 13, 28, 34, 32, 21]
```

### 2.3 Fast calculation of cyclotomic polynomials

This module provides a function `cyclo\textit{t}	extit{\textit{t}}\textit{\textit{i}	extit{c}	extit{t}	extit{i}	extit{i}	extit{m}	extit{e}	extit{c}	extit{o}	extit{e}	extit{f}	extit{s}()`, 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 `cyclo\textit{t}	extit{i}	extit{c}	extit{t}	extit{i}	extit{o}	extit{m}	extit{e}	extit{c}	extit{y}	extit{c}	extit{t}	extit{o}	extit{m}	extit{i}	extit{i}	extit{n}	extit{e}	extit{a}	extit{l}()\` method of univariate polynomial ring objects and the top-level `cyclo\textit{t}	extit{i}	extit{c}	extit{t}	extit{i}	extit{c}	extit{y}	extit{c}	extit{t}	extit{o}	extit{m}	extit{i}	extit{i}	extit{n}()\` function.

sage.rings.polynomial.cyclotomic.bateman_bound(nn)

Reference:

Bateman, P. T.; Pomerance, C.; Vaughan, R. C. \textit{On the size of the coefficients of the cyclotomic polynomial}. 

EXAMPLES:

```
sage: from sage.rings.polynomial.cyclotomic import bateman_bound
sage: bateman_bound(2**8 * 1234567893377)  # needs sage.libs.pari
66944986927
```

sage.rings.polynomial.cyclotomic.cyclo\textit{t}	extit{i}	extit{c}	extit{t}	extit{i}	extit{m}	extit{e}	extit{c}	extit{o}	extit{e}	extit{f}	extit{s}(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 `sparse` is True, the result is returned as a dictionary of the non-zero entries, otherwise the result is returned as a list of python ints.

EXAMPLES:

```
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:
The coefficients are not always +/-1:

```sage
cyclotomic_coeffs(105)
[1, 1, 1, 0, 0, -1, -1, -2, -1, -1, 0, 1, 1, 1, 1, 0, 0, -1, 0, -1, 0, -1, 0, 1, 1, 1, 1, 0, 0, 1, 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:

```sage
v = cyclotomic_coeffs(1181895)
max(v)
```

14102773

The polynomial is a palindrome for any \( n \):

```sage
n = ZZ.random_element(50000)
v = cyclotomic_coeffs(n, sparse=False)
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/

```
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)
sage: cyclotomic_polynomial(51)(3)
```

It works for non-integral values as well:

```
sage: cyclotomic_value(144, 4/3)
sage: cyclotomic_polynomial(144)(4/3)
```
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

\[ \mathbb{F}_2[x_1, ..., x_n]/\langle x_1^2 + x_1, ..., x_n^2 + x_n \rangle. \]

are implemented using the PolyBoRi library (cf. `sage.rings.polynomial.pbori`).

3.1.1 Term orders

Sage supports the following term orders:

**Lexicographic (lex)**

\[ a^x < b^x \text{ if and only if there exists } 1 \leq i \leq n \text{ such that } a_1 = b_1, \ldots, a_{i-1} = b_{i-1}, a_i < b_i. \]
This term order is called ‘lp’ in Singular.

**EXAMPLES:**

```python
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:**
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:

<table>
<thead>
<tr>
<th>sage:</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sage: P.&lt;x,y,z&gt; = PolynomialRing(QQ, 3, order='deglex')</td>
<td></td>
</tr>
<tr>
<td>sage: x &gt; y</td>
<td>True</td>
</tr>
<tr>
<td>sage: x &gt; y^2*z</td>
<td>False</td>
</tr>
<tr>
<td>sage: x &gt; 1</td>
<td>True</td>
</tr>
<tr>
<td>sage: x^1<em>y^5</em>z^2 &gt; x^4<em>y^1</em>z^3</td>
<td>True</td>
</tr>
<tr>
<td>sage: x^2<em>y</em>z^2 &gt; x<em>y^3</em>z</td>
<td>False</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>sage:</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sage: P.&lt;x,y,z&gt; = PolynomialRing(QQ, 3, order='invlex')</td>
<td></td>
</tr>
<tr>
<td>sage: x &gt; y</td>
<td>False</td>
</tr>
<tr>
<td>sage: y &gt; x^2</td>
<td>True</td>
</tr>
<tr>
<td>sage: x &gt; 1</td>
<td>True</td>
</tr>
<tr>
<td>sage: x^y &gt; z</td>
<td>False</td>
</tr>
</tbody>
</table>

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:
Polynomials, Release 10.1

```python
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order='neglex')
sage: x > y
False
sage: x > 1
False
sage: x^1*y^2 > y^3*z^4
False
sage: x^3*y^2*z^4 < x^3*y^2*z^1
True
sage: x*y > z
False
```

**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='wdegrevlex')
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
```

3.1. Multivariate Polynomials and Polynomial Rings 283
Weighted degree lexicographic (wdeglex), 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 ‘Wp’ in Singular.

EXAMPLES:

\[
\text{sage: } P.<x,y,z> = PolynomialRing(QQ, 3, order=TermOrder('wdegrevlex',(1,2,3)))
\text{sage: } x > y
\text{False}
\text{sage: } x > x^2
\text{False}
\text{sage: } x > 1
\text{True}
\text{sage: } x^1*y^2 > x^2*z
\text{True}
\text{sage: } y*z > x^3*y
\text{False}
\]

Negative weighted degree reverse lexicographic (negwdegrevlex), 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 ‘ws’ in Singular.

EXAMPLES:

\[
\text{sage: } P.<x,y,z> = PolynomialRing(QQ, 3, order=TermOrder('negwdegrevlex',(1,2,3)))
\text{sage: } x > y
\text{True}
\text{sage: } x > x^2
\text{True}
\text{sage: } x > 1
\text{False}
\text{sage: } x^1*y^2 > x^2*z
\text{False}
\text{sage: } y*z > x^3*y
\text{False}
\]

Degree negative lexicographic (degneglex)

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_asc’ in PolyBoRi. Singular has the extra weight vector ordering \((a(1:n), 1s)\) for this purpose.
EXAMPLES:

```python
sage: t = TermOrder('degneglex')
sage: P.<x,y,z> = PolynomialRing(QQ, order=t)
sage: x*y > y*z # indirect doctest
False
sage: x*y > x
True
```

Negative weighted degree lexicographic (negwdeglex), positive integral weights

Let \( \deg_w(x^a) = a_1w_1 + a_2w_2 + \cdots + a_nw_n \) 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:

```python
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:

```python
sage: # needs sage.modules
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
```

3.1. Multivariate Polynomials and Polynomial Rings

285
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
sage: T.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
class sage.rings.polynomial.term_order.TermOrder(name='lex', n=0, force=False)

Bases: 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 github issue #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)
```

property 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
# needs sage.rings.number_field
c^4
sage: g = a + e^4; g.lm() # needs sage.rings.number_field
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
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order='deglex')
# needs sage.rings.number_field
sage: f = x + y; f.lm() # indirect doctest
# needs sage.rings.number_field
x
sage: f = x + y^2*z; f.lm()
# needs sage.rings.number_field
y^2*z
```

This method is called by the \(lm/lc/lt\) methods of \texttt{MPolynomial\_polydict}.

greater_tuple_degneglex \((f, g)\)

Return the greater exponent tuple with respect to the degree negative lexicographical term order.

**INPUT:**

- \(f\) - exponent tuple
- \(g\) - exponent tuple

**EXAMPLES:**

```sage
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order='degneglex')
# needs sage.rings.number_field
sage: f = x + y; f.lm() # indirect doctest
# needs sage.rings.number_field
y
sage: f = x + y^2*z; f.lm()
# needs sage.rings.number_field
y^2*z
```

This method is called by the \(lm/lc/lt\) methods of \texttt{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
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order='degrevlex')
# needs sage.rings.number_field
sage: f = x + y; f.lm() # indirect doctest
# needs sage.rings.number_field
x
```

(continues on next page)
This method is called by the lm/lc/lt methods of \texttt{MPolynomial\_polydict}.

**\texttt{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')  # needs sage.rings.number_field
sage: f = x + y; f.lm() # indirect doctest  # needs sage.rings.number_field
y
sage: f = y + x^2; f.lm()  # needs sage.rings.number_field
y
```

This method is called by the lm/lc/lt methods of \texttt{MPolynomial\_polydict}.

**\texttt{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')  # needs sage.rings.number_field
sage: f = x + y^2; f.lm() # indirect doctest  # needs sage.rings.number_field
x
```

This method is called by the lm/lc/lt methods of \texttt{MPolynomial\_polydict}.

**\texttt{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:**

```
``
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: # needs sage.rings.number_field
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/lt methods of \texttt{MPolynomialPolydict}.

greater_tuple_negdegrevlex\((f, g)\)

Return the greater exponent tuple with respect to the negative degree reverse lexicographical term order.

**INPUT:**

- \(f\) - exponent tuple
- \(g\) - exponent tuple

**EXAMPLES:**

```
sage: # needs sage.rings.number_field
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
```

This method is called by the \lm/lc/lt methods of \texttt{MPolynomialPolydict}.

greater_tuple_neglex\((f, g)\)

Return the greater exponent tuple with respect to the negative lexicographical term order.

This method is called by the \lm/lc/lt methods of \texttt{MPolynomialPolydict}.
• \( f \) - exponent tuple
• \( g \) - exponent tuple

EXAMPLES:

```python
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
    ....: needs sage.rings.number_field
c^4
sage: g = a + e^4; g.lm() # indirect doctest
    ....: needs sage.rings.number_field
a
greater_tuple_negwdeglex(f, g)

Return the greater exponent tuple with respect to the negative weighted degree lexicographical term order.

INPUT:
• \( f \) - exponent tuple
• \( g \) - exponent tuple

EXAMPLES:

```python
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
    ....: needs sage.rings.number_field
x
sage: f = x + x^2; f.lm()
    ....: needs sage.rings.number_field
x
sage: f = x^3 + z; f.lm()
    ....: needs sage.rings.number_field
x^3
```

This method is called by the \texttt{lm/lc/lt} methods of \texttt{MPolynomial_polydict}.

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:

```python
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
    ....: needs sage.rings.number_field
x
sage: f = x + x^2; f.lm()
    ....: needs sage.rings.number_field
x
```
This method is called by the lm/lc/lt methods of `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:**

```python
sage: t = TermOrder('wdeglex', (1, 2, 3))
sage: P.<x, y, z> = PolynomialRing(QQbar, 3, order=t)  # indirect doctest
sage: f = x + y; f.lm()  # indirect doctest
```

This method is called by the lm/lc/lt methods of `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:**

```python
sage: t = TermOrder('wdegrevlex', (1, 2, 3))
sage: P.<x, y, z> = PolynomialRing(QQbar, 3, order=t)  # indirect doctest
sage: f = x + y^2*z; f.lm()  # indirect doctest
```

This method is called by the lm/lc/lt methods of `MPolynomial_polydict`.

`is_block_order()`

Return `True` if `self` is a block term order.

**EXAMPLES:**
```python
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:**
```python
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
sage: T = TermOrder('invlex', 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:**
```python
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:**
```python
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 }
```

```python
magma_str()
```

Return a MAGMA representation of self.

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
```

```python
sage: T = P.term_order()
sage: T.magma_str()
'"grevlex"'
```

```python
matrix()
```

Return the matrix defining matrix term order.

EXAMPLES:

```python
sage: t = TermOrder("M(1,2,0,1)")  # needs sage.modules
sage: t.matrix()  # needs sage.modules
[1 2]
[0 1]
```

```python
name()
```

EXAMPLES:

```python
sage: TermOrder('lex').name()
'lex'
```

```python
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)')
```

(continues on next page)
### 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:**

```python
sage: P = PolynomialRing(GF(127), 10, names='x',
  .....:    order='lex(3),degneglex(5),lex(2)')
```

```python
sage: T = P.term_order()
```

```python
sage: T.singular_str()  
'(lp(3),Dp(5),lp(2))
```

```python
sage: P._singular_()  
˓→ needs sage.libs.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:

```python
sage: T = TermOrder("degneglex", 2)
```

```python
sage: P = PolynomialRing(QQ,2, names='x', order=T)
```

```python
sage: T = P.term_order()
```

```python
sage: T.singular_str()  
'(a(1:2),ls(2))'
```

```python
sage: T = TermOrder("degneglex", 2) + TermOrder("degneglex", 2)
```

```python
sage: P = PolynomialRing(QQ,4, names='x', order=T)
```

```python
sage: T = P.term_order()
```

```python
sage: T.singular_str()  
'(a(1:2),ls(2),a(1:2),ls(2))'
```

```python
sage: P._singular_()  
˓→ needs sage.libs.singular
```

(continues on next page)
The position of the ordering \texttt{C} block can be controlled by setting \texttt{_singular_ringorder_column} attribute to an integer:

```python
sage: T = TermOrder("degrevlex", 2) + TermOrder("degrevlex", 2)
sage: T._singular_ringorder_column = 0
sage: P = PolynomialRing(QQ, 4, names='x', order=T)
sage: P._singular_()
```

```plaintext
needs sage.libs.singular polynomial ring, over a field, global ordering
// coefficients: QQ
// number of vars : 4
// block 1 : ordering a
// : names x0 x1
// : weights 1 1
// block 2 : ordering ls
// : names x0 x1
// block 3 : ordering a
// : names x2 x3
// : weights 1 1
// block 4 : ordering ls
// : names x2 x3
// block 5 : ordering C
```

```python
sage: T._singular_ringorder_column = 1
sage: P = PolynomialRing(QQ, 4, names='y', order=T)
sage: P._singular_()
```

```plaintext
needs sage.libs.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
```

(continues on next page)
Polynomials, Release 10.1

(continued from previous page)

```plaintext
// block 4 : ordering a
// : names y2 y3
// : weights 1 1
// block 5 : ordering ls
// : names y2 y3

sage: T._singular_ringorder_column = 2
sage: P = PolynomialRing(QQ, 4, names='z', order=T)
sage: P._singular_()

# needs sage.libs.singular

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
```

property sortkey

The default sortkey method for this term order.

EXAMPLES:

```plaintext
sage: 0 = TermOrder()

sage: 0.sortkey.__func__ is 0.sortkey_lex.__func__
True
sage: 0 = TermOrder('deglex')

sage: 0.sortkey.__func__ is 0.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:

```plaintext
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
```

(continues on next page)
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') # needs sage.rings.number_field
sage: x > y^2 # indirect doctest # needs sage.rings.number_field
False
sage: x > 1 # needs sage.rings.number_field
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') # needs sage.rings.number_field
sage: x*y > y*z # indirect doctest # needs sage.rings.number_field
False
sage: x*y > x # needs sage.rings.number_field
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') # needs sage.rings.number_field
sage: x > y^2 # indirect doctest # needs sage.rings.number_field
False
sage: x > 1 # needs sage.rings.number_field
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') # needs sage.rings.number_field
sage: x > y^2 # indirect doctest # needs sage.rings.number_field
False
sage: x > 1 # needs sage.rings.number_field
True
```

sortkey_lex(f)
Return the sortkey of an exponent tuple with respect to the lexicographical term order.

INPUT:
• f – exponent tuple

EXAMPLES:

```
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='lex') # needs sage.rings.number_field
sage: x > y^2 # indirect doctest # needs sage.rings.number_field
True
sage: x > 1 # needs sage.rings.number_field
True
```

sortkey_matrix(f)
Return the sortkey of an exponent tuple with respect to the matrix term order.

INPUT:
• f – exponent tuple

EXAMPLES:

```
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='m(1,3,1,0)') # needs sage.rings.number_field
sage: y > x^2 # indirect doctest # needs sage.rings.number_field
True
sage: y > x^3 # needs sage.rings.number_field
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:**

```python
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='negdeglex')  # needs sage.rings.number_field
sage: x > y^2 # indirect doctest  # needs sage.rings.number_field
True
sage: x > 1  # needs sage.rings.number_field
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:**

```python
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='negdegrevlex')  # needs sage.rings.number_field
sage: x > y^2 # indirect doctest  # needs sage.rings.number_field
True
sage: x > 1  # needs sage.rings.number_field
False
```

### sortkey_neglex(\( f \))

Return the sortkey of an exponent tuple with respect to the negative lexicographical term order.

**INPUT:**

• \( f \) – exponent tuple

**EXAMPLES:**

```python
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='neglex')  # needs sage.rings.number_field
sage: x > y^2 # indirect doctest  # needs sage.rings.number_field
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:**

```python
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='negwdeglex')  # needs sage.rings.number_field
sage: x > y^2 # indirect doctest  # needs sage.rings.number_field
False
```

---

Chapter 3. Multivariate Polynomials
```
sage: t = TermOrder('negwdeglex', (3,2))
sage: P.<x,y> = PolynomialRing(QQbar, 2, order=t)  # needs sage.rings.number_field
sage: x > y^2  # indirect doctest  # needs sage.rings.number_field
True
sage: x^2 > y^3  # needs sage.rings.number_field
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)  # needs sage.rings.number_field
sage: x > y^2  # indirect doctest  # needs sage.rings.number_field
True
sage: x^2 > y^3  # needs sage.rings.number_field
True
```

```
sage: t = TermOrder('wdeglex', (3,2))
sage: P.<x,y> = PolynomialRing(QQbar, 2, order=t)  # needs sage.rings.number_field
sage: x > y^2  # indirect doctest  # needs sage.rings.number_field
False
sage: x > y  # needs sage.rings.number_field
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)  # needs sage.rings.number_field
sage: x > y^2  # indirect doctest  # needs sage.rings.number_field
False
sage: x > y  # needs sage.rings.number_field
True
```

### 3.1. Multivariate Polynomials and Polynomial Rings

301
```
sage: t = TermOrder('wdegrevlex',(3,2))
sage: P.<x,y> = PolynomialRing(QQbar, 2, order=t)  # needs sage.rings.number_field
sage: x > y^2 # indirect doctest  # needs sage.rings.number_field
False
sage: x^2 > y^3  # needs sage.rings.number_field
True

tuple_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)  # needs sage.rings.number_field
sage: P.term_order().tuple_weight([3,2,1])  # needs sage.rings.number_field
10

tuple_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)  # needs sage.rings.number_field
sage: P.term_order().tuple_weight([3,2,1])  # needs sage.rings.number_field
10
```
weights()
Return the weights for weighted term orders.

EXAMPLES:
```
sage: t = TermOrder('wdeglex',(2,3))
sage: t.weights()
(2, 3)
```
sage.rings.polynomial.term_order.termorder_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 termorder_from_singular
sage: singular.eval('ring rl = (9,x),(a,b,c,d,e,f),(M((1,2,3,0)),wp(2,3),lp)')  # needs sage.libs.singular
sage: termorder_from_singular(singular)  # needs sage.libs.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 
_singular_ringorder_column attribute of the term order.

```sage
# needs sage.libs.singular
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
# needs sage.libs.singular
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

```sage
class sage.rings.polynomial.multi_polynomial_ring_base.BooleanPolynomialRing_base
Bases: MPolynomialRing_base

Abstract base class for BooleanPolynomialRing.

This class is defined for the purpose of isinstance tests. It should not be instantiated.

EXAMPLES:
```

```sage
from sage.rings.polynomial.multi_polynomial_ring_base import, BooleanPolynomialRing_base
sage: R.<x, y, z> = BooleanPolynomialRing()
```

(continues on next page)
By design, there is only one direct implementation subclass:

```python
sage: len(BooleanPolynomialRing_base.__subclasses__()) <= 1
True
```

```python
class sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base
Bases: CommutativeRing

Create a polynomial ring in several variables over a commutative ring.

EXAMPLES:

```python
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,self)
    ....:     def __call__(self,x):
    ....:         return None
sage: cr = CR()
sage: cr.is_commutative()
True
sage: cr['x,y']
Multivariate Polynomial Ring in x, y over
<__main__.CR_with_category object at ...>
```

```python
change_ring(base_ring=None, names=None, order=None)
Return a new multivariate polynomial ring which is 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:

```python
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
```

```python
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=None, prec=None, extras=None, **kwds)

Return the completion of self with respect to the ideal generated by the variable(s) names.

INPUT:

* names – (optional) variable or list/tuple of variables (given either as elements of the polynomial ring or as strings); the default is all variables of self
* prec – default precision of resulting power series ring, possibly infinite
* extras – passed as keywords to PowerSeriesRing or LazyPowerSeriesRing; can also be keyword arguments

EXAMPLES:

sage: P.<x,y,z,w> = PolynomialRing(ZZ)
sage: P.completion()
Power Series Ring in w over Multivariate Polynomial Ring in x, y, z over Integer Ring
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
sage: Q.<w,x,y,z> = P.completion(); Q
Multivariate Power Series Ring in w, x, y, z over Integer Ring
sage: H = PolynomialRing(PolynomialRing(ZZ,3,'z'),4,'f'); H
Multivariate Polynomial Ring in f0, f1, f2, f3 over Multivariate Polynomial Ring in z0, z1, z2 over Integer Ring
sage: H.completion(H.gens())
Multivariate Power Series Ring in f0, f1, f2, f3 over Multivariate Polynomial Ring in z0, z1, z2 over Integer Ring
sage: H.completion(H.gens()[2])
Power Series Ring in f2 over Multivariate Polynomial Ring in f0, f1, f3 over Multivariate Polynomial Ring in z0, z1, z2 over Integer Ring
sage: P.<x,y,z,w> = PolynomialRing(ZZ)
sage: P.completion(prec=oo) # needs sage.combinat
Multivariate Lazy Taylor Series Ring in x, y, z, w over Integer Ring
sage: P.completion((w,x,y), prec=oo) # needs sage.combinat
Multivariate Lazy Taylor Series Ring in w, x, y over Univariate Polynomial Ring in z over Integer Ring

construction()

Returns a functor F and base ring R such that F(R) == self.

3.1. Multivariate Polynomials and Polynomial Rings 305
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)**

**interpolation(bound, *args)**

Create a polynomial with specified evaluations.

CALL FORMATS:

This function can be called in two ways:
1. interpolation(bound, points, values)
2. interpolation(bound, function)

INPUT:

* bound – either an integer bounding the total degree or a list/tuple of integers bounding the degree of the variables
* points – list/tuple containing the evaluation points
* values – list/tuple containing the desired values at points
* function – evaluable function in \( n \) variables, where \( n \) is the number of variables of the polynomial ring

OUTPUT:

1. A polynomial respecting the bounds and having \( \text{values} \) as values when evaluated at \( \text{points} \).
2. A polynomial respecting the bounds and having the same values as \( \text{function} \) at exactly so many points so that the polynomial is unique.

EXAMPLES:
ALGORITHM:

Solves a linear system of equations with the linear algebra module. If the points are not specified, it samples exactly as many points as needed for a unique solution.

Note: It will only run if the base ring is a field, even though it might work otherwise as well. If your base ring is an integral domain, let it run over the fraction field.

Also, if the solution is not unique, it spits out one solution, without any notice that there are more.

Lastly, the interpolation function for univariate polynomial rings is called \texttt{lagrange_polynomial()}. 

\textbf{Warning:} If you don’t provide point/value pairs but just a function, it will only use as many points as needed for a unique solution with the given bounds. In particular it will not notice or check whether the result yields the correct evaluation for other points as well. So if you give wrong bounds, you will get a wrong answer without any warning.
See also:

**lagrange_polynomial**

**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[]
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:

```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:

```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:

```sage
ZZ['x,y'].is_integral_domain()
```

True

```sage
Integers(8)['x,y'].is_integral_domain()
```

False

**is_noetherian()**

EXAMPLES:

```sage
ZZ['x,y'].is_noetherian()
```

True

```sage
Integers(8)['x,y'].is_noetherian()
```

True
**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, the 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:

```python
sage: R.<x,y,z> = PolynomialRing(QQ, 3)
sage: R.macaulay_resultant([y, x + z])
# needs sage.modules
```

Traceback (most recent call last):
...
TypeError: number of polynomials(= 2) must equal number of variables (= 3)

The polynomials need to be all homogeneous:

```python
sage: R.<x,y,z> = PolynomialRing(QQ, 3)
sage: R.macaulay_resultant([y, x + z, z + x^3])
# needs sage.modules
```

Traceback (most recent call last):
...
TypeError: resultant for non-homogeneous polynomials is not supported

All polynomials must be in the same ring:
Polynomials, Release 10.1

```python
sage: S.<x,y> = PolynomialRing(QQ, 2)
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: S.macaulay_resultant([y, z+x])
# needs sage.modules
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]:

```python
sage: K.<x,y> = PolynomialRing(ZZ, 2)
sage: flist, R = K._macaulay_resultant_universal_polynomials([1,1,2])
sage: R.macaulay_resultant(flist)
# needs sage.modules
u2^2*u4^2*u6^2 - 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*u2*u3*u5*u9 + u0^2*u5^2*u9 - u1*u2*u3^2*u10 + u0*u2*u3*u4*u10 + u0*u1*u3*u5*u10 - u0*u2*u4*u5*u10 +
u1*u2*u3^2*u11 - 2*u0*u1*u3*u4*u11 + u0*u2*u4^2*u11
```

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: R.macaulay_resultant(flist)
# needs sage.modules
-u2*u4*u6^2 + 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])
# needs sage.modules
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])
# needs sage.modules
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])
# needs sage.modules
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])
# needs sage.libs.pari sage.modules
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)
# needs sage.libs.pari sage.modules
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])
# needs sage.modules sage.rings.finite_rings
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):

```python
sage: R.<x,y,z> = PolynomialRing(QQ, 3)
sage: R.macaulay_resultant([y, x + z, z^2])
# needs sage.libs.pari sage.modules
-1
```

When there are only 2 polynomials, the Macaulay resultant degenerates to the traditional resultant:

```python
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])
# needs sage.modules
True
```

```
monomial(*exponents)
```

Return the monomial with given exponents.

**EXAMPLES:**

```python
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
```
monomials_of_degree(degree)
Return a list of all monomials of the given total degree in this multivariate polynomial ring.

EXAMPLES:

```python
sage: # needs sage.combinat
sage: R.<x,y,z> = ZZ[]
sage: mons = R.monomials_of_degree(2)
sage: mons
[z^2, y*z, x*z, y^2, x*y, x^2]
```

The number of such monomials equals $\binom{n+k-1}{k}$ where $n$ is the number of variables and $k$ the degree:

```python
sage: len(mons) == binomial(3 + 2 - 1, 2)  # needs sage.combinat
True
```

ngens()

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.

INPUT:

- degree – maximal degree (likely to be reached) (default: 2)
- 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 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: f = P.random_element(2, 5)
sage: f.degree() <= 2
True
```
sage: f.parent() is P
True
sage: len(list(f)) <= 5
True
sage: f = P.random_element(2, 5, choose_degree=True)
sage: f.degree() <= 2
True
sage: f.parent() is P
True
sage: len(list(f)) <= 5
True

Stacked rings:

sage: R = QQ['x,y']
sage: S = R['t,u']
sage: f = S._random_nonzero_element(degree=2, terms=1)
sage: len(list(f))
1
sage: f.degree() <= 2
True
sage: f.parent() is S
True

Default values apply if no degree and/or number of terms is provided:

sage: # needs sage.modules
sage: M = random_matrix(QQ['x,y,z'], 2, 2)
sage: all(a.degree() <= 2 for a in M.list())
True
sage: all(len(list(a)) <= 5 for a in M.list())
True
sage: M = random_matrix(QQ['x,y,z'], 2, 2, terms=1, degree=2)
sage: all(a.degree() <= 2 for a in M.list())
True
sage: all(len(list(a)) <= 1 for a in M.list())
True
sage: P.random_element(0, 1) in QQ
True
sage: P.random_element(2, 0)
0
sage: R.<x> = PolynomialRing(Integers(3), 1)
sage: f = R.random_element()
sage: f.degree() <= 2
True
sage: len(list(f)) <= 3
True

To produce a dense polynomial, pick terms=Infinity:
sage: P.<x,y,z> = GF(127)[]
sage: f = P.random_element(degree=2, terms=Infinity)
sage: while len(list(f)) != 10:
    ....: f = P.random_element(degree=2, terms=Infinity)
sage: f = P.random_element(degree=3, terms=Infinity)
sage: while len(list(f)) != 20:
    ....: f = P.random_element(degree=3, terms=Infinity)
sage: f = P.random_element(degree=3, terms=Infinity, choose_degree=True)

sage: while len(list(f)) != 20:
    ....: f = P.random_element(degree=3, terms=Infinity)

The number of terms is silently reduced to the maximum available if more terms are requested:

sage: P.<x,y,z> = GF(127)[]
sage: f = P.random_element(degree=2, terms=1000)
sage: len(list(f)) <= 10
True

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

```python
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:**

```python
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

**some_elements()**

Return a list of polynomials.

This is typically used for running generic tests.

**EXAMPLES:**

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

**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:**

```python
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:**
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

3.1.3 Base class for elements of multivariate polynomial rings

class sage.rings.polynomial.multi_polynomial.MPolynomial

Bases: CommutativePolynomial

args()

Returns the names 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')[]
# needs sage.rings.finite_rings
sage: (x+2*y).change_ring(GF(3))
x - y

sage: # needs sage.rings.number_field
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 - 0.866025403784438*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()
[23, 6, 1]
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 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()

Return the content of this polynomial. Here, we define content as the gcd of the coefficients in the base ring.
See also:

\texttt{content\_ideal()}

EXAMPLES:

\begin{verbatim}
sage: R.<x,y> = ZZ[]
sage: f = 4*x+6*y
sage: f.content()
2
sage: f.content().parent()
Integer Ring
\end{verbatim}

\texttt{content\_ideal()}

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

See also:

\texttt{content()}

EXAMPLES:

\begin{verbatim}
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
\end{verbatim}

\texttt{denominator()}

Return a denominator of \texttt{self}.

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

Note that some subclasses may implement its own denominator function.

\textbf{Warning:} This is not the denominator of the rational function defined by \texttt{self}, which would always be 1 since \texttt{self} is a polynomial.

EXAMPLES:

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

\begin{verbatim}
sage: R.<x,y> = ZZ[]
sage: f = x^3 + 17*y + x + y
sage: f.denominator()
1
\end{verbatim}

Next we compute the denominator of a polynomial over a number field.

\begin{verbatim}
sage: # needs sage.rings.number_field sage.symbolic
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
\end{verbatim}

(continues on next page)
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[]
# needs sage.rings.real_mpfr
sage: f = a + b + RR('0.3'); f
# needs sage.rings.real_mpfr
a + b + 0.300000000000000
sage: f.denominator()                                     # needs sage.rings.real_mpfr
1.00000000000000
```

Check that the denominator is an element over the base whenever the base has no denominator function. This closes github issue #9063:

```
sage: R.<a,b,c> = GF(5)[]
sage: x = R(0)
sage: x.denominator()
1
sage: type(x.denominator())                               <class 'sage.rings.finite_rings.integer_mod.IntegerMod_int'>
sage: type(a.denominator())                               <class 'sage.rings.finite_rings.integer_mod.IntegerMod_int'>
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 function derivative() for more details.

See also:

_derivative()

EXAMPLES:

Polynomials implemented via Singular:

```
sage: # needs sage.libs.singular
sage: R.<x, y> = PolynomialRing(FiniteField(5))
sage: f = x^3*y^5 + x^7*y
sage: type(f)<class 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular'>
sage: f.derivative(x)
2*x^6*y - 2*x^2*y^5
```

(continues on next page)
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: R.<x,y,z> = QQ[]
sage: f = 4*x^3*y^2 + 1/4*x^3*y*z + 3/2*x^2*z^2 - 1/2*z^2
sage: f.discriminant(x)  # needs sage.libs.singular
1
sage: f.discriminant(y)  # needs sage.libs.singular
1
```

(continues on next page)
Polynomials, Release 10.1

(continued from previous page)

\[-383/16\times^2z^2 + 8\times z^2\]
\[\text{sage: } f\text{.discriminant}(z)\] #→ \text{needs sage.libs.singular}
\[-383/16\times^2y^2 + 8\times y^2\]

Note that, unlike the univariate case, the result lives in the same ring as the polynomial:

\[\text{sage: } R.<x,y> = \text{QQ}[z] \]
\[\text{sage: } f = x^5y + 3x^2y^2 - 2x + y - 1\]
\[\text{sage: } f\text{.discriminant}(y)\] #→ \text{needs sage.libs.singular}
\[x^10 + 2x^5 + 24x^3 + 12x^2 + 1\]
\[\text{sage: } f\text{.polynomial(y).discriminant()}\]
\[x^10 + 2x^5 + 24x^3 + 12x^2 + 1\]
\[\text{sage: } f\text{.discriminant(y).parent()} == f\text{.polynomial(y).discriminant().parent()} \] #→ \text{needs sage.libs.singular}
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:

\[\text{sage: } Q.<z> = \text{Frac(QQ}['z'])\]
\[\text{sage: } R.<x,y> = \text{QQ}[z] \]
\[\text{sage: } r = x^y - (2z-1)/(z^2+z+1) * x + y/z\]
\[\text{sage: } p = r * (x + z*y - 1/z^2)\]
\[\text{sage: } q = r * (x^y*z + 1)\]
\[\text{sage: } \text{gcd}(p, q)\]
\[(z^3 + z^2 + z)*x*y + (-2z^2 + z)*x + (z^2 + z + 1)*y\]

Polynomials over polynomial rings are converted to a simpler polynomial ring with all variables to compute the gcd:

\[\text{sage: } A.<z,t> = \text{ZZ}[z] \]
\[\text{sage: } B.<x,y> = A[z] \]
\[\text{sage: } r = x^y*z^t + 1\]
\[\text{sage: } p = r * (x - y + z - t + 1)\]
\[\text{sage: } q = r * (x*z - y^t)\]
\[\text{sage: } \text{gcd}(p, q)\]
\[z^t*x*y + 1\]
\[\text{sage: } \_\_\_.parent()\]
Multivariate Polynomial Ring in x, y over
Multivariate Polynomial Ring in z, t over Integer Ring

Some multivariate polynomial rings have no gcd implementation:

\[\text{sage: } R.<x,y> = \text{GaussianIntegers}()[z] \] #→ \text{needs sage.rings.number_field}

(continues on next page)
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

gradient()
Return a list of partial derivatives of this polynomial, ordered by the variables of self.parent()

EXAMPLES:

sage: P.<x,y,z> = PolynomialRing(ZZ, 3)
sage: f = x*y + 1
sage: f.gradient()
[y, x, 0]

homogeneous_components()
Return the homogeneous components of this polynomial.

OUTPUT:
A dictionary mapping degrees to homogeneous polynomials.

EXAMPLES:

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()
{}

In case of weighted term orders, the polynomials are homogeneous with respect to the weights:

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}

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

```plaintext
sage: g = f.homogenize('z'); g
5*x*y^10 + x^2*z^9 + y*z^10 + z^11
```

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

```plaintext
sage: f = x^2 + y^2
sage: g = f.homogenize('z'); g
x^2 + y^2
```

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:

```plaintext
sage: R.<x,y,z> = QQ[]
sage: f = x^2 - y
sage: g = f.homogenize(x); g
x^2 - x*y
```

The parameter `var` can also be given as a zero-based index in the list of variables:

```plaintext
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:

```plaintext
sage: g(x,y,1)
x^2 - y
```

If it is present already, this might not be the case:

```plaintext
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:
Polynomials, Release 10.1

```sage
sage: R.<x,y> = GF(2)[
sage: f = x + 1
sage: f.homogenize(x)
```

**inverse_mod(I)**

Returns an inverse of `self` modulo the polynomial ideal `I`, namely a multivariate polynomial `f` such that `self * f - 1` belongs to `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:**

```sage
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
```

```bash
1/16*x1 + 3/16
```

```sage
sage: (f*g).reduce(I)
```

```bash
1
```

Test a non-invertible element:

```sage
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)
```

```bash
Traceback (most recent call last):
...
ArithmeticError: element is non-invertible
```

**is_generator()**

Returns `True` if this polynomial is a generator of its parent.

**EXAMPLES:**

```sage
sage: R.<x,y> = ZZ[
sage: x.is_generator()
```

```bash
True
```

```sage
sage: (x + y - y).is_generator()
```

```bash
True
```

```sage
sage: (x*y).is_generator()
```

```bash
False
```

```sage
sage: R.<x,y> = QQ[
```

```bash
(continues on next page)```
```
sage: (x*y).is_generator()
False
```

**is_homogeneous()**

Return True if `self` is a homogeneous polynomial.

**Note:** This is a generic implementation which is likely overridden by subclasses.

**is_lorentzian(explain=False)**

Return whether this is a Lorentzian polynomial.

**INPUT:**

- `explain` – boolean (default: False); if True return a tuple whose first element is the boolean result of the test, and the second element is a string describing the reason the test failed, or None if the test succeeded.

Lorentzian polynomials are a class of polynomials connected with the area of discrete convex analysis. A polynomial $f$ with positive real coefficients is Lorentzian if:

- $f$ is homogeneous;
- the support of $f$ is $M$-convex
- $f$ has degree less than 2, or if its degree is at least two, the collection of sequential partial derivatives of $f$ which are quadratic forms have Gram matrices with at most one positive eigenvalue.

Note in particular that the zero polynomial is Lorentzian. Examples of Lorentzian polynomials include homogeneous stable polynomials, volume polynomials of convex bodies and projective varieties, and Schur polynomials after renormalizing the coefficient of each monomial $x^\alpha$ by $1/\alpha!$.

**EXAMPLES:**

Renormalized Schur polynomials are Lorentzian, but not in general if the renormalization is skipped:

```
sage: P.<x,y> = QQ[]
sage: p = (x^2 / 2) + x*y + (y^2 / 2)
sage: p.is_lorentzian()
True
sage: p = x^2 + x*y + y^2
sage: p.is_lorentzian()
False
```

Homogeneous linear forms and constant polynomials with positive coefficients are Lorentzian, as well as the zero polynomial:

```
sage: p = x + 2*y
sage: p.is_lorentzian()
True
sage: p = P(5)
sage: p.is_lorentzian()
True
sage: P.zero().is_lorentzian()
True
```

Inhomogeneous polynomials and polynomials with negative coefficients are not Lorentzian:
Polynomials, Release 10.1

```python
sage: p = x^2 + 2*x + y^2
sage: p.is_lorentzian()
False
sage: p = 2*x^2 - y^2
sage: p.is_lorentzian()
False
```

It is an error to check if a polynomial is Lorentzian if its base ring is not a subring of the real numbers, as the notion is not defined in this case:

```python
sage: Q.<z,w> = CC[]
sage: q = z^2 + w^2
sage: q.is_lorentzian()
Traceback (most recent call last):
  ...  
NotImplementedError: is_lorentzian only implemented for real polynomials
```

The method can give a reason for a polynomial failing to be Lorentzian:

```python
sage: p = x^2 + 2*x + y^2
sage: p.is_lorentzian(explain=True)
(False, 'inhomogeneous')
```

REFERENCES:
For full definitions and related discussion, see [BrHu2019] and [HMMS2019]. The second reference gives the characterization of Lorentzian polynomials applied in this implementation explicitly.

```python
is_nilpotent()
Return True if self is nilpotent, i.e., some power of self is 0.

EXAMPLES:
```
```python
sage: R.<x,y> = QQbar[]
#needs sage.rings.number_field
sage: (x + y).is_nilpotent()
#needs sage.rings.number_field
False
sage: R(0).is_nilpotent()
#needs sage.rings.number_field
True
sage: _.<x,y> = Zmod(4[])
sage: (2*x).is_nilpotent()
True
sage: (2 + y*x).is_nilpotent()
False
sage: _.<x,y> = Zmod(36[])
sage: (4 + 6*x).is_nilpotent()
False
sage: (6*x + 12*y + 18*x*y + 24*(x^2+y^2)).is_nilpotent()
True
```

```python
is_square(root=False)
Test whether this polynomial is a square.

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:

```python
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.

INPUT:

• group (default: symmetric group) – if set, test whether the polynomial is invariant with respect to the given permutation group

EXAMPLES:

```python
sage: # needs sage.groups
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()
# needs sage.groups
True
sage: (x + 2*y).is_symmetric()
# needs sage.groups
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)]
```

(continues on next page)
is_unit()

Return True if self is a unit, that is, has a multiplicative inverse.

EXAMPLES:

```python
sage: # needs sage.rings.number_field
sage: R.<x,y> = QQbar[]
sage: (x + y).is_unit()
False
sage: R(0).is_unit()
False
sage: R(-1).is_unit()
True
sage: R(-1 + x).is_unit()
False
sage: R(2).is_unit()
True
```

Check that github issue #22454 is fixed:

```python
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:

```python
sage: R.<a,b,c> = QQ[]
```

```python
sage: f = a*c^3 + a^2*b + 2*b^4
```

```python
sage: list(f.iterator_exp_coeff())
[(0, 4, 0), (1, 0, 3), (2, 1, 0)]
```

```python
sage: list(f.iterator_exp_coeff(as_ETuples=False))
```

(continues on next page)
**jacobian_ideal()**

Return the Jacobian ideal of the polynomial self.

**EXAMPLES:**

```python
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 \( I = (f_1, \ldots, f_r) \) that contains self, find \( s_1, \ldots, s_r \) such that \( \text{self} = s_1 f_1 + \ldots + s_r f_r \).

**EXAMPLES:**

```python
sage: # needs sage.rings.real_mpfr
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); M

\[ \text{needs sage.libs.singular} \]
\[y^7, x^7*y^4 + x^8 + x^5*y^3 + x^6*y + x^3*y^4 + x^4*y^2 + x^5*y^5 + x^2*y^3 + y^4\]

```python
sage: sum(map(mul, zip(M, I.gens()))) == f

\[ \text{needs sage.libs.singular} \]
True
```

**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:
The polynomials need to be all homogeneous:

```
sage: R.<x,y,z> = PolynomialRing(QQ, 3)
sage: y.macaulay_resultant(x + z)
"needs sage.modules"
Traceback (most recent call last):
...
TypeError: number of polynomials (= 2) must equal number of variables (= 3)
```

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)
"needs sage.modules"
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:

```
sage: K.<x,y> = PolynomialRing(ZZ, 2)
sage: flist, R = K._macaulay_resultant_universal_polynomials([1,1,2])
sage: flist[0].macaulay_resultant(flist[1:])
"needs sage.modules"
```

The following example degenerates into the determinant of a $3 \times 3$ matrix:

```
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:])
```

The following example is by Patrick Ingram (arXiv 1310.4114):

```
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*x2 + 2*x1*x2
sage: f1 = y1*x2^2 - x1*x2 + 2*x0*x2
sage: f2 = x0*x1 - x2^2
```

(continues on next page)
sage: f0.macaulay_resultant(f1, f2)  # needs sage.modules
y0^2*y1^2 - 4*y0^3 - 4*y1^3 + 18*y0*y1 - 27

a simple example with constant rational coefficients:

sage: R.<x,y,z,w> = PolynomialRing(QQ, 4)
sage: w.macaulay_resultant([z, y, x])  # needs sage.modules
1

an example where the resultant vanishes:

sage: R.<x,y,z> = PolynomialRing(QQ, 3)
sage: (x + y).macaulay_resultant([y^2, x])  # needs sage.modules
0

an example of bad reduction at a prime \( p = 5 \):

sage: R.<x,y,z> = PolynomialRing(QQ, 3)
sage: y.macaulay_resultant([x^3 + 25*y^2*x, 5*z])  # needs sage.libs.pari sage.modules
125

The input can given as an unpacked list of polynomials:

sage: R.<x,y,z> = PolynomialRing(QQ, 3)
sage: y.macaulay_resultant([x^3 + 25*y^2*x, 5*z])  # needs sage.libs.pari sage.modules
125

an example when the coefficients live in a finite field:

sage: F = FiniteField(11)
sage: R.<x,y,z,w> = PolynomialRing(F, 4)
sage: z.macaulay_resultant([x^3, 5*y, w])  # needs sage.modules sage.rings.finite_rings
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: y.macaulay_resultant([x + z, z^2])  # needs sage.libs.pari sage.modules
-1

When there are only 2 polynomials, the 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()
map_coefficients($f, \text{new\_base\_ring}=\text{None}$)

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:

```
sage: k.<a> = GF(9); R.<x,y> = k[]; f = x*a + 2*x^3*y*a + a
# needs sage.rings.finite_rings
sage: f.map_coefficients(lambda a: a + 1)
# needs sage.rings.finite_rings
(-a + 1)*x^3*y + (a + 1)*x + (a + 1)
```

Examples with different base ring:

```
sage: # needs sage.rings.finite_rings
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> = R[]
sage: f = r*X + Y
sage: g = f.map_coefficients(h); g
(-a + 1)*x^3*y + (a + 1)*x + (a + 1)
sage: g.parent()
Multivariate 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 Polynomial Ring in X, Y over Finite Field of size 3
```

newton_polytope()

Return the Newton polytope of this polynomial.

EXAMPLES:
Polynomials, Release 10.1

```plaintext
sage: R.<x,y> = QQ[]
sage: f = 1 + x*y + x^3 + y^3
sage: P = f.newton_polytope(); P  # needs sage.geometry.polyhedron
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3 vertices
sage: P.is_simple()  # needs sage.geometry.polyhedron
True
```

**nth_root(n)**

Return a \( n \)-th root of this element.

If there is no such root, a ValueError is raised.

**EXAMPLES:**

```plaintext
sage: R.<x,y,z> = QQ[]
sage: a = 32 * (x*y + 1)^5 * (x+y+z)^5
sage: a.nth_root(5)
2*x^2*y + 2*x*y^2 + 2*x*y*z + 2*x + 2*y + 2*z
sage: b = x + 2*y + 3*z
sage: b.nth_root(42)
Traceback (most recent call last):
... ValueError: not a 42nd power
sage: R.<x,y> = QQ[]
sage: S.<z,t> = R[]
sage: T.<u,v> = S[]
sage: p = (1 + x*u + y + v) * (1 + z*t)
sage: (p**3).nth_root(3)
(x*z*t + x)*u + (z*t + 1)*v + (y + 1)*z*t + y + 1
sage: (p**3).nth_root(3).parent() is p.parent()
True
sage: ((1+x+z+t)**2).nth_root(3)
Traceback (most recent call last):
... ValueError: not a 3rd power
```

**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.

```plaintext
sage: R.<x, y> = ZZ[]
sage: f = x^3 + 17*x + y + 1
```

(continues on next page)
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^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*w^3 + 3*w^x + z^5 + x^3

sage: f.polynomial(z)
```
**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

---

\[ z^5 + w^5 + 17x^3w^3 + x^3 + 3xw \]

**sage:** \(R.<x,w,z,k> = \mathbb{Z}[]\)

**sage:** \(f = x^3 + 3w^5x + w^5 + (17w^3)x^3 + z^5 + xwz^5k + 5\)

**sage:** \(f\.polynomial(x)\)

\(x^3 + (17w^3 + wz^5k + 3w)x + w^5 + z^5 + 5\)

**sage:** \(f\.polynomial(w)\)

\(w^5 + 17x^3w^3 + (x^3z^5k + 3x^3w + z^5 + x^3 + 5\)

**sage:** \(f\.polynomial(z)\)

\(z^5 + x^3w^5z + w^5 + 17x^3w^3 + x^3 + 3x^3w^5 + 5\)

**sage:** \(f\.polynomial(k)\)

\(x^3w^5z^5k + w^5 + z^5 + 17x^3w^3 + x^3 + 3x^3w^5 + 5\)

**sage:** \(R.<x,y> = \mathbb{F}(5)[]\)

**sage:** \(f = x^2 + x + y\)

**sage:** \(f\.polynomial(x)\)

\(x^2 + x + y\)

**sage:** \(f\.polynomial(y)\)

\(y + x^2 + x\)
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: R.<x,h> = PolynomialRing(QQ)
sage: f = 19*x^8 - 262*x^7*h + 1507*x^6*h^2 - 4784*x^5*h^3 + 9202*x^4*h^4 - 10962*x^3*h^5 + 7844*x^2*h^6 - 3040*x*h^7 + 475*h^8
sage: f.reduced_form(prec=200, smallest_coeffs=False)  # needs sage.modules sage.rings.complex_interval_field
(-x^8 - 2*x^7*h + 7*x^6*h^2 + 16*x^5*h^3 + 2*x^4*h^4 - 2*x^3*h^5 + 4*x^2*h^6 - 5*h^8,
[ 1 -2]
[ 1 -1])
```

An example where the multiplicity is too high:

```
sage: R.<x,y> = PolynomialRing(QQ)
sage: f = x^3 + 378666*x^2*y - 12444444*x*y^2 + 1234567890*y^3
sage: j = f * (x-545*y)^9
sage: j.reduced_form(prec=200, smallest_coeffs=False)  # needs sage.modules sage.rings.complex_interval_field
Traceback (most recent call last):
  ...
ValueError: cannot have a root with multiplicity >= 12/2
```

An example where Newton's Method does not find the right root:

```
sage: R.<x,y> = PolynomialRing(QQ)
sage: F = x^6 + 3*x^5*y - 8*x^4*y^2 - 2*x^3*y^3 - 44*x^2*y^4 - 8*x*y^5
sage: F.reduced_form(smallest_coeffs=False, prec=400)  # needs sage.modules sage.rings.complex_interval_field
Traceback (most recent call last):
  ...
ArithmeticError: Newton's method converged to z not in the upper half plane
```

An example with covariant on the boundary, therefore a non-unique form:

```
sage: R.<x,y> = PolynomialRing(QQ)
sage: F = 5*x^2*y - 5*x*y^2 - 30*y^3
sage: F.reduced_form(smallest_coeffs=False)  # needs sage.modules sage.rings.complex_interval_field
([1 1]
(continues on next page)
```
An example where precision needs to be increased:

```
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)           # needs sage.modules sage.rings.complex_interval_field
Traceback (most recent call last):
...
ValueError: accuracy of Newton's root not within tolerance(0.000012... > 1e-06), increase precision
sage: F.reduced_form(prec=100, smallest_coeffs=False)           # needs sage.modules sage.rings.complex_interval_field

([-1 -1]
-24*x^3*y^4 - 3*x^2*y^5 - 2*x*y^6 + 16*y^7, [1 0])
```

```
sage: R.<x,y,z> = PolynomialRing(QQ)
sage: F = x^4 + x^3*y^2 + y^2*z
sage: F.reduced_form()                                           # needs sage.modules sage.rings.complex_interval_field
Traceback (most recent call last):
```

(continues on next page)
... ValueError: (=x^3*y*z + x^4 + y^2*z) must have two variables

```
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)  
--- needs sage.modules sage.rings.complex_interval_field
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
```

```
sage: R.<x,y> = PolynomialRing(RR)
sage: F = (217.992172373276*x^3 + 96023.1505442490*x^2*y  
....:  + 1.40987971253579e7*x*y^2 + 6.90016027113216e8*y^3)

sage: F.reduced_form(smallest_coeffs=False)  
--- needs sage.modules sage.rings.complex_interval_field
(  
-39.5673942565918*x^3 + 111.874026298523*x^2*y  
+ 231.052762985229*x*y^2 - 138.380829811096*y^3,  
[-147 -148]  
[ 1  1]  
)
```

```
sage: R.<x,y> = PolynomialRing(CC)  
--- needs sage.rings.real_mpfr
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 + 42882.5139724962*CC.0)*y^3)

sage: F.reduced_form(smallest_coeffs=False)  
--- needs sage.modules sage.rings.complex_interval_field sage.rings.real_mpfr
(  
(-0.759099196558145 - 0.845425869641446*I)*x^3  
+ (-0.5717099908900118 - 0.041813346027929*I)*x^2*y  
+ (0.856525964330103 - 0.0721403997649759*I)*x*y^2  
+ (-0.96531044130330 + 0.75425231465703*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)
• \(\phi\) – \texttt{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*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:

```python
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
```

\texttt{subresultants}(other, variable=None)

Return the nonzero subresultant polynomials of \texttt{self} and \texttt{other}.

INPUT:

• other – a polynomial

OUTPUT: a list of polynomials in the same ring as \texttt{self}

EXAMPLES:

```python
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]
```

\texttt{sylvester\_matrix}(right, variable=None)

Given two nonzero polynomials \texttt{self} and \texttt{right}, return 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:

```python
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)  # needs sage.modules
sage: M

[ 3 y + 1 0 0]
[ 0 3 y + 1 0]
[ 4 y + 2 0 0]
[ 0 4 y + 2 0]
```

If the polynomials share a non-constant common factor then the determinant of the Sylvester matrix will be zero:

```python
sage: M.determinant()  # needs sage.modules
0
sage: f.sylvester_matrix(1 + g, x).determinant()  # needs sage.modules
y^2 - y + 7
```

If both polynomials are of positive degree with respect to variable, the determinant of the Sylvester matrix is the resultant:

```python
sage: f = R.random_element(4) or (x^2 * y^2)
sage: g = R.random_element(4) or (x^2 * y^2)
sage: f.sylvester_matrix(g, x).determinant() == f.resultant(g, x)  # needs sage.libs.singular sage.modules
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*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)
6805673841876926945195958937245974528
sage: q = R.random_element(100, 20)
sage: q.weighted_degree(1, 1, 1) == q.total_degree()
True
```

You may also work with negative weights

```
sage: p.weighted_degree(-1, -2, -1)
-2
```

Note that only integer weights are allowed

```
sage: p.weighted_degree(x, 1, 1)
Traceback (most recent call last):
...
TypeError: unable to convert non-constant polynomial x to Integer Ring
```

The `weighted_degree()` coincides with the `degree()` of a weighted polynomial ring, but the latter is faster.

```
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
```

class sage.rings.polynomial.multi_polynomial.MPolynomial_libsingular

    Bases: MPolynomial

    Abstract base class for MPolynomial_libsingular

    This class is defined for the purpose of `isinstance()` tests. It should not be instantiated.

    EXAMPLES:

```
sage: from sage.rings.polynomial.multi_polynomial import MPolynomial_libsingular
sage: R1.<x> = QQ[]
```
By design, there is a unique direct subclass:

```python
sage: len(sage.rings.polynomial.multi_polynomial.MPolynomial_libsingular.__subclasses__()) <= 1
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)
-x^20 + x^10 + 2*y^5
sage: frob((x + 2*y)^3)
# needs sage.rings.finite_rings
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
```

342 Chapter 3. Multivariate Polynomials
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: MPolynomialRing_macaulay2_repr, PolynomialRing_singular_repr, 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

Element_hidden
alias of MPolynomial_polydict

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, x^2, z, x^2*z, z^2, x^2*z^2, 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
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
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
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
sage: P.monomial_pairwise_prime(x^2*z^3, y^4)
True
```

```sage
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:
Polynomials, Release 10.1

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^2*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.

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 ring.

INPUT:
• f - monomial
• G - list/set of mpolynomials

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: 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)

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)

sum(terms)
Return a sum of elements of this multipolynomial ring.
This is method is much faster than the Python builtin sum().

EXAMPLES:

sage: R = QQ['x']
sage: S = R['y, z']
sage: x = R.gen()
sage: y, z = S.gens()
Comparison with built-in `sum()`:

```
sage: sum([[x*y, 2*x^2*z - 2*x*y, 1 + y + z]])
(-x + 1)*y + (2*x^2 + 1)*z + 1
```

3.1.5 Generic Multivariate Polynomials

AUTHORS:

- David Joyner: first version
- William Stein: use dict’s instead of lists
- Martin Albrecht: 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 `x*y^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)

EXAMPLES:

We verify Lagrange’s four squares identity:

```
sage: R.<a0,a1,a2,a3,b0,b1,b2,b3> = QQbar[]

sage: ((a0^2 + a1^2 + a2^2 + a3^2) * (b0^2 + b1^2 + b2^2 + b3^2) ==
     (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
```

(continued from previous page)
class sage.rings.polynomial.multi_polynomial_element.MPolynomial_element(parent, x)

Bases: MPolynomial

Generic multivariate polynomial.

This implementation is based on the PolyDict.

Todo: As mentioned in their docstring, PolyDict objects never clear zeros. In all arithmetic operations on MPolynomial_element there is an additional call to the method remove_zeros to clear them. This is not ideal because of the presence of inexact zeros, see github issue #35174.

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: # needs sage.rings.number_field
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()

hamming_weight()

Return the number of non-zero coefficients of this polynomial.

This is also called weight, hamming_weight() or sparsity.

EXAMPLES:

```
sage: # needs sage.rings.real_mpfr
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:
number_of_terms()
Return the number of non-zero coefficients of this polynomial.
This is also called weight, \textit{hamming_weight}() or sparsity.

EXAMPLES:

```python
sage: # needs sage.rings.real_mpfr
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 \textit{hamming_weight}() is an alias:

```python
sage: f.hamming_weight()  # needs sage.rings.real_mpfr
sage: f.number_of_terms()
101
```

class sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict (parent, x)
Bases: \texttt{Polynomial}._singular._repr., MPolynomial_element

Multivariate polynomials implemented in pure python using polydicts.

\textbf{coefficient}(\texttt{degrees})
Return the coefficient of the variables with the degrees specified in the python dictionary \texttt{degrees}. Mathematically, this is the coefficient in the base ring adjoined by the variables of this ring not listed in \texttt{degrees}. However, the result has the same parent as this polynomial.

This function contrasts with the function \texttt{monomial_coefficient} which returns the coefficient in the base ring of a monomial.

\textbf{INPUT}:

\begin{itemize}
  \item \texttt{degrees} - Can be any of:
    \begin{itemize}
      \item a dictionary of degree restrictions
      \item a list of degree restrictions (with None in the unrestricted variables)
      \item a monomial (very fast, but not as flexible)
    \end{itemize}
\end{itemize}

\textbf{OUTPUT}:

\texttt{element} of the parent of \texttt{self}

\textbf{See also}:
For coefficients of specific monomials, look at \texttt{monomial_coefficient}().

\textbf{EXAMPLES}:
```python
sage: # needs sage.rings.number_field
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, None})
y^2 + y + 1
sage: f.coefficient(x)
y^2 + y + 1
sage: # Be aware that this may not be what you think!
sage: # 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^2*y^2 + x^2*y + x*y^2 + x^2 + x*y + y^2 + x + y + 1
```

```python
sage: # needs sage.rings.real_mpfr
sage: R.<x,y> = RR[]
sage: f = x*y + 5
sage: c = f.coefficient({x: 0, y: 0}); c
5.00000000000000
sage: parent(c)
Multivariate Polynomial Ring in x, y over Real Field with 53 bits of precision
```

AUTHORS:

- Joel B. Mohler (2007-10-31)

**constant_coefficient()**

Return the constant coefficient of this multivariate polynomial.

EXAMPLES:

```python
sage: # needs sage.rings.number_field
sage: R.<x,y> = QQbar[]
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 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:

```sage
sage: R.<x,y> = RR[]
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
```

Note that total degree takes into account if we are working in a polynomial ring with a weighted term order.

```sage
sage: R = PolynomialRing(QQ, 'x,y', order=TermOrder('wdeglex', (2,3)))
sage: x,y = R.gens()
sage: x.degree()
2
sage: y.degree()
3
sage: x.degree(y), x.degree(x), y.degree(x), y.degree(y)
(0, 1, 0, 1)
sage: f = x^2*y + x*y^2
sage: f.degree(x)
2
sage: f.degree(y)
2
sage: f.degree()
8
sage: f.degree(std_grading=True)
3
```

Note that if $x$ is not a generator of the parent of self, for example if it is a generator of a polynomial algebra which maps naturally to this one, then it is converted to an element of this algebra. (This fixes the problem reported in github issue #17366.)

```sage
sage: x, y = ZZ['x','y'].gens()
sage: GF(3037000453)['x','y'].gen(0).degree(x) # needs sage.rings.finite_rings
1
```

(continues on next page)
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: # needs sage.rings.number_field
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)  # needs sage.rings.number_field
sage: f = a^3 + b + 2*b^2  # needs sage.rings.number_field
```

(continues on next page)
sage: f.exponents()  # needs sage.rings.number_field
[(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])  # needs sage.rings.number_field
<class 'sage.rings.polynomial.polydict.ETuple'>

sage: type(f.exponents(as_ETuples=False)[0])  # needs sage.rings.number_field
<... '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.)

global_height(prec=None)
Return the (projective) global height of the polynomial.
This returns the absolute logarithmic height of the coefficients thought of as a projective point.

INPUT:

• prec – desired floating point precision (default: default RealField precision).

OUTPUT: a real number.

EXAMPLES:

sage: R.<x,y> = PolynomialRing(QQbar, 2)  # needs sage.rings.number_field
sage: f = QQbar(i)*x^2 + 3*x*y  # needs sage.rings.number_field
sage: f.global_height()  # needs sage.rings.number_field
1.09861228866811

Scaling should not change the result:

sage: R.<x> = QQ[]

sage: K.<k> = NumberField(x^2 + 1)
(continues on next page)
sage: Q.<q,r> = PolynomialRing(K, implementation='generic')
sage: f = 12 * q
def: f.global_height()
0.000000000000000

sage: R.<x,y> = PolynomialRing(QQ, implementation='generic')
sage: f = 1/123*x*y + 12
sage: f.global_height(prec=2)
˓→needs sage.symbolic
8.0

\texttt{integral}(\texttt{var}=\texttt{None})

Integrate \texttt{self} with respect to variable \texttt{var}.

\textbf{Note:} The integral is always chosen so the constant term is 0.

If \texttt{var} is not one of the generators of this ring, \texttt{integral(\texttt{var})} is called recursively on each coefficient of this polynomial.

\textbf{EXAMPLES:}

On polynomials with rational coefficients:

\begin{verbatim}
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

sage: R = ZZ['x']['y, z']
sage: y, z = R.gens()
sage: R.an_element().integral(y).parent()
Multivariate Polynomial Ring in y, z
over Univariate Polynomial Ring in x over Rational Field
\end{verbatim}

On polynomials with coefficients in power series:

\begin{verbatim}
sage: # needs sage.rings.number_field
sage: R.<t> = PowerSeriesRing(QQbar)
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: 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
\end{verbatim}
(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: # needs sage.rings.number_field
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 = 10*x^0
sage: g.is_constant()
True

is_generator()

Return True if self is a generator of its parent.

EXAMPLES:

sage: # needs sage.rings.number_field
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: # needs sage.rings.number_field
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
\[
\text{sage: (x^2 + y^2*x).is_homogeneous()}
\]
False
\[
\text{sage: (x^2*y + y^2*x).is_homogeneous()}
\]
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:**

\[
\text{sage: # needs sage.rings.number_field}
\]
\[
\text{sage: R.<x,y> = QQbar[]}
\]
\[
\text{sage: x.is_monomial()}
\]
True
\[
\text{sage: (x + 2*y).is_monomial()}
\]
False
\[
\text{sage: (2*x).is_monomial()}
\]
False
\[
\text{sage: (x*y).is_monomial()}
\]
True

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

\[
\text{sage: (2*x*y).is_term()}
\]
\[
\text{# needs sage.rings.number_field}
\]
True
\[
\text{sage: (2*x*y).is_monomial()}
\]
\[
\text{# needs sage.rings.number_field}
\]
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:**

\[
\text{sage: # needs sage.rings.number_field}
\]
\[
\text{sage: R.<x,y> = QQbar[]}
\]
\[
\text{sage: x.is_term()}
\]
True
\[
\text{sage: (x + 2*y).is_term()}
\]
False
\[
\text{sage: (2*x).is_term()}
\]
True
\[
\text{sage: (7*x^5*y).is_term()}
\]
True

To require leading coefficient 1, use *is_monomial()*:
Polynomials, Release 10.1

```sage
(2*x*y).is_monomial()  # needs sage.rings.number_field
False
(2*x*y).is_term()      # needs sage.rings.number_field
True
```

**is_univariate()**

Return True if this multivariate polynomial is univariate and False otherwise.

**EXAMPLES:**

```sage
# needs sage.rings.number_field
R.<x,y> = QQbar[]
f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
f.is_univariate()
False
g = f.subs({x: 10}); g
700*y^2 + (-2)*y + 305
g.is_univariate()
True
f = x^0
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')  # needs sage.rings.number_field
f = (x^1*y^5*z^2 + x^2*z + x^4*y^1*z^3)
list(f.iterator_exp_coeff())  # needs sage.rings.number_field
[[(4, 1, 3), 1], [(2, 0, 1), 1], [(1, 5, 2), 1]]
```

```sage
R.<x,y,z> = PolynomialRing(QQbar, order='deglex')  # needs sage.rings.number_field
f = (x^1*y^5*z^2 + x^2*z + x^4*y^1*z^3)
list(f.iterator_exp_coeff(as_ETuples=False))  # needs sage.rings.number_field
[[(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:**
Given an ideal \( I = (f_1, ..., f_r) \) and some \( g = (\text{self}) \) in \( I \), find \( s_1, ..., s_r \) such that \( g = s_1 f_1 + ... + s_r f_r \).

**ALGORITHM:** Use Singular.

**EXAMPLES:**

```python
sage: # needs sage.rings.real_mpfr
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^8*y^13 + y^12
sage: M = f.lift(I); M
    # needs sage.libs.singular
[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
    # needs sage.libs.singular
True
```

### \texttt{lm()} 

Return the lead monomial of \texttt{self} with respect to the term order of \texttt{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
```

```python
sage: # needs sage.rings.real_mpfr
sage: R.<x,y,z> = PolynomialRing(CC, 3, order='deglex')
sage: (x^1*y^2*z^4 + x^1*y^1*z^5).lm()
x^1*y^1*z^5
```

```python
sage: # needs sage.rings.number_field
sage: R.<x,y,z> = PolynomialRing(QQbar, 3, order='degrevlex')
sage: (x^4*y^7*z^1 + x^4*y^2*z^3).lm()
x^4*y^7*z^1
```

### \texttt{local_height(v, prec=None)}

Return the maximum of the local height of the coefficients of this polynomial.
INPUT:

- v – a prime or prime ideal of the base ring.
- prec – desired floating point precision (default: default RealField precision).

OUTPUT: a real number.

EXAMPLES:

```sage
R.<x,y> = PolynomialRing(QQ, implementation='generic')
f = 1/1331*x^2 + 1/4000*y
f.local_height(1331)  # needs sage.rings.real_mpfr
7.19368581839511
```

```sage
# needs sage.rings.number_field
R.<x> = QQ[]
K.<k> = NumberField(x^2 - 5)
T.<t,w> = PolynomialRing(K, implementation='generic')
I = K.ideal(3)
f = 1/3*t*w + 3
f.local_height(I)  # needs sage.symbolic
1.09861228866811
```

```sage
R.<x,y> = PolynomialRing(QQ, implementation='generic')
f = 1/2*x*y + 2
f.local_height(2, prec=2)  # needs sage.rings.real_mpfr
0.75
```

`local_height_arch(i, prec=None)`

Return the maximum of the local height at the i-th infinite place of the coefficients of this polynomial.

INPUT:

- i – an integer.
- prec – desired floating point precision (default: default RealField precision).

OUTPUT: a real number.

EXAMPLES:

```sage
R.<x,y> = PolynomialRing(QQ, implementation='generic')
f = 210*x*y
f.local_height_arch(0)  # needs sage.rings.real_mpfr
5.34710753071747
```

```sage
# needs sage.rings.number_field
R.<x> = QQ[]
K.<k> = NumberField(x^2 - 5)
T.<t,w> = PolynomialRing(K, implementation='generic')
f = 1/2*t*w + 3
```

(continues on next page)
lt()

Return 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: # needs sage.rings.number_field
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")
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:

```python
sage: # needs sage.rings.number_field
sage: R.<x,y> = QQbar[]
sage: f = 2 * x * y
sage: c = f.monomial_coefficient(x*y); c
2
sage: c.parent()
Algebraic Field
```

```python
sage: # needs sage.rings.number_field
sage: f = y^2 + y^2*x - x^9 - 7*x + 5*x*y
sage: f.monomial_coefficient(y^2)
```
### sage: f.monomial_coefficient(x*y)
5
### sage: f.monomial_coefficient(x^9)
-1
### sage: f.monomial_coefficient(x^10)
0

### sage: # needs sage.rings.number_field
### sage: a = polygen(ZZ, 'a')
### sage: K.<a> = NumberField(a^2 + a + 1)
### sage: P.<x,y> = K[]
### sage: f = (a*x - 1) * ((a+1)*y - 1); f
- x*y + (-a)*x + (-a - 1)*y + 1
### sage: f.monomial_coefficient(x)
-a

### monomials()

Return the list of monomials in self. The returned list is decreasingly ordered by the term ordering of self.parent().

**OUTPUT:** list of MPolynomial instances, representing monomials

**EXAMPLES:**

```python
### sage: R.<x,y> = QQbar[]
### # needs sage.rings.number_field
### sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
### # needs sage.rings.number_field
### sage: f.monomials()
### # needs sage.rings.number_field
### [x^2*y^2, x^2, y, 1]
```

```python
### sage: R.<fx,fy,gx,gy> = QQbar[]
### sage: F = (fx*gy - fy*gx)^3; 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()

Return the number of variables in this polynomial.

**EXAMPLES:**

```python
### sage: # needs sage.rings.number_field
### sage: R.<fx,fy,gx,gy> = QQbar[]
### sage: F = (fx*gy - fy*gx)^3; 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.nvariables()
### 3
```

(continues on next page)
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[]  # needs sage.rings.real_mpfr
sage: f = y*x^2 + x + 1  # needs sage.rings.real_mpfr
sage: f.quo_rem(x)  # needs sage.libs.singular sage.rings.real_mpfr
(x*y + 1.00000000000000, 1.00000000000000)
sage: R = QQ['a','b'][x,y,z]  # needs sage.libs.singular
sage: p1 = R('a + (1+2*b)*x*y + (3-a^2)*z')
sage: p2 = R('x-1')
sage: p1.quo_rem(p2)  # needs sage.libs.singular
((2*b + 1)*y, (2*b + 1)*y + (-a^2 + 3)*z + a)
sage: R.<x,y> = Qp(5)[]  # needs sage.rings.padics
sage: x.quo_rem(y)  # needs sage.rings.padics
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:**

```
sage: # needs sage.rings.number_field
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
```

(continues on next page)
Polynomials, Release 10.1

- sage: g.reduce(F.gens())
  (-6)*x^2 + 2*y

- sage: f = 3*x
  → needs sage.rings.number_field
- sage: f.reduce([2^x, y])
  → needs sage.rings.number_field
  0

- sage: # needs sage.rings.number_field
- 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 github issue #13903:

- sage: R.<y1,y2> = PolynomialRing(Qp(5), 2, order='lex')
  → needs sage.rings.padics
- sage: G = [y1^2 + y2^2, y1*y2 + y2^2, y2^3]
  → needs sage.rings.padics
- sage: type((y2^3).reduce(G))
  → needs sage.rings.padics
  <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:

- sage: P.<x,y> = PolynomialRing(QQ, 2)
- sage: a = x + y
- sage: b = x^3 - y^3
- sage: a.resultant(b)
  → needs sage.libs.singular
  -2*y^3
- sage: a.resultant(b, y)
  → needs sage.libs.singular
  2*x^3
subresultants\( (\text{other}, \text{variable}=\text{None}) \)
Return the nonzero subresultant polynomials of \text{self} and \text{other}.

INPUT:

- other – a polynomial

OUTPUT: a list of polynomials in the same ring as \text{self}

EXAMPLES:

```sage
sage: # needs sage.rings.number_field
sage: R.<x,y> = QQbar[]
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]
```

subs\( (\text{fixed}=\text{None}, **\text{kw}) \)
Fix some given variables in a given multivariate polynomial and return 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 \{\text{variable}: \text{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
sage: # needs sage.rings.number_field
sage: R.<x,y> = QQbar[]
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
```

total_degree()
Return the total degree of \text{self}, which is the maximum degree of any monomial in \text{self}.

EXAMPLES:

```sage
sage: # needs sage.rings.number_field
sage: R.<x,y,z> = QQbar[]
sage: f = 2*x*y^3*z^2
sage: f.total_degree()
6
sage: f = 4*x^2*y^2*z^3
```
Polynomials, Release 10.1

(continued from previous page)

```
sage: f.total_degree()
sage: f = 99*x^6*y^3*z^9
sage: f.total_degree()
sage: f = x*y^3*z^6 + 3*x^2
sage: f.total_degree()
sage: f = z^3 + 8*x^4*y^5*z
sage: f.total_degree()
sage: f = z^9 + 10*x^4 + y^8*x^2
sage: f.total_degree()
```

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 method *is_univariate*. The new *Polynomial* is over the same base ring as the given *MPolynomial*.

EXAMPLES:

```
sage: # needs sage.rings.number_field
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($i$)

Return the $i$-th variable occurring in this polynomial.

EXAMPLES:

```
sage: # needs sage.rings.number_field
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:

```python
sage: # needs sage.rings.number_field
sage: R.<x,y> = QQbar[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
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

OUTPUT: integer – the difference $val_x(p) - val_x(q)$ where $r = p/q$

Note: This function should be made a method of the FractionFieldElement class.

EXAMPLES:

```python
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
sage: f/g
(-2*b*c^2 - 1)/(2*a*b^3*c^6 + a*c)
sage: degree_lowest_rational_function(f/g, a)
-1
sage: degree_lowest_rational_function(f/g, b)
0
sage: degree_lowest_rational_function(f/g, c)
-1
```

Consider the quotient $f/g = \frac{4+3b^2}{ac+2a^2b^2c^2}$ (note the cancellation).

```python
sage: # needs sage.rings.finite_rings
sage: r = f/g; r
(-2*b*c^2 - 1)/(2*a*b^3*c^6 + 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
```
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:

```sage
x,y,z = QQ['x,y,z'].gens()
I = ideal(x^5 + y^4 + z^3 - 1, x^3 + y^3 + z^2 - 1)
B = I.groebner_basis()
type(B)
<class 'sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic'>
```

Groebner bases can be used to solve the ideal membership problem:

```sage
f,g,h = B
(2*x^2f + g).reduce(B)
0
(2*x^2f + g) in I
True
(2*x^2f + 2*z^2h + y^3).reduce(B)
y^3
(2*x^2f + 2*z^2h + y^3) in I
False
```

We compute a Groebner basis for Cyclic 6, which is a standard benchmark and test ideal.

```sage
R.<x,y,z,t,u,v> = QQ['x,y,z,t,u,v']
I = sage.rings.ideal.Cyclic(R,6)
B = I.groebner_basis()
len(B)
45
```

We compute in a quotient of a polynomial ring over \(\mathbb{Z}/17\mathbb{Z}\):

```sage
R.<x,y> = ZZ[]
S.<a,b> = R.quotient((x^2 + y^2, 17))
S
Quotient of Multivariate Polynomial Ring in x, y over Integer Ring by the ideal (x^2 + y^2, 17)
a^2 + b^2 == 0
True
a^3 - b^2
-a*b^2 - b^2
```

Note that the result of a computation is not necessarily reduced:
Polynomials, Release 10.1

```sage
(a+b)^17
a*b^16 + b^17
sage: S(17) == 0
True
```

Or we can work with \( \mathbb{Z}/17\mathbb{Z} \) directly:

```sage
R.<x,y> = Zmod(17)[]
S.<a,b> = R.quotient((x^2 + y^2,))
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[]
I = ideal(x^2 + y^2 - z^2 - w^2, x-y)
J = I^2
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 - 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)
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 +
```

(continues on next page)
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()
[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()
[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*z + y*z + z^2, -3*x*y + 2*y*z + 6*z^2)
sage: I.change_ring(P.change_ring(QQ)).groebner_basis()
[1]
```

However, when we compute the Groebner basis of $I$ (defined over $\mathbb{Z}$), we note that there is a certain integer in the ideal which is not 1.

```python
sage: I.groebner_basis()
[x + y + 57119*z + 4, y^2 + 3*y + 17220, y*z + y, z^2 + 1
2*y + 158864, z^2 + 17223, 2*z + 41856, 164878]
```

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() → # needs sage.rings.finite_rings
[x + y + z, y^2 + y, y*z + y, z^2 + 1]
sage: I.change_ring(P.change_ring(GF(7))).groebner_basis() → # needs sage.rings.finite_rings
[x - 1, y + 3, z - 2]
sage: I.change_ring(P.change_ring(GF(11777))).groebner_basis() →
```

(continues on next page)
The Groebner basis modulo any product of the prime factors is also non-trivial:

```sage
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:

```sage
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 `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

```python
class sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal(ring, gens, coerce=True)
Bases: MPolynomialIdeal_singular_repr, MPolynomialIdeal_macaulay2_repr, MPolynomialIdeal_magma_repr, Ideal_generic
Create an ideal in a multivariate polynomial ring.
INPUT:
- ring - the ring the ideal is defined in
- gens - a list of generators for the ideal
- coerce - whether to coerce elements to the ring ring
```

### EXAMPLES:

```sage
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
elements in GF(3)
```
sage: P.<x,y> = PolynomialRing(QQ,2)
sage: I = Ideal([x, y + 1])
sage: I.basis
[x, y + 1]

change_ring(P)

Return the ideal I in P spanned by the generators \( g_1, \ldots, g_n \) of self as returned by self.gens().

INPUT:

- \( P \) - a multivariate polynomial ring

EXAMPLES:

sage: P.<x,y,z> = PolynomialRing(QQ,3,order='lex')
sage: I = sage.rings.ideal.Cyclic(P)
sage: I
Ideal (x + y + z, x*y + x*z + y*z, x*y*z - 1) of Multivariate Polynomial Ring in x, y, z over Rational Field

sage: I.groebner_basis()
[x + y + z, y^2 + y*z + z^2, z^3 - 1]

sage: Q.<x,y,z> = P.change_ring(order='degrevlex'); Q
Multivariate Polynomial Ring in x, y, z over Rational Field
sage: Q.term_order()
Degree reverse lexicographic term order

sage: J = I.change_ring(Q); J
Ideal (x + y + z, x*y + x*z + y*z, x*y*z - 1) of Multivariate Polynomial Ring in x, y, z over Rational Field

sage: J.groebner_basis()
[z^3 - 1, y^2 + y*z + z^2, x + y + z]

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 \( g \in K[x_1, \ldots, x_n] \): \( \deg(g \cdot f_i) < D \) and \( g \cdot f_i \in < f_1, \ldots, f_{i-1} > \) 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:

\[
\sum c_k z^k = \frac{\Pi(1-z^{d_i})}{(1-z)^n}
\]

EXAMPLES:

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:

```
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:

```
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 \( n^2 \):

```
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:**
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 \( g_1, \ldots, g_n \) 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 \( g_1, \ldots, g_n \). 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 \( \text{deg} \text{bound} \), 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 \( \text{mult} \text{bound} \). 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
```
Polynomials, Release 10.1

'singular:slimgb'
    libSingular's slimgb command

'singular:stdhilb'
    libSingular's stdhilb command

'singular: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)

'msolve'
    optional package msolve (degrevlex order, prime fields)

'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, and thus 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*a^3*c^3 + 158/7*a^2*c + 8/7*a - 1, b + 30*a^3 - 79/7*a^2 + 3/7*a, c^4 - 10/21*a^3 + 1/84*a^2 + 1/84*a
```

```python
sage: I = sage.rings.ideal.Katsura(P,3)  # regenerate to prevent caching
sage: I.groebner_basis('libsingular:groebner')
```

(continues on next page)
Although Giac does support lexicographical ordering, we use degree reverse lexicographical ordering here, in order to test against github issue #21884:

```python
sage: I = sage.rings.ideal.Katsura(P,3)  # regenerate to prevent caching
sage: J = I.change_ring(P.change_ring(order='degrevlex'))
sage: gb = J.groebner_basis('giac')  # random
sage: gb
[c^3 - 79/210*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]
```

Giac’s gbasis over Q can benefit from a probabilistic lifting and multi threaded operations:

```python
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)
```

The list of available Giac options is provided at `sage.libs.giac.groebner_basis()`. Note that `toy:buchberger` does not return the reduced Groebner basis,
Polynomials, Release 10.1

```
sage: I = sage.rings.ideal.Katsura(P,3)  # regenerate to prevent caching
sage: gb = I.groebner_basis('toy:buchberger')
sage: gb.is_groebner()
  True
sage: gb == gb.reduced()
  False

but that 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: # optional - macaulay2
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')
[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')
[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')
[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]
```

Over prime fields of small characteristic, we can also use the optional package msolve:

```
sage: # optional - msolve
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('msolve')
[a + 2*b + 2*c - 1, b*c - 19*c^2 + 10*b + 40*c, 
b^2 - 41*c^2 + 20*b - 20*c, c^3 + 28*c^2 - 37*b + 13*c]
```

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, y^3 - x]
sage: I.groebner_basis(algorithm='singular', deg_bound=2)
```

(continues on next page)
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.

```python
sage: from sage.misc.verbose import set_verbose
sage: set_verbose(2)
```

The list of available options is provided at `LibSingularOptions`.

Note that Groebner bases over \( \mathbb{Z} \) can also be computed.

```python
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()
```

```python
sage: I.groebner_basis('macaulay2')  # optional - macaulay2
```

376 Chapter 3. Multivariate Polynomials
Groebner bases over \( \mathbb{Z}/n\mathbb{Z} \) are also supported:

```python
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*c^2 + 732*b*c + 808*b,
  2*c^3 + 884*b*c + 666*c^2 + 320*b,
  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]
```

```python
sage: R.<x,y,z> = PolynomialRing(Zmod(2233497349584))
sage: I = R.ideal([z*(x-3*y), 3^2*x^2-y*z, z^2+y^2])
sage: I.groebner_basis()
[2*z^4, y*z^2 + 81*z^3, 248166372176*z^3, 9*x^2 - y*z, y^2 + z^2, x*z + 2233497349581*y*z, 248166372176*y*z]
```

Sage also supports local orderings:

```python
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:

```python
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
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())
[2*x + 95*z, 1/2*y^2 - 1/4*y*z, 0]
```

Groebner bases over fraction fields of polynomial rings are also supported:

```python
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().ideal() == I
True
```

In cases where a characteristic cannot be determined, we use a toy implementation of Buchberger’s algorithm (see github issue #6581):
polynomials

sage: R.<a,b> = QQ[]; I = R.ideal(a^2+b^2-1)
sage: Q = QuotientRing(R,I); K = Frac(Q)
sage: 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, one of the other systems listed above (if available), or a toy implementation.

groebner_cover()
Compute the Gröbner cover of the ideal, over a field with parameters.
The Groebner cover is a partition of the space of parameters, such that the Groebner basis in each part is
given by the same expression.

EXAMPLES:

sage: F = PolynomialRing(QQ,'a').fraction_field()
sage: F.inject_variables()
Defining a
sage: R.<x,y,z> = F[]
sage: I = R.ideal([-x+3*y+z-5,2*x+a*z+4,4*x-3*z-1/a])
sage: I.groebner_cover()
{Quasi-affine subscheme X - Y of Affine Space of dimension 1 over Rational
   Field,
   where X is defined by:
   0
   and Y is defined by:
   2*a^2 + 3*a: [(2*a^2 + 3*a)*z + (8*a + 1),
   (12*a^2 + 18*a)*y + (-20*a^2 - 35*a - 2), (4*a + 6)*x + 11],
Quasi-affine subscheme X - Y of Affine Space of dimension 1 over Rational
   Field,
   where X is defined by:
   ...
   and Y is defined by:
   1: [1],
Quasi-affine subscheme X - Y of Affine Space of dimension 1 over Rational
   Field,
   where X is defined by:
   ...
   and Y is defined by:
   1: [1]}

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 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 gen-
erators of this ideal.

INPUT:

• h - variable name or variable in cover ring (default: ‘h’)

**EXAMPLES:**

```
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
```

```
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
```

```
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
```

```
sage: I = Ideal([x^2*y + z^3 + y^2*x, x + y^2 + 1])
sage: I.homogenize()
Ideal (x^2*y + x*y^2 + z^3, y^2 + x*h + h^2) of
Multivariate Polynomial Ring in x, y, z, h over Finite
Field of size 2
```

**is_homogeneous()**

Return True if this ideal is spanned by homogeneous polynomials, i.e., if it is a homogeneous ideal.

**EXAMPLES:**

```
sage: P.<x,y,z> = PolynomialRing(QQ,3)
sage: I = sage.rings.ideal.Katsura(P)
sage: I
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: I.is_homogeneous()
False

sage: J = I.homogenize()
sage: J
Ideal (x + 2*y + 2*z - h, x^2 + 2*y^2 + 2*z^2 - x*h, 2*x*y + 2*y*z - y*h) of Multivariate Polynomial Ring in x, y, z, h over Rational Field

sage: J.is_homogeneous()
True
```

```
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:

```
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: I = R.ideal([y^3 - x^2])
sage: I.plot()                      # cusp
# needs sage.plot
Graphics object consisting of 1 graphics primitive

sage: I = R.ideal([y^2 - x^2 - 1])
sage: I.plot((x,-3, 3), (y, -2, 2)) # hyperbola
# needs sage.plot
Graphics object consisting of 1 graphics primitive

sage: I = R.ideal([y^2 + x^2*(1/4) - 1])
sage: I.plot()                      # ellipse
# needs sage.plot
Graphics object consisting of 1 graphics primitive

sage: I = R.ideal([y^2-(x^2-1)*(x-2)])
sage: I.plot()                      # elliptic curve
# needs sage.plot
Graphics object consisting of 1 graphics primitive

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
# needs sage.plot
Graphics object consisting of 1 graphics primitive
```
Polynomials, Release 10.1

sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: I = R.ideal([x - 1])
sage: I.plot((y, -2, 2))  # vertical line
needs sage.plot
Graphics object consisting of 1 graphics primitive

sage: I = R.ideal([-x^2*y + 1])
sage: I.plot()  # blow up
needs sage.plot
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 = 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: f = I.random_element(degree=4, compute_gb=True, terms=infinity)
sage: f.degree() <= 4
True
sage: len(list(f)) <= 35
True

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:

3.1. Multivariate Polynomials and Polynomial Rings 381
We return some element in the ideal with no guarantee on the distribution:

```python
sage: # needs sage.rings.finite_rings
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:

```python
sage: # needs sage.rings.finite_rings
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:

```python
sage: P.<x,y> = QQ[]
sage: I = P.ideal([x^2,y^2])
sage: set_random_seed(5)
sage: I.random_element(degree=2)
-2*x^2 + 2*y^2
```

**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:**

```python
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
```

(continues on next page)
Note: Requires computation of a Groebner basis, which can be a very expensive operation.

\texttt{sage:} \texttt{I.reduce(x^6)}
\texttt{y^4 - 2*x*y^2 + x^2}

\texttt{sage:} \texttt{(y^2 - x)^2}
\texttt{y^4 - 2*x*y^2 + x^2}

\texttt{subs}(\texttt{in\_dict}=\texttt{None}, **\texttt{kwd}s)
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:
- \texttt{in\_dict} – (optional) dictionary of inputs
- \texttt{**kwd}s – named parameters

OUTPUT:
A new ideal with modified generators. If possible, in the same polynomial ring. Raises a \texttt{TypeError} if no common polynomial ring of the substituted generators can be found.

EXAMPLES:

\texttt{sage:} \texttt{R.<x,y> = PolynomialRing(ZZ, 2, \textquoteleft xy\textquoteright\)}
\texttt{sage:} \texttt{I = R.ideal(x^5 + y^5, x^2 + y + x^2*y^2 + 5);} \texttt{I}
Ideal \((x^5 + y^5, x^2 + y + x^2*y^2 + 5)\)
of Multivariate Polynomial Ring in \(x, y\) over Integer Ring
\texttt{sage:} \texttt{I.subs(x=y)}
Ideal \((2*y^5, y^4 + y^2 + y + 5)\)
of Multivariate Polynomial Ring in \(x, y\) over Integer Ring
\texttt{sage:} \texttt{I.subs({x: y})} \# same substitution but with dictionary
Ideal \((2*y^5, y^4 + y^2 + y + 5)\)
of Multivariate Polynomial Ring in \(x, y\) over Integer Ring

The new ideal can be in a different ring:

\texttt{sage:} \texttt{R.<a,b> = PolynomialRing(QQ, 2)}
\texttt{sage:} \texttt{S.<x,y> = PolynomialRing(QQ, 2)}
\texttt{sage:} \texttt{I = R.ideal(a^2 + b^2 + a - b + 2);} \texttt{I}
Ideal \((a^2 + b^2 + a - b + 2)\)
of Multivariate Polynomial Ring in \(a, b\) over Rational Field
\texttt{sage:} \texttt{I.subs(a=x, b=y)}
Ideal \((x^2 + y^2 + x - y + 2)\)
of Multivariate Polynomial Ring in \(x, y\) over Rational Field

The resulting ring need not be a multivariate polynomial ring:

\texttt{sage:} \texttt{T.<t> = PolynomialRing(QQ)}
\texttt{sage:} \texttt{I.subs(a=t, b=t)}
Principal ideal \((t^2 + 1)\) of Univariate Polynomial Ring in \(t\) over Rational Field
\texttt{sage:} \texttt{var("z")} 
#
needs sage.symbolic

Principal ideal (2*z^2 + 2) of Symbolic Ring

Variables that are not substituted remain unchanged:

Principa ideal (2*z^2 + 2) of Symbolic Ring

weil_restriction()

Compute 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 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: \[ \sum_{i=0}^{d-1} a^ix_i. \] 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.

If the input and the output ideals are radical, this is equivalent to the statement about algebraic varieties above.

OUTPUT: MPolynomialIdeal

EXAMPLES:

sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(2^2)
sage: P.<x,y> = PolynomialRing(k, 2)
sage: I = Ideal([x*y + 1, a*x + 1])
sage: I.variety()
[[y: a + 1, x: a + 1]]
sage: J = I.weil_restriction()
sage: 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
sage: J += sage.rings.ideal.FieldIdeal(J.ring()) # ensure radical ideal
sage: J.variety()
[[y1: 1, y0: 0, x1: 1, x0: 1]]
Weil restrictions are often used to study elliptic curves over extension fields so we give a simple example involving those:

```python
sage: E += sage.rings.ideal.FieldIdeal(E.ring())  # ensure radical ideal
sage: print(reproducible_repr(E.variety()))
[(x: 1, y: 0, z: 1, x0: 1, x1: 0, x2: 0, x3: 0, x4: 0, y0: 1, y1: 0, y2: 0, y3: 0, y4: 0, z0: 0, z1: 0, z2: 0, z3: 0, z4: 0)]
```

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)  # needs sage.rings.number_field
sage: E = EllipticCurve(K, [1,2,3,4,5])  # needs sage.rings.number_field
```
We pick a point on $E$:

```python
sage: p = E.lift_x(1); p
(1 : -6 : 1)
```

Of course, the point $p$ is a root of all generators of $I$:

```python
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
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(); J
Ideal (0, 0) 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
sage: J.subs(x0=1, y0=2, z0=1, x1=0, y1=0, z1=0)
Ideal (0, 0) of Multivariate Polynomial Ring in x0, x1, y0, y1, z0, z1 over Rational Field
```

Example for relative number fields:

```python
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
   An ideal in a quotient of a multivariate polynomial ring.
   EXAMPLES:
   sage: Q.<x,y,z,w> = QQ['x,y,z,w'].quotient(['x*y-z^2', 'y^2-w^2'])
   sage: J = Q.ideal([u*v - x, u*w - y, t - z])
   sage: J.reduce(t^2 - z^2)
   0

reduce(f)
Reduce an element modulo a Gröbner 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: J.reduce(u^2)
-z^2 + 1
sage: t^2 - z^2 in J
True
sage: u^2 in J
False

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

class sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr
Bases: 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: 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 (`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 $Syz(LM(I))$:

$$S * G = \sum_{i=0}^{m} h_i g_i - \cdots >_G 0.$$  

ALGORITHM:
Uses Singular.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
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:

```
sage: R.<U6,U5,U4,U3,U2, u6,u5,u4,u3,u2, h> = PolynomialRing(GF(7583))  # needs sage.rings.finite_rings
sage: l = [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,
(continues on next page)```
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 no 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()**

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():
    ....:     @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)
<class '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():
    ....:     @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():
    ....:     @cached_method
```

(continues on next page)
def f(self, x): return None
sage: import __main__

When `do_pickle` is set, the pickle contains the contents of the cache:

sage: class A():
  ....:     @cached_method(do_pickle=True)
  ....:     def f(self, x): return None

Cached methods cannot be copied like usual methods, see github issue #12603. Copying them can lead to very surprising results:

sage: class A:
  ....:     @cached_method
  ....:     def f(self):
  ....:         return 1
sage: class B:
  ....:     g=A.f
  ....:     def f(self):
  ....:         return 2

The dimension of the ring modulo this ideal.

EXAMPLES:

sage: P.<x,y,z> = PolynomialRing(GF(32003), order='degrevlex')

(continues on next page)
needs sage.rings.finite_rings
sage: I.dimension()

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:

sage: I = R.ideal([x*y, x*y + 1])
sage: I.dimension()

sage: I = R.ideal([x^3*y, x*y^2])
sage: I.dimension()

sage: R.<x,y> = PolynomialRing(GF(2147483659^2), order='lex')

sage: I = ideal([x*(x*y+1), y*(x*y+1)])
sage: I.dimension()

sage: I = R.ideal(0)
sage: I.dimension()

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:

```python
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:

```python
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.

---

**free_resolution**(*args, **kwds*)

Return a free resolution of `self`.

For input options, see `FreeResolution`.

EXAMPLES:

```python
sage: R.<x,y> = PolynomialRing(QQ)
sage: f = 2*x^2 + y
data: g = y
data: h = 2*f + g
data: I = R.ideal([f,g,h])
data: res = I.free_resolution()
data: res
S^1 <-- S^2 <-- S^1 <-- 0
data: ascii_art(res.chain_complex())
[-x^2]
[ y x^2] [ y]
0 <-- C_0 <------- C_1 <------- C_2 <-- 0
sage: q = ZZ['q'].fraction_field().gen()
sage: R.<x,y,z> = q.parent()[]
sage: I = R.ideal([x^2+y^2+z^2, x*y*z^4])
sage: I.free_resolution()
Traceback (most recent call last):
  ... Not ImplementedError: the ring must be a polynomial ring using Singular
```
genus()

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():
    ....:     @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)
<class '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():
    ....:     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():
    ....:     @cached_method
```

(continues on next page)
When `do_pickle` is set, the pickle contains the contents of the cache:

```
sage: class A():
    ....:     @cached_method(do_pickle=True)
    ....:     def f(self, x):
    ....:         return None
sage: __main__.A = A
sage: a = A()
sage: len(a.f.cache)
1
sage: b = loads(dumps(a))
sage: len(b.f.cache)
0
```

Cached methods cannot be copied like usual methods, see [github issue #12603](https://github.com/sagemath/sage/issues/12603). Copying them can lead to very surprising results:

```
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
```

`graded_free_resolution(*args, **kwds)`

Return a graded free resolution of `self`.

For input options, see `GradedFreeResolution`.

EXAMPLES:
sage: R.<x,y> = PolynomialRing(QQ)
sage: f = 2*x^2 + y^2
sage: g = y^2
sage: h = 2*f + g
sage: I = R.ideal([f,g,h])
sage: I.graded_free_resolution(shifts=[1])
S(-1) <-- S(-3)⊕ S(-3) <-- S(-5) <-- 0
sage: f = 2*x^2 + y
sage: g = y
sage: h = 2*f + g
sage: I = R.ideal([f,g,h])
sage: I.graded_free_resolution(degrees=[1,2])
S(0) <-- S(-2)⊕ S(-2) <-- S(-4) <-- 0
sage: q = ZZ['q'].fraction_field().gen()
sage: R.<x,y,z> = q.parent()

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 $\mathbb{Z}$ and $n$ the number of variables in $R$. This method returns $Q(t)$, the numerator; hence the name, 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()
# needs sage.libs.flint
-t^5 + 1
sage: R.<a,b> = PolynomialRing(QQ)
sage: J = R.ideal([a^2*b, a*b^2])
sage: J.hilbert_numerator()
# needs sage.libs.flint
t^4 - 2*t^3 + 1
sage: J.hilbert_numerator(grading=(10,3))
# needs sage.libs.flint
t^26 - t^23 - t^16 + 1
```
hilbert_polynomial\( (algorithm='sage') \)

Return the Hilbert polynomial of this ideal.

INPUT:

- 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()
# needs sage.libs.flint
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()
# needs sage.libs.flint
0
```

It is possible to request a computation using the Singular library:

```python
sage: I.hilbert_polynomial(algorithm='singular') == I.hilbert_polynomial() # needs sage.libs.flint
True
sage: J.hilbert_polynomial(algorithm='singular') == J.hilbert_polynomial() # needs sage.libs.flint
True
```

Here is a bigger examples:

```python
sage: n = 4; m = 11; P = PolynomialRing(QQ, n * m, "x"); x = P.gens()
sage: N = Matrix(n, x)
sage: Minors = P.ideal(N.minors(2))
sage: hp = Minors.hilbert_polynomial(); hp # needs sage.libs.flint
1/21772800*t^13 + 61/21772800*t^12 + 1661/21772800*t^11 + 26681/21772800*t^10 + 93841/7257600*t^9 + 685421/7257600*t^8 + 1524869/3110400*t^7 + 39780323/21772800*t^6 + 6638071/1360800*t^5 + 12509761/1360800*t^4 + 2689031/226800*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 github issue #26300):
Polynomials, Release 10.1

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:

sage: P = PowerSeriesRing(QQ, 't', default_prec=50)
sage: hs = Minors.hilbert_series()  # needs sage.libs.flint
sage: list(P(hs.numerator()) / P(hs.denominator())) == [hp(t=k)  # needs sage.libs.flint
....: 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 $HS(t) = \sum_{d=0}^{\infty} H(d)t^d$.

This power series can be expressed as $HS(t) = Q(t)/(1 - t)^n$ where $Q(t)$ is a polynomial over $\mathbb{Z}$ 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:

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()  # needs sage.libs.flint
(t^4 + t^3 + t^2 + t + 1)/(t^2 - 2*t + 1)
sage: R.<a,b> = PolynomialRing(QQ)
sage: J = R.ideal([a^2*b, a*b^2])
sage: J.hilbert_series()  # needs sage.libs.flint
(t^3 - t^2 - t - 1)/(t - 1)
sage: J.hilbert_series(grading=(10,3))  # needs sage.libs.flint
(t^25 + t^24 + t^23 - t^15 - t^14 - t^13 - t^12 - t^11 - t^10 - t^9 - t^8 - t^7 - t^6 - t^5 - t^4 - t^3 - t^2 - t - 1)/(t^12 + t^11 + t^10 - t^2 - t - 1)
integral_closure($p=0,$ $r=True,$ $\text{singular}='\text{singular\_default}')$

Let $I = \text{self}$.

Return the integral closure of $I, ..., I^p$, where $sI$ is an ideal in the polynomial ring $R = k[x(1), ..., x(n)]$. 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:

```
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 $(f_1, ..., f_n)$, return $(g_1, ..., g_s)$ such that:

- $(f_1, ..., f_n) = (g_1, ..., g_s)$
- $LT(g_i) \neq LT(g_j)$ for all $i \neq j$
- $LT(g_i)$ does not divide $m$ for all monomials $m$ of $\{g_1, ..., g_{i-1}, g_{i+1}, ..., g_s\}$
- $LC(g_i) = 1$ for all $i$ if the coefficient ring is a field.

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ)
sage: I = Ideal([z^3 + y^3, z + y^3, z + x*y])
sage: I.interreduced_basis()
[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: I = Ideal([z^3 + y^3, z + y^3, z + x*y])
sage: I.interreduced_basis()
[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
sage: I.interreduced_basis()
[x]

The interreduced basis of 0 is 0:

sage: P.<x,y,z> = GF(2)[]
sage: Ideal(P(0)).interreduced_basis()
[0]

ALGORITHM:

Uses Singular's interred command or \texttt{sage.rings.polynomial.toy_buchberger.inter_reduction()} if conversion to Singular fails.

intersection(*others)

Return the intersection of the arguments with this ideal.

EXAMPLES:

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.

sage: I = (x^2, y) * R
sage: J = (y^2, x) * R
sage: K = I.intersection(J)
Ideal (y^2, x*y, x^2) of Multivariate Polynomial Ring in x, y over Rational Field
sage: IJ = I*J
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:

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*z) * R
sage: I1.intersection(I2, I3, I4).groebner_basis()
[x^2*y + x*y*z^4, x*y^2 - x*y*z^3, x*y*z^20 - x*y*z^3]

The ideals must share the same ring:

sage: R2.<x,y> = PolynomialRing(QQ, 2, order='lex')
sage: R3.<x,y,z> = PolynomialRing(QQ, 3, order='lex')

(continues on next page)
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:**

```
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 github issue #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:

```
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)
```

(continues on next page)
[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.

\texttt{normal\_basis}(\texttt{degree=None}, \texttt{algorithm='libsingular'}, \texttt{singular='singular\_default'})

Return a vector space basis of the quotient ring of this ideal.

INPUT:

\begin{itemize}
  \item \texttt{degree} – integer (default: None)
  \item \texttt{algorithm} – string (default: "libsingular"); if not the default, this will use the \texttt{kbase()} or \texttt{weightKB()} command from Singular
  \item \texttt{singular} – the singular interpreter to use when \texttt{algorithm} is not "libsingular" (default: the default instance)
\end{itemize}

OUTPUT:

Monomials in the basis. If \texttt{degree} is given, only the monomials of the given degree are returned.

EXAMPLES:

\begin{verbatim}
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]
\end{verbatim}

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.

\begin{verbatim}
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^2, x*y], [z^3, y*z^2, x^2*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^2, x*y], [z^3, y*z^2, x^2*z^2, x*y*z]]
\end{verbatim}

In case of a polynomial ring with a weighted term order, the degree of the monomials is taken with respect to the weights.

\begin{verbatim}
sage: T = TermOrder('wdegrevlex', (1, 2, 3))
sage: R.<x,y,z> = PolynomialRing(QQ, order=T)
sage: B = R.ideal(x^2*y^2 + x^5, z^2*y + x^3*y).normal_basis(9); B
[x^2*y^2*z, x^3*z^2, x*y*z^2, z^3]
sage: all(f.degree() == 9 for f in B)
True
\end{verbatim}
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:

\[
\begin{align*}
\text{sage: } & \text{R.<x,y> = PolynomialRing(QQ,2)} \\
\text{sage: } & \text{I = R.ideal([y^3 - x^2])} \\
\text{sage: } & \text{I.plot() } \# \text{ cusp} \\
\text{sage: } & \text{I = R.ideal([y^2 - x^2 - 1])} \\
\text{sage: } & \text{I.plot() } \# \text{ hyperbola} \\
\text{sage: } & \text{I = R.ideal([y^2 + x^2*(1/4) - 1])} \\
\text{sage: } & \text{I.plot() } \# \text{ ellipse} \\
\text{sage: } & \text{I = R.ideal([y^2-(x^2-1)*(x-2)])} \\
\text{sage: } & \text{I.plot() } \# \text{ elliptic curve}
\end{align*}
\]

Implicit plotting in 3-d:

\[
\begin{align*}
\text{sage: } & \text{R.<x,y,z> = PolynomialRing(QQ,3)} \\
\text{sage: } & \text{I = R.ideal([y^2 + x^2*(1/4) - z])} \\
\text{sage: } & \text{I.plot() } \# \text{ a cone; optional - surf} \\
\text{sage: } & \text{I = R.ideal([y^2 + z^2*(1/4) - x])} \\
\text{sage: } & \text{I.plot() } \# \text{ same code, from a different angle; optional - surf} \\
\text{sage: } & \text{I = R.ideal([x^2*y^2+x^2*z^2+y^2*z^2-16*x*y*z])} \\
\text{sage: } & \text{I.plot() } \# \text{ Steiner surface; optional - surf}
\end{align*}
\]

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 \( a \not\in Q \), 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 self.

EXAMPLES:

```sage
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.primary_decomposition(); sorted(pd, key=str)
[Ideal (z^2 + 1, y + 1)
  of Multivariate Polynomial Ring in x, y, z over Rational Field,
Ideal (z^6 + 4*z^3 + 4, y - z^2)
  of Multivariate Polynomial Ring in x, y, z over Rational Field]
```

```sage
sage: from functools import reduce
sage: reduce(lambda Qi, Qj: Qi.intersection(Qj), pd) == I
True
```

ALGORITHM:
Uses Singular.

REFERENCES:
• Thomas Becker and Volker Weispfenning. Groebner Bases - A Computational Approach To Commu-

`primary_decomposition_complete()`
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
sage: class Foo():
    ....:     @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
sage: res = a.f(3, 2); res
computing
9
sage: a.f(t = 3, x = 2) is res
True
```

(continues on next page)
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)
<class '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():
    ...:     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')
```

The parameter `do_pickle` can be used to enable pickling of the cache. Usually the cache is not stored when pickling:

```
sage: class A():
    ...:     @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():
    ...:     @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)
1
`sage: b = loads(dumps(a))
sage: len(b.f.cache)
1
```
Cached methods cannot be copied like usual methods, see github issue #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()
```

```python
sage: b.f() 2
sage: b.g() 1
sage: b.f() 1
```

quotient($J$)

Given ideals $I = self$ 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: # needs sage.rings.finite_rings
sage: R.<x,y,z> = PolynomialRing(GF(181), 3)
```

```python
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:

```python
sage: R.<x,y,z> = PolynomialRing(QQ, 3)
sage: I = (x^2, y^3, (x*z)^4 + y^3 + 10*x^2) * R
```

```python
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: # needs sage.rings.finite_rings
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^5*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
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]
```

(continues on next page)
sage: G = vector(I.gens())
sage: M*G
(0, 0)

ALGORITHM:
Uses Singular's syz command.

\texttt{transformed\_basis(algorithm=}'gwalk', other\_ring=None, singular=}'singular\_default'\texttt{)}

Return a lex or other\_ring Groebner Basis for this ideal.

INPUT:

- \texttt{algorithm} - see below for options.
- \texttt{other\_ring} - only valid for \texttt{algorithm=}'fglm'; if provided, conversion will be performed to this ring. Otherwise a lex Groebner basis will be returned.

ALGORITHMS:

- \texttt{"fglm"} - FGLM algorithm. The input ideal must be given with a reduced Groebner Basis of a zero-dimensional ideal
- \texttt{"gwalk"} - Groebner Walk algorithm (default)
- \texttt{"awalk1"} - 'first alternative' algorithm
- \texttt{"awalk2"} - 'second alternative' algorithm
- \texttt{"twalk"} - Tran algorithm
- \texttt{"fwalk"} - Fractal Walk algorithm

EXAMPLES:

sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: I = Ideal([y^3+x^2,y^2*z+x^2, x^3-x^2, z^4-x^2-y])
sage: J = Ideal(I.groebner_basis())
sage: S.<z,x,y> = PolynomialRing(QQ,3,order=}'lex')
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

sage: R.<z,y,x> = PolynomialRing(GF(32003), 3, order=}'lex')  # needs sage.rings.finite_rings
sage: I = Ideal([y^3 + x*y*z + y^2*z + x*z^3, 3 + x*y + x^2*y + y^2*z])  # needs sage.rings.finite_rings
sage: I.transformed_basis('gwalk')  # needs sage.rings.finite_rings
[z^2*y^2 + y*x^2 + y^3, x^2 + 8297*y^8*x^2 + 8297*y^8*x + 3556*y^7 - 8297*y^6*x^4 + 15409*y^6*x^3 - 8297*y^5*x^5 + 15409*y^5*x^4 - 8297*y^5*x^3 + 3556*y^5*x^2 + 3556*y^4*x^3 + 10668*y^4*x^2 - 10668*y^3*x + 1185*y^2*x^8 + 14224*y^2*x^7 - 1185*y^2*x^6 - 8297*y^2*x^5 - 14223*y^2*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]
polynomials, Release 10.1

(continued from previous page)

- 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 - 27*x

algorithm:
Uses Singular.

triangular_decomposition(algorithm=None, singular='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:
• "singular:triangL" - decomposition of self into triangular systems (Lazard).
• "singular:triangLfak" - decomposition of self into triangular systems plus factorization.
• "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:

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 + a^3 + a^2 + a + 1, -d + a^3,
e + a^2) 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 - 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) 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 - 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 + 6*b*a^2 + 10*b*a + 2 + b^2 + 16*a^3 + 23*a^2 + 104*a - 124,
  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

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, 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]

variety(ring=None)

Return the variety of this ideal.

Given a zero-dimensional ideal \( I = \text{self} \) of a polynomial ring \( P \) whose order is lexicographic, return the variety of \( I \) as a list of dictionaries with (variable, value) pairs. By default, the variety of the ideal over its coefficient field \( K \) is returned; 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 ring is specified, then a triangular decomposition of self is found over the original coefficient field \( K \); then the triangular systems are solved using root-finding over ring. This is particularly useful when \( K \) is QQ (to allow fast symbolic computation of the triangular decomposition) and ring is RR, AA, CC, or QQbar (to compute the whole real or complex variety of the ideal).

Note that with ring=RR or CC, computation is done numerically and potentially inaccurately; in particular, the number of points in the real variety may be miscomputed. With ring=AA or QQbar, computation is done exactly (which may be much slower, of course).

INPUT:

- ring - return roots in the ring instead of the base ring of this ideal (default: None)
- algorithm - algorithm or implementation to use; see below for supported values
• proof - return a provably correct result (default: True)

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
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)  # check that all polynomials vanish
    .....: for f in I.gens() for v in V]
[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()  # needs sage.rings.finite_rings
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
sage: I.variety()
[{{y: 1, x: 1}}]
sage: sorted(I.variety(ring=RR), key=str)
[{{y: 0.3611030852864740, x: 2.7692923542386323}},
 {{y: 1.0000000000000000, x: 0.0000000000000000}}]
sage: I.variety(ring=AA)  # needs sage.rings.number_field
[{{y: 1, x: 1},
 {{y: 0.3611030852864740, x: 2.7692923542386323}}]
```

and a total of four intersections:
We can also use the optional package msolve to compute the variety. See msolve for more information.

```python
sage: I.variety(RBF, algorithm='msolve', proof=False) # optional - msolve
[[x: [2.76929235423863 +/- 2.08e-15], y: [0.361103080528647 +/- 4.53e-16]],
 [x: 1.000000000000000, y: 1.000000000000000]]
```

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:

```python
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:

```python
sage: # needs sage.rings.finite_rings
sage: R.<x,y> = PolynomialRing(GF(2147483659^3), 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:

```python
sage: # needs sage.rings.number_field
sage: K.<x,y> = QQ[]
sage: I = ideal([x^2 + 2*y - 5, x + y + 3])
sage: v = I.variety(AA)[0]; v[x], v[y]
(4.464101615137755?, -7.464101615137755?)
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?
```

msolve also works over finite fields:

```python
sage: R.<x, y> = PolynomialRing(GF(536870909), 2, order='lex')
# needs sage.rings.finite_rings
sage: I = Ideal([x^2 - 1, y^2 - 1])
# needs sage.rings.finite_rings
sage: sorted(I.variety(algorithm='msolve', proof=False, key=str))

but may fail in small characteristic, especially with ideals of high degree with respect to the characteristic:

```python
sage: R.<x, y> = PolynomialRing(GF(3), 2, order='lex')
sage: I = Ideal([x^2 - 1, y^2 - 1])
sage: I.variety(algorithm='msolve', proof=False)  # optional - msolve
Traceback (most recent call last):
...:
NotImplementedError: characteristic 3 too small
```

**ALGORITHM:**

- With `algorithm = "triangular_decomposition"` (default), uses triangular decomposition, via Singular if possible, falling back on a toy implementation otherwise.

- With `algorithm = "msolve"`, uses the optional package msolve. Note that msolve uses heuristics and therefore requires setting the `proof` flag to `False`. See msolve for more information.

**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:**
When the ideal is not zero-dimensional, we return infinity:

```python
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 github issue #8586:

```python
sage: P.<x,y,z> = PolynomialRing(GF(32003), 3)
# needs sage.rings.finite_rings
sage: sage.rings.ideal.FieldIdeal(P).vector_space_dimension() # known bug,
            # needs sage.rings.finite_rings
32777216864027
```

class `sage.rings.polynomial.multi_polynomial_ideal.NCPolynomialIdeal` (`ring`, `gens`, `coerce=True`, `side='left'`)

Bases: `MPolynomialIdeal_singular_repr`, `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: # needs sage.combinat sage.modules
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()], # indirect doctest
                ....:            coerce=False)
```

Left Ideal $(y^2, x^2, z^2 - 1)$ of Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field, nc-relations: [z^2: x^z + 2*x, z*y: y*z - 2*y, y*x: x*y - z]

```python
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.
```

**elimination_ideal** *(variables)*

Return the elimination ideal of this ideal with respect to the variables given in variables.

EXAMPLES:

```
sage: # needs sage.combinat sage.modules
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,
ncrelations: {...}
sage: J = I.twostd(); 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,
ncrelations: {...}
sage: J.elimination_ideal([x, z])
Twosided Ideal (y^2) of
Noncommutative Multivariate Polynomial Ring in x, y, z over Rational Field,
ncrelations: {...}
```

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.

EXAMPLES:

```
sage: # needs sage.combinat sage.modules
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()],
....: coerc=False, side='twosided')
sage: Q = H.quotient(I); Q #random
Quotient of
```

(continues on next page)
Polynomials, Release 10.1

(continued from previous page)

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:

sage: H.2^2 - H.one() in I.std().gens()  # needs sage.combinat sage.modules
True

Here is the corresponding direct test:

sage: I.reduce(z^2)  # needs sage.combinat sage.modules
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:

sage: # needs sage.combinat sage.modules
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.res(3)
<Resolution>

std()
Compute a GB of the ideal. It is two-sided if and only if the ideal is two-sided.

EXAMPLES:

sage: # needs sage.combinat sage.modules
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.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 \texttt{std()} returns a left Groebner basis. But if it is a two-sided ideal, then the output of \texttt{std()} and \texttt{twostd()} coincide:

\begin{verbatim}
sage: # needs sage.combinat sage.modules
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, y*z: 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, y*z: 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, y*z: 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
\end{verbatim}

ALGORITHM: Uses Singular's \texttt{std} command

\texttt{syzzygy\_module()}

Compute 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:

\begin{verbatim}
sage: # needs sage.combinat sage.modules
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)
\end{verbatim}
sage: G = vector(I.gens()); G

```
[8*z - 15 0 y^2]
[0 -z^2 + 8*z - 15 x^2]
```

sage: M = I.syzygy_module(); M

```
[ -z^2 - 8*z - 15 0 y^2]
[ 0 -z^2 + 8*z - 15 x^2]
[ 2*z + 15*x^2 -y^3]
[ 2*z^2 + 8*y^2*z - 15*y^2 -4*x*y^2*z + 2*z^2 + 2*z]
[ x^2*y^2*z^2 + 9*x^2*y^2*z - 6*x^2*z^3 + 20*x^2*y - 72*x*z^2 -y^2]
[ y^3*z^2 + 7*y^3*z - 12*y^3 -6*y^4]
[ -3*z + 12*x^3 -x^6*y^2*z^2 + 9*x^6*y^2*z - 4*x^6*y^3 - 20*x^6*y^2 +y^6]
[ 52*y^6*z^2 - 224*y^6 + 320*y]
[ x^2*y^2*z^2 + 4*x^2*y^2 - 8*x*y^2*z^2 - 48*x*y^2*z + 12*y^3 - 64*x^2*y + 108*x*z^2 +y^4]
[ y^3*z^2 + 7*y^3*z - 12*y^3 -6*y^4]
[ 0]
```

sage: M*G

```
 needs sage.combinat sage.modules
```

(continues on next page)
ALGORITHM: Uses Singular’s syz command

twostd()

Compute a two-sided GB of the ideal (even if it is a left ideal).

EXAMPLES:

```
sage: # needs sage.combinat sage.modules
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()  # random
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: 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 a 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 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:

```
sage: sr = mq.SR(2, 1, 2, 4, gf2=True, polybori=True)  # needs sage.rings.polynomial.pbori
sage: sr
SR(2,1,2,4)
```

We can construct a polynomial sequence for a random plaintext-ciphertext pair and study it:

```
sage: set_random_seed(1)
sage: while True:  # workaround (see :trac:`31891`)  # needs sage.rings.polynomial.pbori
....:     try:
....:         F, s = sr.polynomial_system()
....:     except ZeroDivisionError:
....:         pass
sage: F  # needs sage.rings.polynomial.pbori
Polynomial Sequence with 112 Polynomials in 64 Variables
```

(continues on next page)
We separate the system in independent subsystems:

\[
\begin{align*}
\text{sage: } C &= \text{Sequence(r2).connected_components(); } C \\
\text{# needs sage.rings.polynomial.pbori} \\
\mathbb{Z}_2 &= \{w100 + x100 + x102 + x103, \\
w101 + x101 + x100 + x101 + x103 + 1, \\
w102 + x102 + x100 + x101 + x102 + 1, \\
w103 + x103 + x101 + x102 + x103 + 1, \\
x100 + x101 + x102 + x103 + 1, \\
x100 + x101 + x100 + x101 + x103 + x102 + 1, \\
x100 + x101 + x102 + x103 + x102 + 1, \\
x100 + x101 + x102 + x103 + x100 + x102 + 1, \\
x100 + x101 + x102 + x103 + x100 + x102 + 1, \\
x100 + x101 + x102 + x103 + x100 + x102 + 1, \\
x100 + x101 + x102 + x103 + x100 + x102 + 1, \\
x100 + x101 + x102 + x103 + x100 + x102 + 1, \\
x100 + x101 + x102 + x103 + x100 + x102 + 1, \\
x100 + x101 + x102 + x103 + x100 + x102 + 1, \\
x100 + x101 + x102 + x103 + x100 + x102 + 1, \\
x100 + x101 + x102 + x103 + x100 + x102 + 1, \\
x100 + x101 + x102 + x103 + x100 + x102 + 1, \\
x100 + x101 + x102 + x103 + x100 + x102 + 1, \\
x100 + x101 + x102 + x103 + x100 + x102 + 1, \\
x100 + x101 + x102 + x103 + x100 + x102 + 1, \\
x100 + x101 + x102 + x103 + x100 + x102 + 1, \\
x100 + x101 + x102 + x103 + x100 + x102 + 1, \\
x100 + x101 + x102 + x103 + x100 + x102 + 1, \\
x100 + x101 + x102 + x103 + x100 + x102 + 1, \\
x100 + x101 + x102 + x103 + x100 + x102 + 1,
\end{align*}
\]

(continues on next page)
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(1,1,1,4, gf2=True, polybori=True, order='lex')
sage: while True:
    try:
        F, s = sr.polynomial_system()
        break
    ....:  
```

(... continues on next page)
..: except ZeroDivisionError:
..: pass

..: # needs sage.rings.polynomial.pbori
..: monomials = [a*b for a in F.variables() for b in F.variables() if a < b]
..: len(monomials)
190
..: F2 = Sequence(map(mul, cartesian_product_iterator((monomials, F))))
..: A, v = F2.coefficient_matrix(sparse=False)
..: A.echelonize()
..: A
6840 x 4474 dense matrix over Finite Field of size 2...
..: A.rank()
4056
..: 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:**

..: P.<a,b,c,d> = PolynomialRing(GF(127), 4)
..: I = sage.rings.ideal.Katsura(P) # needs sage.libs.singular

If a list of tuples is provided, those form the parts:

..: F = Sequence([I.gens(),I.gens()], I.ring()); F # indirect doctest # needs sage.libs.singular
[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,
(continues on next page)
sage: F.nparts()  
needs sage.libs.singular
2

If an ideal is provided, the generators are used:

sage: Sequence(I)  
needs sage.libs.singular
[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  
needs sage.libs.singular
[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]

We test that the ring is inferred correctly:

sage: P.<x,y,z> = GF(2)[]  
sage: from sage.rings.polynomial.multi_polynomial_sequence import PolynomialSequence  
sage: PolynomialSequence([[1,x,y]], [0]).ring()  
Multivariate Polynomial Ring in x, y, z over Finite Field of size 2

sage: PolynomialSequence([[1,x,y], [0]]).ring()  
Multivariate Polynomial Ring in x, y, z over Finite Field of size 2

```
class sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic(parts, 
ring, immutable=False, 
cr=False, 
cr_str=None)
```

Bases: Sequence_generic

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:

```python
sage: P.<a,b,c,d> = PolynomialRing(GF(127), 4)
sage: I = sage.rings.ideal.Katsura(P)  # needs sage.rings.finite_rings
sage: Sequence([I.gens()], I.ring())  # indirect doctest  # needs sage.rings.finite_rings
[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:

```python
sage: Sequence(I)  # needs sage.rings.finite_rings
[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:

```python
sage: Sequence(I.gens(), I.ring())  # needs sage.rings.finite_rings
[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 + T0^8) of Multivariate Polynomial Ring in T0, T1, T2 over Rational Field
```

```python
sage: [F(S) for F in I.gens()]
[0]
```
sage: R.<x,y> = PolynomialRing(GF(7))
sage: S = Sequence([x, (x^2 + y^2 - 1)^2, x*y - 2])
sage: I = S.algebraic_dependence(); I

# needs sage.rings.finite_rings
Ideal (2 - 3*T2 - T0^2 + 3*T2^2 - T0*T2 + T2^3 + 2*T0^4 - 2*T0*T2^2 + T2^4 - T0^4*T1 + T0^4*T2 - 2*T0^6 + 2*T0^4*T2^2 + T0^8)
of Multivariate Polynomial Ring in T0, T1, T2 over Finite Field of size 7
sage: [F(S) for F in I.gens()]

# needs sage.rings.finite_rings
[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:

sage: # needs sage.libs.singular
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,
 2*a*b + 2*b*c + 2*c*d - b,
 b^2 + 2*a^2*c + 2*b^2*d - c]
sage: F = Sequence(I)
sage: A, v = F.coefficient_matrix()
sage: A

[  0  0  0  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]
[  0  2  0  0  2  0  0  2  0  0  126  0  0  0]
[  0  0  1  2  0  0  2  0  0  0  0  126  0  0]
sage: v

[a^2]
[a*b]
[b^2]
[a*c]
[b*c]
[c^2]
Polynomials, Release 10.1

\[ \begin{align*}
& b^d \\
& c^d \\
& d^2 \\
& a \\
& b \\
& c \\
& d \\
& 1 \\
\end{align*} \]

**sage:**

\[
\text{A}^\text{v} \\
[ \begin{align*}
& a + 2b + 2c + 2d - 1 \\
& a^2 + 2b^2 + 2c^2 + 2d^2 - a \\
& 2ab + 2bc + 2cd - b \\
& b^2 + 2ac + 2bd - c \\
\end{align*} ]
\]

**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:

```
sage: # needs sage.rings.polynomial.pbori
sage: sr = mq.SR(2, 4, 4, 8, gf2=True, polybori=True)
sage: while True:
    try:
        F, s = sr.polynomial_system()
        break
    except ZeroDivisionError:
        pass
sage: Fz = Sequence(F.part(2))
sage: Fz.connected_components()
[Polynomial Sequence with 128 Polynomials in 128 Variables,
 Polynomial Sequence with 128 Polynomials in 128 Variables,
 Polynomial Sequence with 128 Polynomials in 128 Variables,
 Polynomial Sequence with 128 Polynomials in 128 Variables]
```

**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: # needs sage.rings.polynomial.pbori
sage: B.<x,y,z> = BooleanPolynomialRing()
sage: F = Sequence([x*y + y + 1, z + 1])
sage: G = F.connection_graph(); G
Graph on 3 vertices
sage: G.is_connected()
False
sage: F = Sequence([x])
sage: F.connection_graph()
Graph on 1 vertex
```

**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: # needs sage.rings.polynomial.pbori
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: # needs sage.rings.polynomial.pbori
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F, s = sr.polynomial_system()
sage: P = F.ring()
sage: I = F.ideal()
sage: J = I.elimination_ideal(P.gens()[4:-4])
sage: J <= I
True
sage: set(J.gens().variables()).issubset(P.gens()[:4] + P.gens()[-4:])
True
```

`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 $Syz(LM(I))$:

$$S * G = \sum_{i=0}^{m} h_i g_i \succ G 0.$$ 

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: 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: sr = mq.SR(allow_zero_inversions=True)  # needs sage.rings.polynomial.pbori
sage: F, s = sr.polynomial_system()           # needs sage.rings.polynomial.pbori
sage: len(F.monomials())                      # needs sage.rings.polynomial.pbori
49
```

**nmonomials()**

Return the number of monomials present in this system.

**EXAMPLES:**

```
sage: sr = mq.SR(allow_zero_inversions=True)  # needs sage.rings.polynomial.pbori
sage: F, s = sr.polynomial_system()           # needs sage.rings.polynomial.pbori
sage: F.nmonomials()                          # needs sage.rings.polynomial.pbori
49
```

**nparts()**

Return number of parts of this system.

**EXAMPLES:**

```
sage: sr = mq.SR(allow_zero_inversions=True)  # needs sage.rings.polynomial.pbori
sage: F, s = sr.polynomial_system()           # needs sage.rings.polynomial.pbori
sage: F.nparts()                              # needs sage.rings.polynomial.pbori
4
```

**nvariables()**

Return number of variables present in this system.

**EXAMPLES:**

```
sage: sr = mq.SR(allow_zero_inversions=True)  # needs sage.rings.polynomial.pbori
sage: F, s = sr.polynomial_system()           # needs sage.rings.polynomial.pbori
sage: F.nvariables()                          # needs sage.rings.polynomial.pbori
4
```
part(i)

Return i-th part of this system.

EXAMPLES:

```python
sage: # needs sage.rings.polynomial.pbori
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:

```python
sage: # needs sage.rings.polynomial.pbori
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 \((f_1, \ldots, f_n)\) then 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 \(\{g_1, \ldots, g_i-1, g_i+1, \ldots, g_s\}\)
- \(LC(g_i) = 1\) for all \(i\) if the coefficient ring is a field.

EXAMPLES:

```python
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:

```python
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
sage: I.basis.reduced()
[x]
```

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')       # needs sage.rings.polynomial.pbori
sage: F, s = sr.polynomial_system()                                           # needs sage.rings.polynomial.pbori
sage: print(F.ring().repr_long())                                            # needs sage.rings.polynomial.pbori
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
```

**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:**

```python
sage: sr = mq.SR(allow_zero_inversions=True)  # needs sage.rings.polynomial.pbori
sage: F, s = sr.polynomial_system(); F  # needs sage.rings.polynomial.pbori
Polynomial Sequence with 40 Polynomials in 20 Variables
sage: F = F.subs(s); F  # needs sage.rings.polynomial.pbori
Polynomial Sequence with 40 Polynomials in 16 Variables
```

### `universe()`

Return the polynomial ring all elements live in.

**EXAMPLES:**

```python
sage: sr = mq.SR(allow_zero_inversions=True, gf2=True, order='block')  # needs sage.rings.polynomial.pbori
sage: F, s = sr.polynomial_system()  # needs sage.rings.polynomial.pbori
sage: print(F.ring().repr_long())  # needs sage.rings.polynomial.pbori
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:**

```python
sage: sr = mq.SR(allow_zero_inversions=True)  # needs sage.rings.polynomial.pbori
sage: F, s = sr.polynomial_system()  # needs sage.rings.polynomial.pbori
sage: F.variables()[10:]  # needs sage.rings.polynomial.pbori
(k003, k002, k001, k000, s003, s002, s001, s000, w103, w102)
```

```python
class sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_gf2(parts, ring, immutable=False, cr=False, cr_str=None)
```

**Bases:** `PolynomialSequence_generic`

Polynomial Sequences over \( F_2 \).
eliminate_linear_variables(maxlength=+Infinity, 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 maxlen.

INPUT:

• maxlen - an optional upper bound on the number of monomials by which a variable is replaced. If maxlen=+Infinity then no condition is checked. (default: +Infinity).

• 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 than what is implemented here, it is less flexible (skip is not supported) and may increase the degree (default: False)

OUTPUT:

With return_reductors=True, 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).

With return_reductors=False, only the first sequence is returned.

EXAMPLES:

sage: # needs sage.rings.polynomial.pbori
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:

sage: # needs sage.rings.polynomial.pbori
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:
Note: This is called “massaging” in [BCJ2007].

reduced()

If this sequence is \( f_1, ..., f_n \), return \( g_1, ..., g_s \) such that:

- \((f_1, ..., f_n) = (g_1, ..., g_s)\)
- \(LT(g_i) \neq LT(g_j)\) for all \( i \neq j \)
- \(LT(g_i)\) does not divide \( m\) for all monomials \( m\) of \( g_1, ..., g_{i-1}, g_{i+1}, ..., g_s\)

EXAMPLES:

```
sage: # needs sage.rings.polynomial.pbori
sage: sr = mq.SR(1, 1, 1, 4, gf2=\text{True}, polybori=\text{True})
```

```
sage: while True: # workaround (see :trac:`31891`
    ....: try:
    ....:     F, s = sr.polynomial_system()
    ....:     break
    ....: except ZeroDivisionError:
    ....:     pass
sage: g = F.reduced()
sage: len(g) == len(set(gi.lt() \text{ for } gi \text{ in } g))
```

True

```
sage: for i in range(len(g)):
    ....:     for j in range(len(g)):
    ....:         if i == j:
    ....:             continue
    ....:     for t in list(g[j]):
    ....:         assert g[i].lt() not in t.divisors()
```

solve(algorithm='\text{polybori}', \text{n=1, eliminate_linear_variables=}\text{True, verbose=}\text{False, **kwds})

Find solutions of this boolean polynomial system.

This function provide 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:

- \text{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 == +Infinity then all solutions are returned. If n < ∞ 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:

```sage
sage: # needs sage.rings.polynomial.pbori
sage: from sage.doctest.fixtures import reproducible_repr
sage: R.<x,y,z> = BooleanPolynomialRing()
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}]
sage: S.subs(sol[0])
# needs sage.rings.polynomial.pbori
[0, 0, 0]
```

We check that it is actually a solution:

```sage
sage: S.subs(sol[0])
# needs sage.rings.polynomial.pbori
[0, 0, 0]
```

We obtain all solutions:

```sage
sage: sol = S.solve(n=Infinity)
# needs sage.rings.polynomial.pbori
sage: print(reproducible_repr(sol))
# needs sage.rings.polynomial.pbori
[[x: 0, y: 1, z: 0], {x: 1, y: 1, z: 1}]
sage: S.subs(sol[0])
# needs sage.rings.polynomial.pbori
[0, 0, 0]
```

We can force the use of exhaustive search if the optional package FES is present:

```sage
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])
# optional - fes
[0, 0, 0]
```

And we may use SAT-solvers if they are available:
sage: sol = S.solve(algorithm='sat') # optional - pycryptosat
˓→needs sage.rings.polynomial.pbori
sage: print(reproducible_repr(sol)) # optional - pycryptosat
˓→needs sage.rings.polynomial.pbori
{[x: 0, y: 1, z: 0]}
sage: S.subs(sol[0]) # optional - pycryptosat
˓→needs sage.rings.polynomial.pbori
[0, 0, 0]

class sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_gf2e(parts, ring, immutable=False, cr=False, cr_str=None)

Bases: PolynomialSequence_generic

PolynomialSequence over \( F_2^e \), i.e extensions over \( F_2 \).

weil_restriction()

Project this polynomial system to \( F_2 \).

That is, compute the Weil restriction of scalars for the variety corresponding to this polynomial system and express it as a polynomial system over \( F_2 \).

EXAMPES:

```sage
# needs sage.rings.finite_rings
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
# needs sage.rings.polynomial.pbori
sage: sr = mq.SR(1, 1, 1, 4, gf2=False)
sage: while True:
  # workaround (see :trac:`31891`)
  try:
    F, s = sr.polynomial_system()
  except ZeroDivisionError:
    pass
sage: 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:

```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
```

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,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'); 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)  # needs sage.rings.number_field
sage: P.<x,y> = PolynomialRing(K, 2)  # needs sage.rings.number_field
sage: 1/2*s*x^2 + 3/4*s
```

(continues on next page)
\( \frac{1}{2} x^2 + \frac{3}{4} s \)

```
sage: P.<x,y,z> = ZZ[]; P
Multivariate Polynomial Ring in x, y, z over Integer Ring
```

```
sage: P.<x,y,z> = Zmod(2^100); 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)
class 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomialRing_libsingular'
```

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)
x^10 + 2*y^5
```

```
sage: (x^5 + 2*y^5)^3
x^15 + x^10*y^5 + 2*x^5*y^10 - 2*y^15
```

```
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_polydict_with_category'
```

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 $\mathbb{Q}(a)$ and $\mathbb{Z}$ support
• 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, refcount singular rings, bugfixes.

class sage.rings.polynomial.multi_polynomial_libsingular.MPolynomialRing_libsingular

Bases: MPolynomialRing_base

Construct a multivariate polynomial ring subject to the following conditions:

INPUT:

• base_ring - base ring (must be either $\mathbb{GF}(q)$, $\mathbb{ZZ}$, $\mathbb{ZZ}/n\mathbb{ZZ}$, $\mathbb{QQ}$ 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[]; 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'); 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: 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

(continues on next page)
Multivariate Polynomial Ring in $x, y, z$ over Ring of integers modulo 1024

```sage
sage: P.<x,y,z> = Zmod(3^10)[]; P
Multivariate Polynomial Ring in $x, y, z$ over Ring of integers modulo 59049
```

```sage
sage: P.<x,y,z> = Zmod(2^100)[]; P
Multivariate Polynomial Ring in $x, y, z$ over
Ring of integers modulo 126765060028229401496703205376
```

```sage
sage: P.<x,y,z> = Zmod(2521352)[]; P
Multivariate Polynomial Ring in $x, y, z$ over Ring of integers modulo 2521352
```

```sage
sage: type(P)
<class 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomialRing_libsingular'>
```

```sage
sage: P.<x,y,z> = Zmod(25213521351515232)[]; P
Multivariate Polynomial Ring in $x, y, z$ over Ring of integers modulo 25213521351515232
```

```sage
sage: type(P)
<class 'sage.rings.polynomial.multi_polynomial_ring.MPolynomialRing_polydict_with_category'>
```

```sage
sage: P.<x,y,z> = PolynomialRing(Integers(2^32), order='lex')
sage: P(2^32-1)
4294967295
```

**Element**

alias of `MPolynomial_libsingular`

**gen**($n=0$)

Returns the $n$-th generator of this multivariate polynomial ring.

**INPUT:**

- **$n$** = an integer $\geq 0$

**EXAMPLES:**

```sage
sage: P.<x,y,z> = QQ[]
sage: P.gen(), P.gen(1)
(x, y)
```

```sage
sage: P = PolynomialRing(GF(127), 1000, 'x')
sage: P.gen(500)
x500
```

```sage
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)  # needs sage.rings.function_field
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:**

```
sage: P.<x,y,z> = QQ[

sage: P.monomial_lcm(3/2*x*y,x)

x*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:**

```
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 $f/g$, 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:**

```
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 $Z$ behaves different if coeff=True:

```
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):
```

(continues on next page)
...  
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:**

```python
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:**

```python
sage: P.<x,y> = QQ[]
sage: P.ngens()  
2
sage: k.<a> = GF(2^16)  
# needs sage.rings.finite_rings
sage: P = PolynomialRing(k, 1000, 'x')  
# needs sage.rings.finite_rings
sage: P.ngens()  
1000
```

**class sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular**

**Bases:** **MPolynomial_libsingular**

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:

```python
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:

```python
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.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:

```python
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.

```python
sage: f.coefficient(x^0) # outputs the full polynomial
x^2*y^2 + x^2*y + x*y^2 + x^2 + x*y + y^2 + x + y + 1
```

(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:

```python
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:

```python
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:

```python
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.

```python
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.

```python
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.

```python
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`.

```python
sage: f.degree(std_grading=True)
5
```
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:

\[
\text{sage: } R.<y0,y1,y2> = PolynomialRing(QQ,3)
\text{sage: } q = 3*y0*y1*y1*y2; q
3*y0*y1^2*y2
\text{sage: } q.degrees()
(1, 2, 1)
\text{sage: } (q + y0^5).degrees()
(5, 2, 1)
\]

\text{dict()}

Return a dictionary representing self. This dictionary is in the same format as the generic \texttt{MPolynomial}: The dictionary consists of \texttt{ETuple:coefficient} pairs.

EXAMPLES:

\[
\text{sage: } R.<x,y,z> = QQ[
\text{sage: } f=2*x*y^3*z^2 + 1/7*x^2 + 2/3
\text{sage: } f.dict()
{(0, 0, 0): 2/3, (1, 3, 2): 2, (2, 0, 0): 1/7}
\]

\text{divides}(\text{other})

Return True if this polynomial divides \text{other}.

EXAMPLES:

\[
\text{sage: } R.<x,y,z> = QQ[
\text{sage: } p = 3*x*y + 2*y*z + x*z
\text{sage: } q = x + y + z + 1
\text{sage: } r = p * q
\text{sage: } p.divides(r)
\text{True}
\text{sage: } q.divides(p)
\text{False}
\text{sage: } r.divides(0)
\text{True}
\text{sage: } R.zero().divides(r)
\text{False}
\text{sage: } R.zero().divides(0)
\text{True}
\]

\text{exponents}(\text{as\_ETuples=}\text{True})

Return the exponents of the monomials appearing in this polynomial.

INPUT:

\begin{itemize}
\item \texttt{as\_ETuples} – (default: \texttt{True}) if \texttt{True} returns the result as an list of \texttt{ETuples}, otherwise returns a list of tuples
\end{itemize}

EXAMPLES:
```
\begin{verbatim}
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)]
sage: f.exponents(as_ETuples=False)
[(3, 0, 0), (0, 2, 0), (0, 1, 0)]
sage: factor(proof=None)
Return the factorization of this polynomial.

INPUT:

• proof – ignored.

EXAMPLES:

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(); F
x * (x^2 + x + 1) * (x^2 + 2*y^2)

Next we factor the same polynomial, but over the finite field of order 3.:

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
(-1) * x * (-x + y) * (x + y) * (x - 1)^2

Next we factor a polynomial, but over a finite field of order 9.:

sage: # needs sage.rings.finite_rings
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(); F
((-a)) * x * (x - 1)^2 * ((-a + 1)*x^2 + y^2)
sage: f - F
0

Next we factor a polynomial over a number field.:

sage: # needs sage.rings.number_field
sage: p = polygen(ZZ, '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}
```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^2 + 1)
sage: R.<y, z> = PolynomialRing(K)
sage: f = 2*y^2 + 2*z^2
sage: F = f.factor(); F.unit()
2

Another example:

sage: R.<x,y,z> = GF(32003)[]
# needs sage.rings.finite_rings
sage: f = 9*(x-1)^2(y+z)
# needs sage.rings.finite_rings
sage: f.factor()
# needs sage.rings.finite_rings
(9) * (y + z) * (x - 1)^2

sage: R.<x,w,v,u> = QQ[
# needs sage.rings.finite_rings
sage: p = (4*v^4*u^2 - 16*v^2*u^4 + 16*u^6 - 4*v^4*u + 8*v^2*u^3 + v^4)

sage: p.factor()
(-2*v^2*u + 4*u^3 + v^2)^2

sage: R.<a,b,c,d> = QQ[
# needs sage.rings.finite_rings
sage: f = (-2) * (a - d) * (-a + b) * (b - d) * (a - c) * (b - c) * (c - d)
sage: F = f.factor(); F
(-2) * (c - d) * (-b + c) * (b - d) * (-a + c) * (-a + b) * (a - d)
sage: F[0][0]
c - d
sage: F.unit()
-2

Constant elements are factorized in the base rings.

sage: P.<x,y> = ZZ[
# needs sage.rings.finite_rings
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

sage: q = 1073741789
sage: T.<aa, bb> = PolynomialRing(GF(q))
# needs sage.rings.finite_rings
sage: f = aa^2 + 12124343*bb*aa + 32434598*bb^2
# needs sage.rings.finite_rings
sage: f.factor()
# needs sage.rings.finite_rings
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 github issue #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

```python
sage: R.<x,y> = PolynomialRing(Zmod(4))
sage: f = (2*x + 1) * (x^2 + x + 1)
sage: f.factor()
Traceback (most recent call last):
  ... Not Implemented Error: Factorization of multivariate polynomials over Ring of integers modulo 4 is not implemented.
```

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:

```python
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:

```python
sage: # needs sage.rings.finite_rings
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)
```
We compute a gcd over a number field:

```
sage: # needs sage.rings.number_field
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
```

**global_height** *(prec=None)*

Return the (projective) global height of the polynomial.

This returns the absolute logarithmic height of the coefficients thought of as a projective point.

**INPUT:**

- *prec* – desired floating point precision (default: default `RealField` precision).

**OUTPUT:** a real number.

**EXAMPLES:**

```
sage: R.<x,y> = PolynomialRing(QQ)
sage: f = 3*x^3 + 2*x*y^2
sage: exp(f.global_height())
3.00000000000000
```

```
sage: # needs sage.rings.number_field
sage: K.<k> = CyclotomicField(3)
sage: R.<x,y> = PolynomialRing(K, sparse=True)
sage: f = k*x*y + 1
sage: exp(f.global_height())
1.00000000000000
```

Scaling should not change the result:

```
sage: R.<x,y> = PolynomialRing(QQ)
sage: f = 1/25*x^2 + 25/3*x*y + y^2
sage: f.global_height()
6.43775164973640
sage: g = 100 * f
sage: g.global_height()
6.43775164973640
```
sage: R.<x> = PolynomialRing(QQ)
sage: K.<k> = NumberField(x^2 + 5)
sage: T.<t,w> = PolynomialRing(K)
sage: f = 1/1331 * t^2 + 5 * w + 7
sage: f.global_height() #
needs sage.symbolic
9.13959596745043

sage: R.<x,y> = QQ[]
sage: f = 1/123*x*y + 12
sage: f.global_height(prec=2) #
needs sage.symbolic
8.0

sage: R.<x,y> = QQ[]
sage: f = 0*x*y
sage: f.global_height()
0.000000000000000

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)

Integrate this polynomial with respect to the provided variable.

One requires that \( Q \) is contained in the ring.

3.1. Multivariate Polynomials and Polynomial Rings 453
INPUT:

- variable – the integral is taken with respect to variable

EXAMPLES:

```python
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*x*y^2 + 3/2*x^2 + 2*x
sage: f.integral(y)
x^3*y^3 + 5/3*y^3 + 3*x*y + 2*y
```

Check that github issue #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:

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:

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:

sage: P.<x,y,z> = PolynomialRing(QQ)
sage: x.is_term()
True
sage: (2*x).is_term()
Polynomials, Release 10.1

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:

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)
True

is_zero()
Return True if this polynomial is zero.

EXAMPLES:

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:

sage: R.<a,b,c> = QQ[]
sage: f = a*c^3 + a^2*b + 2*b^4
sage: list(f.iterator_exp_coeff())
[((0, 4, 0), 2), ((1, 0, 3), 1), ((2, 1, 0), 1)]

sage: list(f.iterator_exp_coeff(as_ETuples=False))
[((0, 4, 0), 2), ((1, 0, 3), 1), ((2, 1, 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), 1), ((1, 0, 3), 1), ((0, 4, 0), 2)]

\texttt{lcm()}\hspace{1cm}

Leading coefficient of this polynomial with respect to the term order of \texttt{self.parent()}.

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.lc()
3

sage: f = 5*x^3*y^2*z^4 + 4*x^3*y^2*z^1
sage: f.lc()
5

\texttt{lcm(g)}\hspace{1cm}

Return the least common multiple of \texttt{self} and \texttt{g}.

EXAMPLES:

sage: P.<x,y,z> = QQ[]
sage: p = (x+y)*(y+z)
sage: q = (z^4+2)*(y+z)
sage: lcm(p,q)
x*y*z^4 + y^2*z^4 + x*z^5 + y*z^5 + 2*x*y + 2*y^2 + 2*x*z + 2*y*z

sage: P.<x,y,z> = ZZ[]
sage: p = 2*(x+y)*(y+z)
sage: q = 3*(z^4+2)*(y+z)
sage: lcm(p,q)
6*x*y*z^4 + 6*y^2*z^4 + 6*x*z^5 + 6*y*z^5 + 12*x*y + 12*y^2 + 12*x*z + 12*y*z

sage: # needs sage.rings.finite_rings
sage: r.<x,y> = PolynomialRing(GF(2**8, 'a'), 2)
sage: a = r.base_ring().0
sage: f = (a^2+a)*x^2*y + (a^4+a^3+a)*y + a^5
sage: f.lcm(x^4)
(a^2 + a)*x^6*y + (a^3 + a - 1)*x^4*y + (-a)*x^4

sage: # needs sage.rings.number_field
sage: w = polygen(ZZ, 'w')
sage: r.<x,y> = PolynomialRing(NumberField(w^4 + 1, 'a'), 2)
sage: a = r.base_ring().0
sage: f = (a^2+a)*x^2*y + (a^4+a^3+a)*y + a^5
sage: f.lcm(x^4)
(a^2 + a)*x^6*y + (a^3 + a - 1)*x^4*y + (-a)*x^4

\texttt{lift(I)}\hspace{1cm}

given an ideal \texttt{I = (f_1,\ldots,f_r)} and some \texttt{g (== self) in I}, find \texttt{s_1,\ldots,s_r} such that \texttt{g = s_1 f_1 + \ldots + s_r f_r.}
A `ValueError` exception is raised if \( g \) (== `self`) does not belong to \( I \).

**EXAMPLES:**

```
sage: A.<x,y> = PolynomialRing(QQ,2,order='degrevlex')
sage: I = A.ideal([x^10 + x^9*y^2, y^8 - x^2*y^7])
sage: f = x^10 + 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
```

Check that [github issue #13671](https://github.com/sagemath/sage/issues/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]
```

\(\text{lm}()\)

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 = x^3*y^2*z^4 + x^3*y^2*z^1
sage: f.lm()
x^3*y^2*z^4
sage: R.<x,y,z> = PolynomialRing(QQ, 3, order='deglex')
sage: f = x^1*y^2*z^3 + x^3*y^2*z^0
sage: f.lm()
x*y^2*z^3
sage: f = x^1*y^2*z^4 + x^1*y^1*z^5
sage: f.lm()
x*y^2*z^4
sage: R.<x,y,z> = PolynomialRing(GF(127), 3, order='degrevlex')
sage: f = x^1*y^5*z^2 + x^4*y^1*z^3
sage: f.lm()
x*y^5*z^2
sage: f = x^4*y^7*z^1 + x^4*y^2*z^3
```

(continues on next page)
local_height\((v, \text{prec}=\text{None})\)

Return the maximum of the local height of the coefficients of this polynomial.

**INPUT:**
- \(v\) – a prime or prime ideal of the base ring.
- \text{prec} – desired floating point precision (default: default RealField precision).

**OUTPUT:** a real number.

**EXAMPLES:**

```
sage: R.<x,y> = PolynomialRing(QQ)
sage: f = 1/1331*x^2 + 1/4000*y^2
sage: f.local_height(1331)
7.19368581839511
```

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: K.<k> = NumberField(x^2 - 5)
sage: T.<t,w> = K[]
sage: I = K.ideal(3)
sage: f = 1/3*t*w + 3
sage: f.local_height(I)
1.09861228866811
```

```
sage: R.<x,y> = QQ[]
sage: f = 1/2*x*y + 2
sage: f.local_height(2, prec=2)
0.75
```

local_height_arch\((i, \text{prec}=\text{None})\)

Return the maximum of the local height at the \(i\)-th infinite place of the coefficients of this polynomial.

**INPUT:**
- \(i\) – an integer.
- \text{prec} – desired floating point precision (default: default RealField precision).

**OUTPUT:** a real number.

**EXAMPLES:**

```
sage: R.<x,y> = PolynomialRing(QQ)
sage: f = 210*x*y
sage: f.local_height_arch(0)
5.34710753071747
```

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: K.<k> = NumberField(x^2 - 5)
sage: f = 210*x*y
sage: f.local_height_arch(0)
5.34710753071747
```

(continues on next page)
sage: T.<t,w> = K[
] sage: f = 1/2*t*w + 3
sage: f.local_height_arch(1, prec=52)
1.09861228866811

sage: R.<x,y> = QQ[
] sage: f = 1/2*x*y + 3
sage: f.local_height_arch(0, prec=2)
1.0

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

sage: f = 5*x^3*y^2*z^4 + 4*x^3*y^2*z^1
sage: f.lt()
-2*x^3*y^2*z^4

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)
polynomials()

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*z^2 + 2/3
sage: f.monomials()
[y*z^2, x, 1]
sage: f = P(3/2)
sage: f.monomials()
[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.

\begin{verbatim}
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
\end{verbatim}

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

\begin{verbatim}
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()
3*x^19 - 34*y + 17
sage: f == f.numerator()
False
sage: f.numerator().base_ring()
Integer Ring
\end{verbatim}

We check that the computation of numerator and denominator is valid.

\begin{verbatim}
sage: K=QQ['x,y']
sage: f=K.random_element()
sage: f.numerator() / f.denominator() == f
True
\end{verbatim}

The following tests against a bug fixed in github issue #11780:

\begin{verbatim}
sage: P.<foo,bar> = ZZ[]
sage: Q.<foo,bar> = QQ[]
sage: f = Q.random_element()
sage: f.numerator().parent() is P
True
\end{verbatim}

\section*{nvariables()}

Return the number variables in this polynomial.

EXAMPLES:

\begin{verbatim}
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
\end{verbatim}

\section*{quo_rem(right)}

Return quotient and remainder of self and right.

EXAMPLES:
\texttt{sage: R.<x,y> = QQ[]}  
\texttt{sage: f = y^3x^2 + x + 1}  
\texttt{sage: f.quo_rem(x)}  
\hspace{1em}(x^3y + 1, 1)  
\texttt{sage: f.quo_rem(y)}  
\hspace{1em}(x^2, x + 1)  
\texttt{sage: R.<x,y> = ZZ[]}  
\texttt{sage: f = 2^*y^3x^2 + x + 1}  
\texttt{sage: f.quo_rem(x)}  
\hspace{1em}(2x^3y + 1, 1)  
\texttt{sage: f.quo_rem(y)}  
\hspace{1em}(2x^2, x + 1)  
\texttt{sage: f.quo_rem(3*x)}  
\hspace{1em}(0, 2x^2y^2 + x + 1)  

\textbf{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:**

\texttt{sage: P.<x,y,z> = QQ[]}  
\texttt{sage: f1 = -2 * x^2 + x^3}  
\texttt{sage: f2 = -2 * y + x* y}  
\texttt{sage: f3 = -x^2 + y^2}  
\texttt{sage: F = Ideal([f1,f2,f3])}  
\texttt{sage: g = x^3y - 3^*x^2y^2}  
\texttt{sage: g.reduce(F)}  
\hspace{1em}-6^*y^2 + 2^*y}  
\texttt{sage: g.reduce(F.gens())}  
\hspace{1em}-6^*y^2 + 2^*y}  

\(Z\) is also supported.
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
go.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])
x

The reduction is not canonical when I is not a Groebner basis:

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 provided the first variable of the parent is chosen.

INPUT:

• other - polynomial
• variable - optional variable (default: None)

EXAMPLES:

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:

sage: R.<x,y,z> = PolynomialRing(GF(32003), 3)
# needs sage.rings.finite_rings
sage: f = 3 *(x+2)^3 + y
sage: g = x + y + z
# needs sage.rings.finite_rings
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
Polynomials, Release 10.1

Resultants are also supported over the Integers:

```
sage: R.<x,y,a,b,u> = PolynomialRing(ZZ, 5, order='lex')
sage: r = (x^4*y^2 + x^2*y - y).resultant(x*y - y*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 -
3*b^2*u - 4*y^4*a^2*b^2 + 6*y^4*a^2*u^2 - 2*y^4*a*u + 4*y^4*b - 4*y^3*a^3*b^3 -
12*y^3*a^3*b^2*u^2 + 6*y^3*a^3*b*u^2 - 12*y^3*a^2*b^2*u^2 + 6*y^3*a^2*b*u^2 - 4*y^3*
a^2*u^2 - 3 - 6*y^3*b^2 + y^3*u^2 + y^2*a^4*b^4 + 4*y^2*a^3*b^3 - y^2*a^2*b^3 + 6*y^2*
-2*a^2*b^2*u^2 + 6*y^2*a*b^2*u^2 + 4*y^2*a*b*u^3 + 6*y^2*b^3 - 2*y^2*b*u^2 + y^2
-2*u^4 + y*a^2*b^4 + 2*y*a*b^3*u - y*b^4 + y*b^2*u^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:**

```
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:**

```
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
```

(continues on next page)
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:

sage: R.<x,y>=QQ[]
sage: g=x+y
sage: 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

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^3*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

(continues on next page)
A matrix term ordering changes the grading. To get the total degree using the standard grading, use \code{std_grading=True}:

```python
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 \code{ValueError} exception is raised. This is checked using the \code{is_univariate()} method. The new Polynomial is over the same base ring as the given \code{MPolynomial} and in the variable \(x\) if no ring \(R\) is provided.

**EXAMPLES:**

```python
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):
  ... 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
```

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 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:

```python
sage: P.<x,y,z> = GF(2)[]
```

```python
sage: f = x*z^2 + z + 1
```

```python
sage: f.variables()
(x, z)
```

```python
sage: f.variable(1)
z
```

```
variables()

Return a tuple of all variables occurring in self.
```

EXAMPLES:

```python
sage: P.<x,y,z> = GF(2)[]
```

```python
sage: f = x*z^2 + z + 1
```

```python
sage: f.variables()
(x, z)
```

```python
sage.rings.polynomial.multi_polynomial_libsingular.unpickle_MPolynomialRing_libsingular(base_ring, names, term_order)
```

inverse function for MPolynomialRing_libsingular.__reduce__

EXAMPLES:

```python
sage: P.<x,y> = PolynomialRing(QQ)
```

```python
sage: loads(dumps(P)) is P # indirect doctest
True
```

```python
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)
```

```python
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:
We compute a Groebner basis for cyclic 6, which is a standard benchmark and test ideal:

```sage
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:

  ```sage
  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:

  ```sage
  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.interred_libsingular(I)`

SINGULAR’s `interred()` command.

**INPUT:**
- `I` – a Sage ideal

**EXAMPLES:**

```sage
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
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:**
Polynomials, Release 10.1

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:

```python
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.slimb_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 Solution of polynomial systems using msolve

msolve is a multivariate polynomial system solver based on Gröbner bases.

This module provide implementations of some operations on polynomial ideals based on msolve.

Note that the optional package msolve must be installed.

See also:
- sage.features.msolve
- sage.rings.polynomial.multi_polynomial_ideal

sage.rings.polynomial.msolve.groebner_basis_degrevlex(ideal)
Compute a degrevlex Gröbner basis using msolve

EXAMPLES:

```python
sage: from sage.rings.polynomial.msolve import groebner_basis_degrevlex
sage: R.<a,b,c> = PolynomialRing(GF(101), 3, order='lex')
sage: I = sage.rings.ideal.Katsura(R,3)
sage: gb = groebner_basis_degrevlex(I); gb  # optional - msolve
[a + 2*b + 2*c - 1, b*c - 19*c^2 + 10*b + 40*c, b^2 - 41*c^2 + 20*b - 20*c, c^3 + 28*c^2 - 37*b + 13*c]
sage: gb.universe() is R  # optional - msolve
False
```

(continues on next page)
sage: gb.universe().term_order()  # optional - msolve
Degree reverse lexicographic term order
sage: ideal(gb).transformed_basis(other_ring=R)  # optional - msolve
[c^4 + 38*c^3 - 6*c^2 - 6*c, 30*c^3 + 32*c^2 + b - 14*c, 
a + 2*b + 2*c - 1]

sage.rings.polynomial.msolve.variety(ideal, ring, proof)
Compute the variety of a zero-dimensional ideal using msolve.
Part of the initial implementation was loosely based on the example interfaces available as part of msolve, with the authors’ permission.

EXAMPLES:

sage: from sage.rings.polynomial.msolve import variety
sage: p = 536870909
sage: R.<x, y> = PolynomialRing(GF(p), 2, order='lex')

sage: I = Ideal([x*y - 1, (x-2)^2 + (y-1)^2 - 1])

sage: sorted(variety(I, GF(p^2), proof=False), key=str)  # optional - msolve
[{x: 1, y: 1},
 {x: 254228855*z2 + 114981228, y: 232449571*z2 + 402714189},
 {x: 267525699, y: 473946006},
 {x: 282642054*z2 + 154363985, y: 304421338*z2 + 197081624}]

### 3.1.11 Generic data structures for multivariate polynomials

This module provides an implementation of a generic data structure `PolyDict` and the underlying arithmetic for multivariate polynomial rings. It uses a sparse representation of polynomials encoded as a Python dictionary where keys are exponents and values coefficients.

\{(e1, \ldots, er): c1, \ldots\} \leftrightarrow c1 \cdot x1^{e1} \cdot \ldots \cdot xr^{er} + \ldots

The exponent \((e1, \ldots, er)\) in this representation is an instance of the class `ETuple`.

**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).
**Polynomials, Release 10.1**

**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:**

```sage
from sage.rings.polynomial.polydict import ETuple
e = ETuple([-2, 1, -5, 3, 1, 0])
f = ETuple([1, -3, -3, 4, 0, 2])
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:**

```sage
from sage.rings.polynomial.polydict import ETuple
e = ETuple([1, 0, 2])
f = ETuple([0, 0, 1])
e.common_nonzero_positions(f)
{0, 2}
e.common_nonzero_positions(f, sort=True)
[0, 2]
```

**divide_by_gcd**(other)

Return self / gcd(self, other).

The entries of the result are the maximum of 0 and the difference of the corresponding entries of self and other.

**divide_by_var**(pos)

Return division of self by the variable with index pos.

If self[pos] == 0 then a ArithmeticError is raised. Otherwise, an ETuple is returned that is zero in position pos and coincides with self in the other positions.

**EXAMPLES:**

```sage
from sage.rings.polynomial.polydict import ETuple
e = ETuple([1, 2, 0, 1])
e.divide_by_var(0)
(0, 2, 0, 1)
e.divide_by_var(1)
(1, 1, 0, 1)
e.divide_by_var(3)
(1, 2, 0, 0)
e.divide_by_var(2)
Traceback (most recent call last):
...
ArithmeticError: not divisible by this variable
```

**divides**(other)

Return whether self divides other, i.e., no entry of self exceeds that of other.

**EXAMPLES:**
```
sage: from sage.rings.polynomial.polydict import ETuple
sage: ETuple([1, 1, 0, 1, 0]).divides(ETuple([2, 2, 2, 2, 2]))
True
sage: ETuple([0, 3, 0, 1, 0]).divides(ETuple([2, 2, 2, 2, 2]))
False
sage: ETuple([0, 3, 0, 1, 0]).divides(ETuple([0, 3, 2, 2, 2]))
True
sage: ETuple([0, 0, 0, 0, 0]).divides(ETuple([2, 2, 2, 2, 2]))
True
False
True
```

**dotprod**(*other*)

Return the dot product of this tuple by *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
```

**eadd**(*other*)

Return the 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.eadd(f)
(1, 1, 3)
```

Verify that github issue #6428 has been addressed:

```
sage: # needs sage.libs.singular
sage: R.<y, z> = Frac(QQ['x'])[]
sage: type(y)
<class 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular'>
sage: y^(2^32)
Traceback (most recent call last):
...```
OverflowError: exponent overflow (...) # 64-bit
OverflowError: Python int too large to convert to C unsigned long # 32-bit

\textbf{eadd\_p(}other, pos\textbf{)}

Add other to self at position pos.

EXAMPLES:

\begin{verbatim}
sage: from sage.rings.polynomial.polydict import ETuple
e = ETuple([1, 0, 2])
sage: e.eadd_p(5, 1)
(1, 5, 2)
sage: e = ETuple([0]*7)
sage: e.eadd_p(5, 4)
(0, 0, 0, 0, 5, 0, 0)
sage: ETuple([0,1]).eadd_p(1, 0) == ETuple([1,1])
True
sage: e = ETuple([0, 1, 0])
sage: e.eadd_p(0, 0).nonzero_positions()
[1]
sage: e.eadd_p(0, 1).nonzero_positions()
[1]
sage: e.eadd_p(0, 2).nonzero_positions()
[1]
\end{verbatim}

\textbf{eadd\_scaled(}other, scalar\textbf{)}

Vector addition of self with scalar * other.

EXAMPLES:

\begin{verbatim}
sage: from sage.rings.polynomial.polydict import ETuple
e = ETuple([1, 0, 2])
f = ETuple([0, 1, 1])
sage: e.eadd_scaled(f, 3)
(1, 3, 5)
\end{verbatim}

\textbf{emax(}other\textbf{)}

Vector of maximum of components of self and other.

EXAMPLES:

\begin{verbatim}
sage: from sage.rings.polynomial.polydict import ETuple
e = ETuple([1, 0, 2])
f = ETuple([0, 1, 1])
sage: e.emax(f)
(1, 1, 2)
sage: e = ETuple((1, -2, -2, 4))
f = ETuple((4, 0, 2, 1))
sage: f.emax(e)
(4, 2, 3, 4)
sage: e = ETuple((1, -2, -2, 4))
\end{verbatim}
**emin(other)**

Vector of minimum of components of `self` and `other`.

**EXAMPLES:**

```python
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)
```

**emul(factor)**

Scalar Vector multiplication of `self`.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([1, 0, 2])
sage: e.emul(2)
(2, 0, 4)
```

**escalar_div(n)**

Divide each exponent by `n`.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.polydict import ETuple
sage: ETuple([1, 0, 2]).escalar_div(2)
(0, 0, 1)
sage: ETuple([0, 3, 12]).escalar_div(3)
(0, 1, 4)
sage: ETuple([1, 5, 2]).escalar_div(0)
Traceback (most recent call last):
  ...
ZeroDivisionError
```

**esub(other)**

Vector subtraction of `self` with `other`.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.polydict import ETuple
sage: e = ETuple([1, 0, 2])
```

(continues on next page)
Polynomials, Release 10.1

(continued from previous page)

```python
sage: f = ETuple([0, 1, 1])
sage: 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
sage: e = ETuple([1, 0, 2])
sage: e.is_constant()
False
sage: e = ETuple([0, 0])
sage: e.is_constant()
True
```

**is_multiple_of(n)**
Test whether each entry is a multiple of \( n \).

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.polydict import ETuple
sage: ETuple([0, 0]).is_multiple_of(3)
True
sage: ETuple([0, 3, 12, 0, 6]).is_multiple_of(3)
True
sage: ETuple([0, 0, 2]).is_multiple_of(3)
False
```

**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
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:**
from sage.rings.polynomial.polydict import ETuple

e = ETuple([2, 0, 1])
e.nonzero_values()
[2, 1]
f = ETuple([0, -1, 1])
f.nonzero_values(sort=True)
[-1, 1]

reversed()
Return the reversed ETuple of self.

EXAMPLES:

from sage.rings.polynomial.polydict import ETuple
e = ETuple([1, 2, 3])
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:

from sage.rings.polynomial.polydict import ETuple
e = ETuple([1, 0, 2, 0, 3])
list(e.sparse_iter())
[(0, 1), (2, 2), (4, 3)]

unweighted_degree()
Return the sum of entries.

EXAMPLES:

from sage.rings.polynomial.polydict import ETuple
etuple = ETuple([1, 1, 0, 2, 0]).unweighted_degree()
4
etuple = ETuple([-1, 1]).unweighted_degree()
0

unweighted_quotient_degree(other)
Return the degree of self divided by its gcd with other.
It amounts to counting the non-negative entries of self.esub(other).

weighted_degree(w)
Return the weighted sum of entries.

INPUT:
- w – tuple of non-negative integers

EXAMPLES:

from sage.rings.polynomial.polydict import ETuple
e = ETuple([1, 1, 0, 2, 0])

(continues on next page)
sage: e.weighted_degree((1, 2, 3, 4, 5))
11
sage: ETuple([-1, 1]).weighted_degree((1, 2))
1
sage: ETuple([1, 0]).weighted_degree((1, 2, 3))
Traceback (most recent call last):
...
ValueError: w must be of the same length as the ETuple

weighted_quotient_degree(other, w)

Return the weighted degree of self divided by its gcd with other.

INPUT:
• other – an ETuple
• w – tuple of non-negative integers.

class sage.rings.polynomial.polydict.PolyDict

Bases: object

Data structure for multivariate polynomials.

A PolyDict holds a dictionary all of whose keys are ETuple and whose values are coefficients on which it is implicitly assumed that arithmetic operations can be performed.

No arithmetic operation on PolyDict clear zero coefficients as of now there is no reliable way of testing it in the most general setting, see github issue #35319. For removing zero coefficients from a PolyDict you can use the method remove_zeros() which can be parametrized by a zero test.

apply_map(f)

Apply the map f on the coefficients (inplace).

EXAMPLES:

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]
**coerce_coefficients(A)**

Coerce the coefficients in the parent A

EXAMPLES:

```python
sage: from sage.rings.polynomial.polydict import PolyDict
e
sage: f = PolyDict(((2, 3): 0))

sage: f
PolyDict with representation {(2, 3): 0}

sage: f.coerce_coefficients(QQ)
...  # DeprecationWarning

sage: f
PolyDict with representation {(2, 3): 0}
```

**degree(x=None)**

Return the total degree or the maximum degree in the variable x.

EXAMPLES:

```python
sage: from sage.rings.polynomial.polydict import PolyDict
e
sage: f = PolyDict({(2, 3): 2, (1, 2): 3, (2, 1): 4})

sage: f.degree()
5

sage: f.degree(PolyDict({(1, 0): 1}))
2

sage: f.degree(PolyDict({(0, 1): 1}))
3
```

**derivative(x)**

Return the derivative of self with respect to x

EXAMPLES:

```python
sage: from sage.rings.polynomial.polydict import PolyDict
e
sage: f = PolyDict({(2, 3): 2, (1, 2): 3, (2, 1): 4})

sage: f.derivative(PolyDict({(1, 0): 1}))
PolyDict with representation {(0, 2): 3, (1, 1): 8, (2, 0): 4, (2, 2): 6}

sage: f.derivative(PolyDict({(0, 1): 1}))
PolyDict with representation {(1, 1): 6, (2, 0): 4, (2, 2): 6}

sage: PolyDict({(-1,): 1}).derivative(PolyDict({(1,): 1}))
PolyDict with representation {(-2,): -1}

sage: PolyDict({(-2,): 1}).derivative(PolyDict({(1,): 1}))
PolyDict with representation {(-3,): -2}

sage: PolyDict({}).derivative(PolyDict({(1, 1): 1}))
Traceback (most recent call last):
...
ValueError: x must be a generator
```

**derivative_i(i)**

Return the derivative of self with respect to the i-th variable.
EXAMPLES:

```python
sage: from sage.rings.polynomial.polydict import PolyDict
sage: PolyDict({(1, 1): 1}).derivative_i(0)
PolyDict with representation {(0, 1): 1}
```

dict()
Return a copy of the dict that defines self.

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:

```python
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)]
```

get(e, default=None)
Return the coefficient of the ETuple e if present and default otherwise.

EXAMPLES:

```python
sage: from sage.rings.polynomial.polydict import PolyDict, ETuple
sage: f = PolyDict({(2, 3): 2, (1, 2): 3, (2, 1): 4})
sage: f.get(ETuple([1,2]))
3
sage: f.get(ETuple([1,1]), 'hello')
'hello'
```

homogenize(var)
Return the homogeneization of self by increasing the degree of the variable var.

EXAMPLES:

```python
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict({(0, 0): 1, (2, 1): 3, (1, 1): 5})
sage: f.homogenize(0)
PolyDict with representation {(2, 1): 8, (3, 0): 1}
sage: f.homogenize(1)
PolyDict with representation {(0, 3): 1, (1, 2): 5, (2, 1): 3}
sage: PolyDict({(0, 1): 1, (1, 1): -1}).homogenize(0)
PolyDict with representation {(1, 1): 0}
```

integral(x)
Return the integral of self with respect to x

EXAMPLES:
```python
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict({(2, 3): 2, (1, 2): 3, (2, 1): 4})
sage: f.integral(PolyDict({(0, 1): 1}))
sage: f.integral(PolyDict({(0, 1): 1}))
PolyDict with representation {(1, 3): 1, (2, 2): 2, (2, 4): 1/2}
sage: PolyDict({(-1,): 1}).integral(PolyDict({(1,): 1}))
Traceback (most recent call last):
  ...  ArithmeticError: integral of monomial with exponent -1
sage: PolyDict({(-2,): 1}).integral(PolyDict({(1,): 1}))
PolyDict with representation {(-1,): -1}
sage: PolyDict({}).integral(PolyDict({(1, 1): 1}))
Traceback (most recent call last):
  ...  ValueError: x must be a generator
```

**integral_i(i)**

Return the derivative of `self` with respect to the `i`-th variable.

**is_constant()**

Return whether this polynomial is constant.

**is_homogeneous()**

Return whether this polynomial is homogeneous.
latex\( (vars, \text{atomic\_exponents}=True, \text{atomic\_coefficients}=True, \text{sortkey}=None) \)

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: \text{f.latex(['a', 'WW'])}
'a^{2} WW^{3} + 4 a^{2} WW + 3 a WW^{2}'
```

\text{lcm}(\text{greater\_etuple})

Provides functionality of lc, lm, and lt by calling the tuple compare function on the provided term order T.

INPUT:

- greater\_etuple – a term order

\text{list}()

Return a list that defines self.

EXAMPLES:

```
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict({(2, 3): 2, (1, 2): 3, (2, 1): 4})
sage: \text{sorted(f.list())}
[[2, [2, 3]], [3, [1, 2]], [4, [2, 1]]]
```

\text{max\_exp}()

Returns an ETuple containing the maximum exponents appearing. If there are no terms at all in the PolyDict, it returns None.

The \text{nvars} parameter is necessary because a PolyDict doesn’t know it from the data it has (and an empty PolyDict offers no clues).

EXAMPLES:

```
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict({(2, 3): 2, (1, 2): 3, (2, 1): 4})
sage: \text{f.max\_exp()}
(2, 3)
sage: PolyDict({}).max\_exp() # returns None
```

\text{min\_exp}()

Returns an ETuple containing the minimum exponents appearing. If there are no terms at all in the PolyDict, it returns None.

The \text{nvars} parameter is necessary because a PolyDict doesn’t know it from the data it has (and an empty PolyDict offers no clues).

EXAMPLES:
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

`monomial_coefficient(mon)`
Return the coefficient of the monomial `mon`.

**INPUT:**
- `mon` – a PolyDict with a single key

**EXAMPLES:**

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())

```
doctest:warning
...
DeprecationWarning: PolyDict.monomial_coefficient is deprecated; use PolyDict.
...get instead
See https://github.com/sagemath/sage/issues/34000 for details.
```
4

`poly_repr(vars, atomic_exponents=True, atomic.coefficients=True, sortkey=None)`
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: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict({(2, 3): RIF(1/2, 3/2), (1, 2): 3, (2, 1): 4})
# needs sage.rings.real_interval_field
sage: f.poly_repr(['x', 'y'])
'1.?*x^2*y^3 + 0.?*x*y^2'

We check to make sure that we are in characteristic two, we don’t put negative signs on the generators.

sage: Integers(2)['x, y'].gens()
(x, y)

We make sure that intervals are correctly represented.

sage: f = PolyDict({(2, 3): RIF(1/2, 3/2), (1, 2): RIF(-1, 1)})  # needs sage.rings.real_interval_field
sage: f.poly_repr(['x', 'y'])
'1.?*x^2*y^3 + 0.?*x*y^2'

3.1. Multivariate Polynomials and Polynomial Rings
polynomial_coefficient(degrees)
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:

```python
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict({(2, 3): 2, (1, 2): 3, (2, 1): 4})
sage: f.polynomial_coefficient([2, 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([0, None])
PolyDict with representation {(0, 2): 3, (0, 3): 2}
```

remove_zeros(zero_test=None)
Remove the entries with zero coefficients.

INPUT:
• zero_test – optional function that performs test to zero of a coefficient

EXAMPLES:

```python
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict({(2, 3): 0})
sage: f
PolyDict with representation {(2, 3): 0}
sage: f.remove_zeros()
sage: f
PolyDict with representation {}
```

The following example shows how to remove only exact zeros from a PolyDict containing univariate power series:

```python
sage: R.<t> = PowerSeriesRing(QQ)
sage: f = PolyDict({(1, 1): O(t), (1, 0): R.zero()})
sage: f.remove_zeros(lambda s: s.is_zero() and s.prec() is Infinity)
sage: f
PolyDict with representation {(1, 1): O(t^1)}
```

rich_compare(other, op, sortkey=None)
Compare two PolyDicts using specified term ordering “sortkey”.

EXAMPLES:

```python
sage: from sage.rings.polynomial.polydict import PolyDict
sage: from sage.structure.richcmp import op_EQ, op_NE, op_LT
sage: p1 = PolyDict({(0,): 1})
sage: p2 = PolyDict({(0,): 2})
sage: O = TermOrder()
sage: p1.rich_compare(PolyDict({(0,): 1}), op_EQ, O.sortkey)
True
```
sage: p1.rich_compare(p2, op_EQ, O.sortkey)
False
sage: p1.rich_compare(p2, op_NE, O.sortkey)
True
sage: p1.rich_compare(p2, op_LT, O.sortkey)
True

sage: p3 = PolyDict({(3, 2, 4): 1, (3, 2, 5): 2})
sage: p4 = PolyDict({(3, 2, 4): 1, (3, 2, 3): 2})
sage: p3.rich_compare(p4, op_LT, O.sortkey)
False

**scalar_lmult(s)**

Return the left scalar multiplication of self by s.

EXAMPLES:

```
sage: from sage.rings.polynomial.polydict import PolyDict
dsage: x, y = FreeMonoid(2, 'x, y').gens() # a strange object to live in a
   → polydict, but non-commutative! # needs sage.combinat
dsage: f = PolyDict({(2,3):x}) #
   → needs sage.combinat
dsage: f.scalar_lmult(y) #
   → needs sage.combinat
PolyDict with representation {(2, 3): y*x}
dsage: f = PolyDict({(2,3):2, (1,2): 3, (2,1): 4})
dsage: f.scalar_lmult(-2)
PolyDict with representation {(1, 2): -6, (2, 1): -8, (2, 3): -4}
dsage: f.scalar_lmult(RIF(-1,1)) #
   → needs sage.rings.real_interval_field
PolyDict with representation {(1, 2): 0.?e1, (2, 1): 0.?e1, (2, 3): 0.?e1}
```

**scalar_rmult(s)**

Return the right scalar multiplication of self by s.

EXAMPLES:

```
sage: from sage.rings.polynomial.polydict import PolyDict
dsage: x, y = FreeMonoid(2, 'x, y').gens() # a strange object to live in a
   → polydict, but non-commutative! # needs sage.combinat
dsage: f = PolyDict({(2,3):x}) #
   → needs sage.combinat
dsage: f.scalar_rmult(y) #
   → needs sage.combinat
PolyDict with representation {(2, 3): x*y}
dsage: f = PolyDict({(2,3):2, (1,2): 3, (2,1): 4})
dsage: f.scalar_rmult(-2)
PolyDict with representation {(1, 2): -6, (2, 1): -8, (2, 3): -4}
dsage: f.scalar_rmult(RIF(-1,1)) #
   → needs sage.rings.real_interval_field
```

(continues on next page)
term_lmult\( (\text{exponent}, s) \)
Return this element multiplied by \( s \) on the left and with exponents shifted by \( \text{exponent} \).

**INPUT:**
- \( \text{exponent} \) – a ETuple
- \( s \) – a scalar

**EXAMPLES:**

```python
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!  # needs sage.combinat
sage: f = PolyDict({(2, 3): x})  # needs sage.combinat
sage: f.term_lmult(ETuple((1, 2)), y)  # needs sage.combinat
PolyDict with representation {(3, 5): x^y}
```

```python
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\( (\text{exponent}, s) \)
Return this element multiplied by \( s \) on the right and with exponents shifted by \( \text{exponent} \).

**INPUT:**
- \( \text{exponent} \) – a ETuple
- \( s \) – a scalar

**EXAMPLES:**

```python
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!  # needs sage.combinat
sage: f = PolyDict({(2, 3): x})  # needs sage.combinat
sage: f.term_rmult(ETuple((1, 2)), y)  # needs sage.combinat
PolyDict with representation {(3, 5): y*x}
```

```python
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\( (w=\text{None}) \)
Return the total degree.

**INPUT:**
• \(w\) – (optional) a tuple of weights

EXAMPLES:

```python
sage: from sage.rings.polynomial.polydict import PolyDict
sage: f = PolyDict({(2, 3): 2, (1, 2): 3, (2, 1): 4})
sage: f.total_degree()
5
sage: f.total_degree((3, 1))
9
sage: PolyDict({}).degree()
-1
```

```
sage.rings.polynomial.polydict.gen_index(x)
Return the index of the variable represented by \(x\) or \(-1\) if \(x\) is not a monomial of degree one.

EXAMPLES:

```python
sage: from sage.rings.polynomial.polydict import PolyDict, gen_index
sage: gen_index(PolyDict({(1, 0): 1}))
0
sage: gen_index(PolyDict({(0, 1): 1}))
1
sage: gen_index(PolyDict({}))
-1
```

```
sage.rings.polynomial.polydict.make_ETuple(data, length)
Ensure support for pickled data from older sage versions.

sage.rings.polynomial.polydict.make_PolyDict(data)
Ensure support for pickled data from older sage versions.

sage.rings.polynomial.polydict.monomial_exponent(p)
Return the unique exponent of \(p\) if it is a monomial or raise a ValueError.

EXAMPLES:

```python
sage: from sage.rings.polynomial.polydict import PolyDict, monomial_exponent
sage: monomial_exponent(PolyDict({(2, 3): 1}))
(2, 3)
sage: monomial_exponent(PolyDict({(2, 3): 3}))
Traceback (most recent call last):
  ... ValueError: not a monomial
sage: monomial_exponent(PolyDict({(1, 0): 1, (0, 1): 1}))
Traceback (most recent call last):
  ... ValueError: not a monomial
```
3.1.12 Compute Hilbert series of monomial ideals

This implementation was provided at github issue #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.

**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.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: from sage.rings.polynomial.hilbert import hilbert_poincare_series
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
```
The following example is taken from github issue #20145:

```python
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
```

### 3.1.13 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:**

```python
sage: R = QQ['x'][y]['s', 't'][X]
sage: from sage.rings.polynomial.flatten import FlatteningMorphism
sage: phi = FlatteningMorphism(R); phi
Flattening morphism:
    From: Univariate Polynomial Ring in X
          over Multivariate Polynomial Ring in s, t
          over Univariate Polynomial Ring in y
          over Univariate Polynomial Ring in x over Rational Field
    To:  Multivariate Polynomial Ring in x, y, s, t, X over Rational Field
sage: phi('x*y*s + t*X').parent()
Multivariate Polynomial Ring in x, y, s, t, X over Rational Field
```

Authors:

Vincent Delecroix, Ben Hutz (July 2016): initial implementation

**class** `sage.rings.polynomial.flatten.FlatteningMorphism(domain)`

**Bases:** `Morphism`

**EXAMPLES:**

```python
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 a, b, 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
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:
Polynomials, Release 10.1

```python
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
```

```python
inverse()  
Return the inverse of this flattening morphism.
This is the same as calling `section()`.
EXAMPLES:

```python
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 Ring in x, y over Rational Field
```

```python
section()  
Inverse of this flattening morphism.
EXAMPLES:

```python
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 Ring in a, b, c over Rational Field
```

```python
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
```

```python
class sage.rings.polynomial.flatten.FractionSpecializationMorphism(domain, D)
Bases: Morphism
A specialization morphism for fraction fields over (stacked) polynomial rings
```

490 Chapter 3. Multivariate Polynomials
class sage.rings.polynomial.flatten.SpecializationMorphism(domain, D)

Bases: 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 Univariate Polynomial Ring in u over Rational Field
  To:   Univariate Polynomial Ring in y, z over Univariate Polynomial Ring in c
         over Univariate Polynomial Ring in v 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: 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^2*y^2 - z^2')); g
```

(continues on next page)
Polynomials, Release 10.1

\[ x^2 + c^*y^2 - z^2 \]

**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

### 3.1.14 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])
\([1, y, y^2, y^3, y^4, x, x^y, x^y^2, x^y^3, x^y^4, x^2, x^2^y, x^2^y^2, x^2^y^3, x^2^y^4, x^3^y^4, x^3^y^3, x^3^y^2, x^3^y^1, x^3^y^0, x^3^y^1, x^3^y^2, x^3^y^3, x^3^y^4, x^4^y^4, x^4^y^3, x^4^y^2, x^4^y^1, x^4^y^0, x^4^y^1, x^4^y^2, x^4^y^3, x^4^y^4, x^5^y^4, x^5^y^3, x^5^y^2, x^5^y^1, x^5^y^0] \)
**sage:** monomials([x,y,z], [2,3,2])
\([1, z, y, y^z, y^2, y^2^z, x, x^z, x^y, x^y^2, x^y^2^z, x^y^2^2, x^y^2^3] \)

### 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 \( SL(n, \mathbb{C}) \) acts on the variables \((x_1, \ldots, x_n)\) linearly,

\[ (x_1, \ldots, x_n)^t \rightarrow A(x_1, \ldots, x_n)^t, \quad A \in SL(n, \mathbb{C}) \]

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...
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 \texttt{invariant\_theory}. For example, take the quartic:

```python
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):

```python
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:

```python
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:

```python
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 \texttt{Glossary\_of\_invariant\_theory}

3.2. Classical Invariant Theory 493
class sage.rings.invariants.invariant_theory.AlgebraicForm(n, d, polynomial, *args, **kwds)

Bases: 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:

```
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)

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)
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
```
```python
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:**

```python
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:**

```python
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
```

**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:**

```python
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
```
polynomial()

Return the defining polynomial.

OUTPUT:

The polynomial used to define the algebraic form.

EXAMPLES:

```sage
 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 $GL(n, C)$ 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
 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()
 3*x^3 + y^3 + 4*x*y*z + 2*z^3
 sage: cyc = matrix([[0,1,0], [0,0,1], [1,0,0]])
 sage: cubic.transformed(cyc) == cubic.transformed({x:y, y:z, z:x})
 True
 sage: g = matrix(QQ, [[1, 0, 0], [-1, 1, -3], [-5, -5, 16]])
 sage: cubic.transformed(g)
```

class sage.rings.invariants.invariant_theory.BinaryQuartic(n, d, polynomial, *args)

Bases: 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.

\[ f(x) = a_0 x_1^4 + 4a_1 x_0 x_1^3 + 6a_2 x_0^2 x_1^2 + 4a_3 x_0^3 x_1 + a_4 x_0^4 \]
\[ \Rightarrow D(f) = a_0 a_4 + 3a_2^2 - 4a_1 a_3 \]

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.

\[ f(x) = a_0 x_1^4 + 4a_1 x_0 x_1^3 + 6a_2 x_0^2 x_1^2 + 4a_3 x_0^3 x_1 + a_4 x_0^4 \]
\[ \Rightarrow E(f) = a_0 a_3^2 + a_2 a_4 - a_0 a_2 a_3 + 2a_1 a_2 a_3 + a_2^3 \]

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.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: 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)

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)

\textbf{g\_covariant()}

The \(g\)-covariant of a binary quartic.

**OUTPUT:**
The \(g\)-covariant of the quartic.

\[
\begin{align*}
f(x) &= a_0 x_1^4 + 4a_1 x_0 x_1^3 + 6a_2 x_0^2 x_1^2 + 4a_3 x_0^3 x_1 + a_4 x_0^4 \\
\Rightarrow D(f) &= \frac{1}{144} \left( \frac{\partial^2 f}{\partial x \partial x} \right)
\end{align*}
\]

**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: 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

sage: inv_inhomogeneous = invariant_theory.binary_quartic(p.subs(y=1), x)
sage: inv_inhomogeneous.g_covariant()
- 2*a1*a3*x^2 - a0*a4*x^2 + 2*a2*a3*x - 2*a1*a4*x + a3^2 - a2*a4

sage: g == 1/144 * (p.derivative(x,y)^2 - p.derivative(x,x)*p.derivative(y,y))
True

\textbf{h\_covariant()}

The \(h\)-covariant of a binary quartic.

**OUTPUT:**
The \(h\)-covariant of the quartic.

\[
\begin{align*}
f(x) &= a_0 x_1^4 + 4a_1 x_0 x_1^3 + 6a_2 x_0^2 x_1^2 + 4a_3 x_0^3 x_1 + a_4 x_0^4 \\
\Rightarrow D(f) &= \frac{1}{144} \left( \frac{\partial^2 f}{\partial x \partial x} \right)
\end{align*}
\]

**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: 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

sage: inv_inhomogeneous = invariant_theory.binary_quartic(p.subs(y=1), x)
sage: inv_inhomogeneous.g_covariant()
- 2*a1*a3*x^2 - a0*a4*x^2 + 2*a2*a3*x - 2*a1*a4*x + a3^2 - a2*a4

sage: g == 1/144 * (p.derivative(x,y)^2 - p.derivative(x,x)*p.derivative(y,y))
True
```python
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*a1*a2*a4*x^2*y^4 + 5*a0*a3*a4*x^2*y^4
+ 6*a2*a3^2*x*y^5 - 9*a2*a4*x*y^5 + 2*a1*a3*a4*x*y^5 + a0*a4^2*x*y^5
+ 2*a3^3*y^6 - 3*a2*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*a4*x^2 + 5*a0*a3*a4*x^2 + 6*a2*a3^2*x - 9*a2*a4*x
+ 2*a1*a3*a4*x + a0*a4^2*x + 2*a3^3 - 3*a2*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:**

```python
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

\[ f(x) = a_0 x_1^4 + 4 a_1 x_0 x_1^3 + 6 a_2 x_0^2 x_1^2 + 4 a_3 x_0^3 x_1 + a_4 x_0^4 \]

this function returns \( a = (a_0, a_1, a_2, a_3, a_4) \)

**EXAMPLES:**

```python
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 + 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)
class sage.rings.invariants.invariant_theory.BinaryQuintic(n, d, polynomial, *args)

Invariant theory of a binary quintic form.

You should use the invariant_theory factory object to construct instances of this class. See binary_quintic() for details.

REFERENCES:
For a description of all invariants and covariants of a binary quintic, see section 73 of [Cle1872].

A_invariant()

Return the invariant $A$ of a binary quintic.

OUTPUT:
The $A$-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.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*a1*a4*a5 - 2*a0^2*a5^2
```

B_invariant()

Return the invariant $B$ of a binary quintic.

OUTPUT:
The $B$-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.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 - 16/125*a0*a2*a3^4*a4 + 4/125*a0*a1*a2^2*a4^2*a5^2 - 4/125*a0^2*a2^2*a3*a5^3 + 2/25*a0*a1^2*a3*a5^3
... + 8/625*a0^2*a1^2*a4^2*a5^2 - 4/125*a0^3*a2^2*a4^2*a5^2 - 2/125*a0^2*a2^2*a3*a5^3 - 4/125*a0^2*a1^2*a3*a5^3 + 2/25*a0^3*a2^2*a3*a5^3
```
\textbf{C\_invariant()} 
Return the invariant $C$ of a binary quintic.

\textbf{OUTPUT:} 
The $C$-invariant of the binary quintic.

\textbf{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.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*a3^4 - 36/3125*a0^3*a1^2*a2^3*a3^4
+ 24/15625*a0^2*a1^4*a3^2*a5^4 - 24/3125*a0^3*a1^2*a2*a3^2*a5^4
+ 6/3125*a0^4*a2*a3^2*a5^4
\end{verbatim}

\textbf{H\_covariant}(\textit{as\_form=False})
Return the covariant $H$ of a binary quintic.

\textbf{INPUT:}
\begin{itemize}
\item \textit{as\_form} – if \textit{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 \texttt{AlgebraicForm}.
\end{itemize}

\textbf{OUTPUT:} 
The $H$-covariant of the binary quintic as polynomial or as binary form.

\textbf{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.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*a2^2*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*x0^1*x1^5 + 3/5*a0*a3*x0^1*x1^5
- 2/25*a1^2*x1^6 + 1/5*a0*a2*x1^6
sage: quintic.H_covariant(as\_form=True)
Binary sextic given by ...
\end{verbatim}

\textbf{R\_invariant()} 
Return the invariant $R$ of a binary quintic.

\textbf{OUTPUT:} 
The $R$-invariant of the binary quintic.

\textbf{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()
3/3906250000000000*a1^2*a2^5*a3^11 - 3/976562500000*a0*a2^6*a3^11
- 51/781250000000000*a1^3*a2^3*a3^12 + 27/976562500000000*a0*a1*a2^4*a3^12
+ 27/1953125000000000*a1^4*a2*a3^13 - 81/15625000000000*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

\textbf{T\_covariant}(\texttt{as\_form=False})

Return the covariant $T$ of a binary quintic.

**INPUT:**

- \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}.

**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*x0^8 + 1/10*a2*a5^2*x0^9
+ 9/250*a3^2*a4^2*x0^8*x1 - 3/25*a3^3*a5*x0^8*x1 + 1/50*a2*a4^3*a5*x0^8*x1
+ 2/5*a1*a5^2*x0^8*x1 + 3/250*a3^2*a4^2*x0^7*x1^2 + 8/125*a2*a4^2*a5*x0^7*x1^2
...
11/25*a0*a1*a4*x0^2*x1^7 - a0^2*a5*x0^2*x1^7 - 9/250*a1^2*a2*a0*x0^2*x1^8
+ 3/25*a0^2*a2^2*x0^2*x1^8 - 1/50*a0*a1^3*a0*x0^2*x1^8 - 2/5*a0^2*a4*x0^2*x1^8
- 2/125*a1^3*x1^9 + 3/50*a0*a1*a2*x1^9 - 1/10*a0^2*a3*x1^9

sage: quintic.T_covariant(as\_form=\texttt{True})

Binary nonic given by ...

\textbf{alpha\_covariant}(\texttt{as\_form=False})

Return the covariant $\alpha$ of a binary quintic.

**INPUT:**

- \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}.

**OUTPUT:**

The $\alpha$-covariant of the binary quintic as polynomial or as binary form.

**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.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*a3^2*a5^2*x1 + 24/625*a1^3*a4*a5*x1 - 8/125*a0*a1*a4*a5^2*x1 - 4/25*a0^2*a3*a4*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 invariant 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 \(\text{BinaryQuintic.clebsch_invariants()}\).

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}
\end{verbatim}

3.2. Classical Invariant Theory {503}
**beta_covariant**(as_form=False)
Return the covariant $\beta$ 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 $\beta$-covariant of the binary quintic as polynomial or as binary form.

**EXAMPLES:**

```plaintext
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.beta_covariant()  # default, as_form=False
-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 - 31/62500*a1*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*a1^3*a4*a5^2*x1 + 6/125*a0*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*a0^2*a2*a5^3*x1

sage: quintic.beta_covariant(as_form=True)  # 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:**

```plaintext
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: f = invariant_theory.binary_quintic(p, x0, x1)
sage: gf = f.transformed(matrix(QQ, [[11,5],[7,2]]))
sage: f.canonical_form() == gf.canonical_form()  # default, reduce_gcd=False
True
sage: h = f.canonical_form(reduce_gcd=True)
sage: 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:**
```
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

\[ f(x) = a_0 x_1^5 + a_1 x_0 x_1^4 + a_2 x_0^2 x_1^3 + a_3 x_0^3 x_1^2 + a_4 x_0^4 x_1 + a_5 x_0^5 \]

this function returns \( a = (a_0, a_1, a_2, a_3, a_4, a_5) \)

**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.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`.

---

3.2. Classical Invariant Theory 505
OUTPUT:

The $\delta$-covariant of the binary quintic as polynomial or as binary form.

EXAMPLES:

```sage
er.a0, a1, a2, a3, a4, a5, x0, x1> = QQ[]
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
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*a2^2*a3^9*x0 + 9/781250000*a0*a2^3*a3^9*x0
- 9/1562500000*a1^3*a3^10*x0 - 81/1562500000*a0*a1*a2*a3^10*x0
... + 64/3125*a0^3*a1^3*a2^2*a5^5*x1 - 12/625*a0^4*a1*a2^3*a5^5*x1
+ 16/3125*a0^3*a1^4*a3*a5^5*x1 - 16/625*a0^4*a1^2*a2*a3*a5^5*x1
+ 4/125*a0^5*a2^2*a3*a5^5*x1
```

```sage
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:

```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:

```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
```

(continues on next page)
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.gamma_covariant()
1/156250000*a2^5*a3^6*x0 - 3/62500000*a1*a2^3*a3^7*x0
+ 27/312500000*a1^2*a2^3*a3^8*x0 + 27/312500000*a0*a2^2*a3^8*x0
- 81/312500000*a0*a1*a3^9*x0 - 19/312500000*a2^6*a3^4*a4^4*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*a1^4*a3^4*a5^4*x1 + 8/625*a0^3*a1^2*a2*a3*a5^4*x1
- 2/125*a0^4*a2^2*a3*a5^4*x1
sage: quintic.gamma_covariant(as_form=True)
Binary monic given by ...

\textbf{i\_covariant\( (as\_form=False)\)}

Return the covariant \(i\) of a binary quintic.

\textbf{INPUT:}

\begin{itemize}
  \item \texttt{as}\_\texttt{form} – if \texttt{as}\_\texttt{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 \(i\)-covariant of the binary quintic as polynomial or as binary form.

\textbf{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 +
   \rightarrow a5*x0^5
sage: quintic = invariant_theory.binary_quintic(p, x0, x1)
sage: quintic.i_covariant()
3/50*a3^2*x0^2 - 4/25*a2*a4*x0^2 + 2/5*a1*a5*x0^2 + 3/50*a2*a3*x0^3 + a4*x1*x0^4 +
   \rightarrow a5*x0^5
sage: quintic.i_covariant(as_form=True)
Binary quadratic given by ...
\end{verbatim}

\textbf{invariants\( (type='clebsch')\)}

Return a tuple of invariants of a binary quintic.

\textbf{INPUT:}

\begin{itemize}
  \item \texttt{type} – The type of invariants to return. The default choice is to return the Clebsch invariants.
\end{itemize}

\textbf{OUTPUT:}

The invariants of the binary quintic.

\textbf{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.invariants()
(-276032/625,
(continues on next page)
... (continued from previous page)

\[ \frac{4983526016}{390625}, -\frac{247056495846408}{244140625}, -\frac{148978972828696847376}{30517578125} \]

sage: quintic.invariants('unknown')
Traceback (most recent call last):
  ...
ValueError: unknown type of invariants unknown for a binary quintic

**j_covariant**(as_form=False)

Return the covariant \( j \) 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 \( j \)-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.j_covariant()
-\frac{3}{500}a3^3x0^3 + \frac{3}{125}a2a3a4x0^3 - \frac{6}{125}a1a4^2x0^3
- \frac{3}{25}a2^2a5x0^3 + \frac{3}{25}a1a3a5x0^3 - \frac{3}{25}a2a3^2x0^2x1
+ \frac{3}{25}a2^2a4x0^2x1 + \frac{3}{125}a1a3a4x0^2x1 - \frac{6}{25}a0a4^2x0^2x1
- \frac{3}{25}a1a2a5x0^2x1 + \frac{3}{5}a0a3a5x0^2x1 - \frac{3}{5}a2a3^2x0^2x1
+ \frac{3}{25}a2^2a4x0^2x1 + \frac{3}{125}a1a2a4x0^2x1 - \frac{3}{25}a0a3a4x0^2x1
+ \frac{6}{25}a1^2a5x0^2x1 + \frac{3}{5}a0a2a5x0^2x1 - \frac{3}{5}a2a3^2x0^2x1
+ \frac{3}{125}a1a2a3x0^2x1 + \frac{3}{5}a0a3^2x0^2x1 - \frac{6}{125}a1a2a4x0^2x1
+ \frac{3}{25}a0a2a4^2x0^2x1
```

```python
sage: quintic.j_covariant(as_form=True)
Binary cubic given by ...
```

**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.

**OUTPUT:**

A tuple of monomials. They are in the same order as *coeffs()*.

**EXAMPLES:**

```python
sage: R.<x,y> = QQ[]
sage: quintic = invariant_theory.binary_quintic(x^5 + y^5)
sage: quintic.monomials()
(y^5, x*y^4, x^2*y^3, x^3*y^2, x^4*y, x^5)
```

**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 \( a = (a_0, a_1, a_2, a_3, a_4, a_5) \)

**EXAMPLES:**

```python
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.tau_covariant()
1/62500*a2^2*a3^4*x0^2 - 3/62500*a1*a3^5*x0^2 - 1/15625*a2^3*a3^2*a4*x0^2 + 1/6250*a1*a2*a3^3*a4*x0^2 + 3/6250*a0*a3^4*a4*x0^2 - 1/31250*a2^4*a4^2*x0^2 - 2/125*a0*a1*a2^2*a4^2*x1^2 - 4/125*a0*a1^2*a4^2*a5*x1^2 + 2/25*a0^2*a2^2*a4^2*a5*x1^2 - 8/625*a0^4*a5^2*x1^2 + 8/125*a0*a1*a2^2*a5^2*x1^2 - 2/25*a0^2*a2^2*a5^2*x1^2

sage: quintic.tau_covariant(as_form=True)
Binary quadratic given by ...
```

**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:**

```python
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.tau_covariant()
(a0, a1, a2, a3, a4, a5)

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*.

**EXAMPLES:**

```python
sage: quintic.theta_covariant(as_form=False)
Binary quadratic given by ...
```
The $\theta$-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.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*a0^3*a2*a3*a4*a5^2*x1^2 + 8/625*a0*a1^4*a5^3*x1^2
- 8/125*a0^2*a1^2*a2*a5^3*x1^2 + 2/25*a0^3*a2^2*a5^3*x1^2
sage: quintic.theta_covariant(as_form=True)
Binary quadratic given by ...
```

### class `sage.rings.invariants.invariant_theory.FormsBase(n, homogeneous, ring, variables)`

Bases: `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:**

```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.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:

```
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:

```
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.variables()
(x, y)

sage: R.<x,y,t> = QQ[]
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:

```
sage: R.<x,y,z> = QQ[]
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)

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.
```

3.2. Classical Invariant Theory 511
• 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: invariant_theory.binary_form_from_invariants(3, [discriminant])
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]
sage: invariant_theory.binary_form_from_invariants(5, invariants)
Binary quintic with coefficients (1, 0, 0, 0, 1, 0)
```

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]
sage: invariant_theory.binary_form_from_invariants(5, invariants,
      scaling='normalized')
Binary quintic with coefficients (-37725479487783/1048576,
  565882192316745/8388608, 0, 1033866765362693115/67108864,
  12849480940936328715/268435456, -23129076493685391687/2147483648)
sage: invariant_theory.binary_form_from_invariants(5, invariants,
      scaling='coprime')
Binary quintic with coefficients (24389/892616806656,
  4205/11019960576, 0, 1015/209952, -145/1296, -3/16)
```

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 $GL(2, \mathbb{Q})$-equivalent to the provided form and hence has the same
canonical form (see \texttt{canonical\_form()}):

\begin{verbatim}
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
(9592267437341790539005557/2441406250000000000,
2149296928207625556323004064707/6103515625000000000,
1114965189034770974453304786783/7629394531250000000,
12265077575189463895648891202734239/47683715820312500000,
32399663094570652847286334593218447/119209289550781250000000,
15945065036468395841632538558481466127/1490116119384765625000000)
sage: quintic.canonical_form() == newquintic.canonical_form()
True
\end{verbatim}

For binary forms of other degrees, no reconstruction has been implemented yet. For forms of degree 6, see \texttt{github issue #26462}:

\begin{verbatim}
sage: invariant_theory.binary_form_from_invariants(6, invariants)
Traceback (most recent call last):
  ...:
NotImplementedError: no reconstruction for binary forms of degree 6 implemented
\end{verbatim}

\textbf{binary\_quadratic}(\textit{quadratic}, \texttt{*args})

Invariant theory of a quadratic in two variables.

\textbf{INPUT:}

\begin{itemize}
  \item \textit{quadratic} – a quadratic form.
  \item \textit{x, y} – the homogeneous variables. If \textit{y} is \texttt{None}, the quadratic is assumed to be inhomogeneous.
\end{itemize}

\textbf{REFERENCES:}

\begin{itemize}
  \item Wikipedia article \texttt{Invariant\_of\_a\_binary\_form}
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
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)
\end{verbatim}

\textbf{binary\_quartic}(\textit{quartic}, \texttt{*args}, \texttt{**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 \texttt{EisensteinD()}) and \(E_6\) (see \texttt{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 \texttt{g\_covariant()}) of degree 2 and order 4, and a covariant \(h\) (see \texttt{h\_covariant()}) of degree 3 and order
6. They are related by a syzygy

\[ jf^3 - g^2f^2i + 4g^3 + h^2 = 0 \]

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:

- Wikipedia article Invariant_of_a_binary_form

EXAMPLES:

```
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 \( \text{A}_{\text{invariant}}() \), \( \text{B}_{\text{invariant}}() \) and \( \text{C}_{\text{invariant}}() \)).

INPUT:

- \( \text{quintic} \) – a homogeneous polynomial of degree five in two variables or a (possibly inhomogeneous) polynomial of degree at most five in one variable.
- \(*\text{args}\) – 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:

- Wikipedia article Invariant_of_a_binary_form
- [Cle1872]

EXAMPLES:

If no variables are provided, they will be guessed:

```
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)
sage: quintic.is_homogeneous()
False
```

If only one variable is given, the quintic is the homogenisation of the provided polynomial:

```
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:

```python
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'>
```

### 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:**

```python
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'>
```

### 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:**

```python
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'>
```

If some of the ring variables are to be treated as coefficients you need to specify the polynomial variables:

```python
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)
```

(continues on next page)
In homogeneous quadratic forms (see also \textit{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)
\end{verbatim}

\textbf{quaternary\_biquadratic}(quadratic1, quadratic2, *\texttt{args}, **\texttt{kwds})

Invariants of two quadratics in four variables.

INPUT:

\begin{itemize}
  \item quadratic1, quadratic2 – two polynomials. Either homogeneous quadratic in 4 homogeneous variables, or inhomogeneous quadratic in 3 variables.
  \item w, x, y, z – the variables. If z is \texttt{None}, the quadratics are assumed to be inhomogeneous.
\end{itemize}

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: 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}(quadratic, *\texttt{args})

Invariant theory of a quadratic in four variables.

INPUT:

\begin{itemize}
  \item quadratic – a quadratic form.
\end{itemize}
• \(w, x, y, z\) – the homogeneous variables. If \(z\) is None, the quadratic is assumed to be inhomogeneous.

REFERENCES:

• Wikipedia article Invariant_of_a_binary_form

EXAMPLES:

```python
sage: R.<w,x,y,z> = QQ[]
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[]
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_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[]
sage: q1 = x^2 + y^2 + z^2
sage: q2 = x*y + y*z + x*z
sage: inv = invariant_theory.ternary_biquadratic(q1, q2)
sage: type(inv)
<class 'sage.rings.invariants.invariant_theory.TwoTernaryQuadratics'>
```

Distance between two circles:

```python
sage: R.<x,y, a,b, r1,r2> = QQ[]
sage: S1 = -r1^2 + x^2 + y^2
sage: S2 = -r2^2 + (x-a)^2 + (y-b)^2
sage: inv = invariant_theory.ternary_biquadratic(S1, S2, [x, y])
sage: inv.Delta_invariant()
-r1^2
sage: inv.Delta_prime_invariant()
-r2^2
sage: inv.Theta_invariant()
a^2 + b^2 - 2*r1^2 - r2^2
sage: inv.Theta_prime_invariant()
a^2 + b^2 - r1^2 - 2*r2^2
sage: inv.F_covariant()
2*x^2*a^2 + y^2*a^2 + 2*x*y*a^2 + 2*x*a^2*b - 2*y*a^2*b + x^2*b^2 + 2*y^2*b^2 - 2*x*a*b^2 + 2*y*b^2 - 2*x^2*r1^2 - 2*y^2*r1^2 + 2*x*a*r1^2 + 2*y*b*r1^2 + r1^4 - 2*x^2*r2^2 - 2*y^2*r2^2 + 2*x*a*r2^2 + 2*y*b*r2^2 + 2*r1^2*r2^2 + r2^4
sage: inv.I_covariant()
-8*x^2*y*a^3 + 8*x^2*a^3*b - 8*x^2*a^2*b - 16*x*y^2*a^2*b + 8*x^2*a^3*b +
```

(continues on next page)
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

\[
J^2 = 4\Theta^3 + TU^2\Theta^2 + \Theta(-4S^3U^4 + 2STU^3H - 72S^2U^2H^2 - 18TUH^3 + 108SH^4) - 16S^3U^5H - 11S^2TU^4H^2 - 4T^2U^3H^3 + 54STU^2H^4 - 432S^2UH^5 - 27TH^6
\]

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: invariant_theory.ternary_quadratic(x^2 + y^2 + z^2)
Ternary quadratic with coefficients (1, 1, 1, 0, 0, 0)

sage: T.<u,v> = QQ[]
sage: invariant_theory.ternary_quadratic(1 + u^2 + v^2)
Ternary quadratic with coefficients (1, 1, 1, 0, 0, 0)

sage: quadratic = x^2 + y^2 + z^2
sage: inv = invariant_theory.ternary_quadratic(quadratic)
sage: type(inv)
<class 'sage.rings.invariants.invariant_theory.TernaryQuadratic'>

class sage.rings.invariants.invariant_theory.QuadraticForm(n, d, polynomial, *args)
Bases: 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 rep-
resentation.

EXAMPLES:

sage: R.<x,y,z> = QQ[]
sage: p = x^2 + y^2 + z^2 + 2*x*y + 3*x*z
sage: quadratic = invariant_theory.ternary_quadratic(p)
sage: matrix(quadratic)
[ 1 1 3/2]
[ 1 1 0]
[3/2 0 1]
sage: quadratic.as_QuadraticForm()
Quadratic form in 3 variables over Multivariate Polynomial
Ring in x, y, z over Rational Field with coefficients:
[ 1 2 3 ]
[ * 1 0 ]
[ * * 1 ]
sage: _.polynomial('X,Y,Z')
X^2 + 2*X*Y + Y^2 + 3*X*Z + Z^2

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:
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: inv = invariant_theory.quadratic_form(p, x,y,z); inv
Ternary quadratic with coefficients (a, b, c, d, e, f)
sage: inv.coefs()
(a, b, c, d, e, f)
sage: inv.scaled_coefs()
(a, b, c, 1/2*d, 1/2*e, 1/2*f)

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:

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:

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])
Ternary quadratic with coefficients (1, 1, -1/4*t^2 + 1, -t, 0, 0)
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])
classmethod from_invariants(discriminant, x, z, *args, **kwargs)

Construct a binary quadratic from its discriminant.

This function constructs a binary quadratic whose discriminant equal the one provided as argument up to scaling.

**INPUT:**

- discriminant – Value of the discriminant used to reconstruct the binary quadratic.

**OUTPUT:**

A QuadraticForm with 2 variables.

**EXAMPLES:**

```python
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:**

```python
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

\[
Q(x, y, z, \ldots) = (x, y, \ldots)A(x, y, \ldots)^\top
\]

**EXAMPLES:**

3.2. Classical Invariant Theory
```python
sage: R.<x,y,z> = QQ[]
sage: quadratic = invariant_theory.ternary_quadratic(x^2+y^2+z^2+x*y)
sage: matrix(quadratic)
\[
\begin{bmatrix}
  1 & 1/2 & 0 \\
  1/2 & 1 & 0 \\
  0 & 0 & 1
\end{bmatrix}
\]
sage: quadratic._matrix_() == matrix(quadratic)
True
```

```python
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)
sage: quadratic = invariant_theory.inhomogeneous_quadratic_form(x^2 + y^2)
sage: quadratic.monomials()
(x^2, y^2, 1, x*y, x, y)
```

```python
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[]
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); 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:** 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
sage: R.<x,y> = QQ[]
sage: p = AlgebraicForm(2, 2, x^2, (x,y))
sage: q = AlgebraicForm(2, 2, y^2, (x,y))
```

```python
sage: pq = SeveralAlgebraicForms([p, q])
```

**get_form(i)**

Return the \(i\)-th form.

**EXAMPLES:**

```python
sage: R.<x,y> = QQ[]
sage: q1 = invariant_theory.quadratic_form(x^2 + y^2)
```

```python
sage: q12 = SeveralAlgebraicForms([q1, q2])
```

```python
sage: q12.get_form(0) is q1
True
```

```python
sage: q12.get_form(1) is q2
True
```

```python
sage: q12[0] is q12.get_form(0)  # syntactic sugar
True
```

```python
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:**

```python
sage: R.<x,y,z> = QQ[]
```

```python
sage: q = invariant_theory.quaternary_biquadratic(x^2 + 1, y^2 + 1, [x,y,z])
```

```python
sage: q
```

```python
Joint quaternary quadratic with coefficients \((1, 0, 0, 1, 0, 0, 0, 0, 0, 0)\) and quaternary quadratic with coefficients \((0, 1, 0, 1, 0, 0, 0, 0, 0, 0)\)
```

```python
sage: q.homogenized()
```

```python
Joint quaternary quadratic with coefficients \((1, 0, 0, 1, 0, 0, 0, 0, 0, 0)\) and quaternary quadratic with coefficients \((0, 1, 0, 1, 0, 0, 0, 0, 0, 0)\)
```

```python
sage: type(q) is type(q.homogenized())
```

True

**n_forms()**

Return the number of forms.
EXAMPLES:

```python
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
```

```python
class sage.rings.invariants.invariant_theory.TernaryCubic(n, d, polynomial, *args)
Bases: 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:

sage: R.<x,y,z> = QQ[]
sage: cubic = invariant_theory.ternary_cubic(x^3 + y^3 + z^3)
sage: cubic.T_invariant()
1
sage: R.<x,y,z,t> = GF(7)[]
sage: cubic = invariant_theory.ternary_cubic(x^3 + y^3 + z^3 + t*x*y*z, [x,y,z])
sage: cubic.T_invariant()
-t^6 - t^3 + 1

Theta_covariant()
Return the $\Theta$ covariant.

EXAMPLES:

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
sage: R.<x,y> = QQ[]
sage: cubic = invariant_theory.ternary_cubic(x^3 + y^3 + 1)
sage: cubic.Theta_covariant()
-x^3*y^3 - x^3 - y^3
sage: R.<x,y,z,a30,a21,a12,a03,a20,a11,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 + a01*x*z^2 + a00*z^3 )
sage: cubic = invariant_theory.ternary_cubic(p, x,y,z)
sage: len(list(cubic.Theta_covariant()))
6952

coeffs()
Return the coefficients of a cubic.

Given

\[ p(x, y) = a_{30}x^3 + a_{21}x^2y + a_{12}xy^2 + a_{03}y^3 + a_{20}x^2 + 
   a_{11}xy + a_{02}y^2 + a_{10}x + a_{01}y + a_{00} \]

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:
Polynomials, Release 10.1

```python
sage: R.<x,y,z,a30,a21,a12,a03,a20,a11,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).coeffs()
(a30, a03, a00, a21, a20, a12, a02, a10, a01, a11)
sage: invariant_theory.ternary_cubic(p.subs(z=1), 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:**

```python
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:**

```python
sage: R.<x,y,z,X,Y,Z,a30,a21,a12,a03,a20,a11,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()
[ 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]
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 `coeffs()`, this method returns rescaled coefficients that are often used in invariant theory.

Given

$$p(x, y) = a_{30}x^3 + a_{21}x^2y + a_{12}xy^2 + a_{03}y^3 + a_{20}x^2 +
.....: a_{11}xy + a_{02}y^2 + a_{10}x + a_{01}y + a_{00}$$

this function returns $a = (a_{30}, a_{03}, a_{00}, a_{21}/3, a_{20}/3, a_{12}/3, a_{02}/3, a_{10}/3, a_{01}/3, a_{11}/6)$

526 Chapter 3. Multivariate Polynomials
EXAMPLES:

```
sage: R.<x,y,z,a30,a21,a12,a03,a20,a11,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*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.

**OUTPUT:**

0 if evaluated for the form, the *S* invariant, the *T* invariant, the Hessian, the Θ covariant and the *J*-covariant of a ternary cubic.

**EXAMPLES:**

```
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)
0
```

### class sage.rings.invariants.invariant_theory.TernaryQuadratic(*n, d, polynomial, *args)

**Bases:** 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:**

```
sage: R.<x,y,z,a20,a02,a11,a10,a01,a00> = QQ[]
sage: p = ( a20*x^2 + a11*x*y + a02*y^2 + ...:
       a10*x*z + a01*y*z + a00*z^2 )
```

(continues on next page)
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^2*y - x^2*z - y^2*z
sage: R.covariant_conic(Q)
-x^2*y - x^2*z - y^2*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^2)
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 \( a = (a_{20}, a_{02}, a_{00}, a_{11}/2, a_{10}/2, a_{01}/2) \).

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 + a01*y + a00 )
.....: a10^*x^2 + a01^*y^2 + 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: 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])
    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: 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 + 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: 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:

```
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 + 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: 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:

```
sage: R.<w,x,y,z,a0,a1,a2,a3,A0,A1,A2,A3> = QQ[]
sage: p1 = a0*w^2 + a1*x^2 + a2*z^2 + a3
sage: p2 = A0*w^2 + A1*x^2 + A2*z^2 + A3
sage: q = invariant_theory.quaternary_biquadratic(p1, p2, [w, x, y, z])
sage: q.J_covariant().factor()
z * w * y * x * (a3*A2 - a2*A3) * (a3*A1 - a1*A3) * (-a2*A1 + a1*A2) * (a3*A0 - a0*A3) * (-a2*A0 + a1*A2) * (-a1*A0 + a0*A1)
```

Phi_invariant()
Return the $\Phi'$ 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.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(lambda m: m.coefficient(t))

sage: M == q.Delta_invariant() * T
True

**T\_prime\_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: Tprime = invariant_theory.quaternary_quadratic(...: q.T_prime_covariant(), [x,y,z]).matrix()

sage: M = q[0].matrix().adjugate() + t^2*q[1].matrix().adjugate()

sage: M = M.adjugate().apply_map(lambda m: m.coefficient(t^2))

sage: M == q.Delta_prime_invariant() * Tprime
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).
    ~coefficients(sparse=False)
sage: q.Theta_prime_invariant() == coeffs[1]
True
```

syzygy($\Delta$, $\Theta$, $\Phi$, $\Theta'$, $\Delta'$, $U$, $V$, $T$, $T'$, $J$)
Return the syzygy evaluated on the invariants and covariants.

INPUT:

• $\Delta$, $\Theta$, $\Phi$, $\Theta'$, $\Delta'$, $U$, $V$, $T$, $T'$, $J$ – 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()
```

(continues on next page)
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)

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: 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 Δ 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 Δ′ 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[3]
True
polynomials, release 10.1

(continued from previous page)

```python
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:

```python
sage: R.<a00, a01, a11, a02, a12, 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:

```python
sage: R.<a00, a01, a11, a02, a12, 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 + 98455303871*y^3 + 543715345595/2*x^2 - 306509356021/2*x*y + 755263948570*y^2 - 1118430692650*x - 509948695327/4*y + 3369951531745/8
```

Theta_invariant()
Return the $\Theta$ invariant.

EXAMPLES:

```python
sage: R.<a00, a01, a11, a02, a12, 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 Θ' invariant.

**EXAMPLES:**

```python
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_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: Theta_prime = biquadratic.Theta_prime_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, 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.

3.2. Classical Invariant Theory
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.
• \( \text{scale} \) – 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*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
```

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: f = AlgebraicForm(2, 5, a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5, x0, x1)

sage: transvectant(f, f, 4)
Traceback (most recent call last):
  ... Try:
  sage: transvectant(f, f, 4, scale='none')

Binary quadratic given by -a3^2*x0^2 + a2*a4*x0^2 + a2*a3*x0*x1 - a1*a5*x0*x1 - a2^2*x1^2 + a1*a3*x1^2
```

The additional factors that appear when \( \text{scale}='\text{none}' \) 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: f = AlgebraicForm(2, 5, a0*x1^5 + a1*x1^4*x0 + a2*x1^3*x0^2 + a3*x1^2*x0^3 + a4*x1*x0^4 + a5*x0^5, x0, x1)

sage: transvectant(f, f, 4)
Binary quadratic given by 3/50*a3^2*x0^2 - 4/25*a2*a4*x0^2 + 2/5*a1*a5*x0^2 + 3/50*a2^2*x1^2 - 6/25*a1*a4*x0*x1 + 2*a0*a5*x0^5
```

(continues on next page)
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:

```
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
```
```
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 + (50*y^5 - 8)/25*x^3*h^3 - 12/25*x^2*h^4 + (3*y^5)/5*x*h^5 + (2*y^5)/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[]
```

(continues on next page)
The two non-equivalent cubics $x^3$ and $x^2 \cdot z$ 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
```

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:**

```
sage: from sage.rings.invariants.reconstruction import binary_quadratic_coefficients_from_invariants
sage: quadratic = invariant_theory.binary_form_from_invariants(2, [24]) # indirect doctest
sage: quadratic
Binary quadratic with coefficients (1, -6, 0)
sage: quadratic.discriminant()
24
sage: binary_quadratic_coefficients_from_invariants(0)
(1, 0, 0)
```

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( # indirect doctest
    ...: 5, invs, variables=quintic.variables())
sage: reconstructed
Binary quintic with coefficients (9592267437341790539005557/2441406250000000000,
21492962820762556532300464707/6103515625000000000,
11149651890347700974453304786783/7629394531250000000,
122650775751894638395648991202734239/4768371582031250000000,
32399663094570652847428634593218447/1192092895507812500000000,
150450650364608395841632538558481466127/14901161193874765625000000)
```

We can see that the invariants of the reconstructed form match the ones of the original form by scaling the invariants \( B \) and \( C \):

```python
sage: scale = invs[0]/reconstructed.A_invariant()
True
True
```

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 \( \beta \):

```python
sage: alpha = quintic.alpha_covariant()
sage: beta = quintic.beta_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()
True
```

This can also be seen by computing the \( \alpha \) covariant of the obtained form:

```python
sage: reconstructed.alpha_covariant().coefficient(x1) == 0
True
sage: reconstructed.alpha_covariant().coefficient(x0) != 0
True
```

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: M
0
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, 5279320040130615234375/34359738368, 19797720015048980712890625/1099511627776, -4454487003386020660400390625/17592186044416)

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()

sage: [3, newquintic.B_invariant()*scale^2, newquintic.C_invariant()*scale^3]
[3, 1, 2]

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^*x1*(x0^3-x1^3), x0, x1)
sage: invs = quintic.clebsch_invariants(as_tuple=True)
sage: binary_quintic_coefficients_from_invariants(invs)
(0, 1, 0, 0, 1, 0)
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(x0^2*(x0^*x1^3 + x1^3), x0, x1)
sage: invs = quintic.clebsch_invariants(as_tuple=True)
sage: binary_quintic_coefficients_from_invariants(invs)
(1, 0, 0, 1, 0, 0)

For fields of characteristic 2, 3 or 5, there is no reconstruction implemented. This is part of github issue #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 \texttt{MPolynomialIdeal.groebner_basis()} method on multivariate polynomial objects instead.

\textbf{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].

\textbf{EXAMPLES:}

Consider Katsura-6 with respect to a \texttt{degrevlex} ordering.

\begin{verbatim}
sage: # needs sage.libs.singular sage.rings.finite_rings
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)
sage: g1 = buchberger(I)
sage: g2 = buchberger_improved(I)
sage: g3 = I.groebner_basis()
sage: Ideal(g1).basis_is_groebner()
True
sage: Ideal(g2).basis_is_groebner()
True
sage: Ideal(g3).basis_is_groebner()
True

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

The results are correct:

\begin{verbatim}
sage: Ideal(g1) == Ideal(g2) == Ideal(g3)
True
\end{verbatim}

If \texttt{getVerbose()} 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... ideal
sage: I = sage.rings.ideal.Katsura(P)
\end{verbatim}
\end{verbatim}

(continues on next page)
Ideal \((a + 2b + 2c - 1, a^2 + 2b^2 + 2c^2 - a, 2a^2b + 2b^2c - b)\) of Multivariate Polynomial Ring in \(a, b, c\) over Finite Field of size 127

**sage:** buchberger(I) # random

\((a + 2b + 2c - 1, a^2 + 2b^2 + 2c^2 - a) \Rightarrow -2b^2 - 6bc - 6b^2 - 6c^2 + b + 2c\)

\(G: \text{set}(\{a + 2b + 2c - 1, a^2 + 2b^2 + 2c^2 - a, -2b^2 - 6b^2 - 6c^2 + b + 2c\}])\)

\((a + 2b + 2c - 1, 2a^2b + 2b^2c - b) \Rightarrow -5b^2c - 6c^2 - 63b + 2c\)

\(G: \text{set}\{a + 2b + 2c - 1, 2a^2b + 2b^2c - b, -5b^2c - 6c^2 - 63b + 2c, a^2 + 2b^2 + -2c^2 - a, -2b^2 - 6b^2c - 6c^2 + b + 2c\}])\)

\((2a^2b + 2b^2c - b, a + 2b + 2c - 1) \Rightarrow \emptyset\)

\(G: \text{set}\{a + 2b + 2c - 1, -22c^3 + 24c^2 - 60b - 62c, a^2 + 2b^2 + -2c^2 - a, -2b^2 - 6b^2c - 6c^2 + b + 2c\}])\)

\((2a^2b + 2b^2c - b, -5b^2c - 6c^2 - 63b + 2c) \Rightarrow 0\)

\(G: \text{set}\{a + 2b + 2c - 1, -22c^3 + 24c^2 - 60b - 62c, 2a^2b + 2b^2c - b, -5b^2c - 6c^2 - 63b + 2c\}])\)

\((a + 2b + 2c - 1, -2b^2 - 6bc - 6b^2c + b + 2c) \Rightarrow 0\)

\(G: \text{set}\{a + 2b + 2c - 1, -22c^3 + 24c^2 - 60b - 62c, 2a^2b + 2b^2c - b, -5b^2c - 6c^2 - 63b + 2c\}])\)

\((a^2 + 2b^2 + 2c^2 - a, 2a^2b + 2b^2c - b) \Rightarrow 0\)

\(G: \text{set}\{a + 2b + 2c - 1, -22c^3 + 24c^2 - 60b - 62c, 2a^2b + 2b^2c - b, -5b^2c - 6c^2 - 63b + 2c\}])\)

\((-2a^2 + 2b + 2c - a, -2b^2 - 6b^2c - 6c^2 + b + 2c, -5b^2c - 6c^2 - 63b + 2c) \Rightarrow 0\)

\(G: \text{set}\{a + 2b + 2c - 1, -22c^3 + 24c^2 - 60b - 62c, 2a^2b + 2b^2c - b, -5b^2c - 6c^2 - 63b + 2c\}])\)

\((-2a^2 + 2b^2 - 6b^2c - 6c^2 + b + 2c, -5b^2c - 6c^2 - 63b + 2c) \Rightarrow 0\)

\(G: \text{set}\{a + 2b + 2c - 1, -22c^3 + 24c^2 - 60b - 62c, 2a^2b + 2b^2c - b, -5b^2c - 6c^2 - 63b + 2c\}])\)

\((-2a^2 + 2b^2 + 2c^2 - a, -2b^2 - 6b^2c - 6c^2 + b + 2c, -5b^2c - 6c^2 - 63b + 2c) \Rightarrow 0\)

\(G: \text{set}\{a + 2b + 2c - 1, -22c^3 + 24c^2 - 60b - 62c, 2a^2b + 2b^2c - b, -5b^2c - 6c^2 - 63b + 2c\}])\)
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, -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) => 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, -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) => 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, -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*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*a*b + 2*b*c - b, -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*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, -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*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*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]

The original Buchberger algorithm performs 15 useless reductions to zero for this example:

```
sage: # needs sage.libs.singular sage.rings.finite_rings
sage: gb = buchberger(I)
...
15 reductions to zero.
```

The ‘improved’ Buchberger algorithm in contrast only performs 1 reduction to zero:

```
sage: # needs sage.libs.singular sage.rings.finite_rings
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.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:**

```python
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)  # needs sage.libs.singular
dominant
sage: gb.is_groebner()  # needs sage.libs.singular
True
sage: gb.ideal() == I  # needs sage.libs.singular
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 `set_verbose()` call. Any value >=1 will result in this function printing intermediate Groebner bases.

**EXAMPLES:**

```python
sage: from sage.rings.polynomial.toy_buchberger import buchberger_improved
c
```
**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:
  - \((f_1, \ldots, f_n) = (g_1, \ldots, g_s)\)
  - \(LM(g_i) \neq LM(g_j)\) 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:**

```
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]))
# needs sage.libs.singular
sage: reduced == set([x*y^2, x^2 - 5*y^2])
# needs sage.libs.singular
True
```

```
sage: reduced == inter_reduction(set([2*(x^2 - 5*y^2), x^3]))
# needs sage.libs.singular
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:**

```
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
```

`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: update(set(), set(), x*y*z)
({x*y*z}, set())
sage: G, B = update(set(), set(), x*y*z - 1)
sage: G, B
({x*y*z - 1, x*y^2 - 1}, {(x*y^2 - 1, x*y*z - 1)})
```

### 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)

Generate 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:

```python
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])
# needs sage.modules
[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.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)
```

(continues on next page)
sage: elim_pol(I.groebner_basis())  # needs sage.libs.singular
z^2 - z

sage.rings.polynomial.toy_variety.is_linearly_dependent(polys)
Decide whether the polynomials of polys are linearly dependent.

Here polys is a collection of polynomials.

The algorithm creates a matrix of coefficients of the monomials of 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 \( R^n \), where \( n \) is the number of distinct monomials in polys. There is a zero vector iff there is a linear dependence among polys.

The case where polys=[] is considered to be not linearly dependent.

INPUT:

• polys – a list/tuple of polynomials

OUTPUT:

True if the elements of polys are linearly dependent; False otherwise.

EXAMPLES:

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])  # needs sage.modules
True

sage: p = x*B[0]
sage: is_linearly_dependent(B + [p])  # needs sage.modules
False

sage: is_linearly_dependent([])  # needs sage.modules
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:
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:

sage: # needs sage.modules sage.rings.finite_rings
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: \(-1\))

OUTPUT:

A list \( T \) of triangular sets \( T_\emptyset, T_{-1}, \text{etc.} \).
EXAMPLES:

```python
sage: # needs sage.rings.finite_rings
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())
```

```plaintext
[ [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:

```python
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: I = A.ideal([f, fx, fy])

sage: gb = d_basis(I); gb
```

```plaintext

[ [x - 2020, y - 11313, 22627] ]
```

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()
```

```plaintext
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:

```python
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.:

```python
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).

```python
sage: P.<x, y, z> = PolynomialRing(IntegerRing(), 3, order='degneglex')
sage: I = ideal(x^2 - 3*y, y^3 - x*y, z^3 - x, x^4 - y*z + 1)
sage: I.change_ring(P.change_ring(RationalField())).groebner_basis()
\[ 1 \]
```

However, when we compute the Groebner basis of \( I \) (defined over \( \mathbb{Z} \)), we note that there is a certain integer in the ideal which is not 1:

```python
sage: gb = d_basis(I); gb
\[ z \ldots, y \ldots, x \ldots, 282687803443 \]
```

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 \( p \).:

```python
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() #!
```

(continues on next page)
needs sage.libs.pari

[z + 10380032, y + 3186055, x - 536027]

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

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
- $strat$ – 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]
```

sage.rings.polynomial.toy_d_basis.gpol(g1, g2)

Return the G-Polynomial of $g_1$ and $g_2$.

Let $a_it_i$ be $LT(g_i), a = a_i \ast c_i + a_j \ast c_j$ with $a = GCD(a_i, a_j)$, and $s_i = t/t_i$ with $t = LCM(t_i, t_j)$. Then the G-Polynomial is defined as: $c_1s_1g_1 - c_2s_2g_2$.

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^2*y - z
sage: gpol(f,g)
x^2*y - y
```
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:

```sage
def select(P):
    # Implementation of the normal selection strategy.
    # This is a placeholder for the actual implementation.
```

sage.rings.polynomial.toy_d_basis.spol($g_1$, $g_2$)

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 $s_i = t / t_i$ 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:

- $g_1$ – polynomial
- $g_2$ – polynomial

EXAMPLES:

```sage
def spol($g_1$, $g_2$):
    # Implementation of the S-Polynomial calculation.
    # This is a placeholder for the actual implementation.
```

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:
\begin{center}
\begin{verbatim}
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()
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}
\end{center}
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

\begin{verbatim}
sage: FractionField(PolynomialRing(IntegerRing(), 'x'))
Fraction Field of Univariate Polynomial Ring in x over Integer Ring
sage: FractionField(PolynomialRing(RationalField(),2,'x'))
Fraction Field of Multivariate Polynomial Ring in x0, x1 over Rational Field
\end{verbatim}

Dividing elements often implicitly creates elements of the fraction field:

\begin{verbatim}
sage: x = PolynomialRing(RationalField(), 'x').gen()
sage: f = x/(x+1)
sage: g = x**3/(x+1)
sage: f/g
1/x^2
sage: g/f
x^2
\end{verbatim}

The input must be an integral domain:

\begin{verbatim}
sage: Frac(Integers(4))
Traceback (most recent call last):
  ... TypeError: R must be an integral domain.
\end{verbatim}

**class** `sage.rings.fraction_field.FractionFieldEmbedding`

Bases: `DefaultConvertMap_unique`

The embedding of an integral domain into its field of fractions.

**EXAMPLES:**

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

\begin{verbatim}
sage: R.<x> = QQ[]
sage: R.fraction_field().coerce_map_from(R).is_injective()
True
\end{verbatim}

is_surjective()  
Return whether this map is surjective.

**EXAMPLES:**

\begin{verbatim}
sage: R.<x> = QQ[]
sage: R.fraction_field().coerce_map_from(R).is_surjective()
False
\end{verbatim}
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: 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(R, element_class=class 'sage.rings.fraction_field_element.FractionFieldElement_1poly_field')
Bases: 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()
# optional - sage.rings.finite_rings
sage: K.class_number() # optional - sage.rings.finite_rings
1
```

function_field()
Return the isomorphic function field.

EXAMPLES:

```
sage: R.<t> = GF(5)[]; K = R.fraction_field()  # optional - sage.rings.finite_rings
sage: K = R.fraction_field()  # optional - sage.rings.finite_rings
sage: K.function_field()  # optional - sage.rings.finite_rings
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'])
    #optional - sage.rings.finite_rings
sage: K.maximal_order()
    #optional - sage.rings.finite_rings
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'])
    #optional - sage.rings.finite_rings
sage: K.ring_of_integers()
    #optional - sage.rings.finite_rings
Univariate Polynomial Ring in t over Finite Field of size 5
```

class sage.rings.fraction_field.FractionField_generic(R, element_class=sage.rings.fraction_field_element.FractionFieldElement, category=Category of quotient fields)

Bases: 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.characteristic()  #optional - sage.rings.finite_rings
0
sage: R = Frac(GF(5)['w']); R.characteristic()  #optional - sage.rings.finite_rings
5
```
construction()

EXAMPLES:

```
sage: Frac(ZZ['x']).construction()
(FractionField, Univariate Polynomial Ring in x over Integer Ring)
sage: K = Frac(GF(3)['t'])
    #optional - sage.rings.finite_rings
sage: f, R = K.construction()
    #optional - sage.rings.finite_rings
sage: f(R)
    #optional - sage.rings.finite_rings
Fraction Field of Univariate Polynomial Ring in t
    over Finite Field of size 3
sage: f(R) == K
    #optional - sage.rings.finite_rings
True
```

\[ \text{gen}(i=0) \]

Return the \( i \)-th generator of \( \text{self} \).

EXAMPLES:

```
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
```

\[ \text{is\_exact}() \]

Return if \( \text{self} \) is exact which is if the underlying ring is exact.

EXAMPLES:

```
sage: Frac(ZZ['x']).is_exact()
True
sage: Frac(CDF['x']).is_exact()
False
```

\[ \text{is\_field}(\text{proof}=\text{True}) \]

Return True, since the fraction field is a field.

EXAMPLES:

```
sage: Frac(ZZ).is_field()
True
```

\[ \text{is\_finite}() \]

Tells whether this fraction field is finite.

\[ \text{Note:} \quad \text{A fraction field is finite if and only if the associated integral domain is finite.} \]
EXAMPLES:

```
sage: Frac(QQ['a','b','c']).is_finite()
False
```

\textbf{ngens()}

This is the same as for the parent object.

EXAMPLES:

```
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
```

\textbf{random_element(*args, **kwds)}

Return a random element in this fraction field.

The arguments are passed to the random generator of the underlying ring.

EXAMPLES:

```
sage: F = ZZ['x'].fraction_field()
sage: F.random_element()
# random
(2*x - 8)/(-x^2 + x)
sage: f = F.random_element(degree=5)
sage: f.numerator().degree() == f.denominator().degree()
True
sage: f.denominator().degree() <= 5
True
sage: while f.numerator().degree() != 5:
    ...
    f = F.random_element(degree=5)
```

\textbf{ring()}

Return the ring that this is the fraction field of.

EXAMPLES:

```
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
```

\textbf{some_elements()}

Return some elements in this field.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: R.fraction_field().some_elements()
[0,
  1,
```

(continues on next page)
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: 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:
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)
```

(continues on next page)
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:

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:

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_1poly_field

Bases: 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:

```python
sage: R.<x> = QQ[]
sage: (2 + 2*x) / (4*x) # indirect doctest
(1/2*x + 1/2)/x
```

Compare with:

```python
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:**

```python
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() # optional - sage.libs.pari
[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:**

```python
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[]
 sage: F = R.fraction_field()
 sage: make_element_old(F, {'_FractionFieldElement__numerator': x + y, 
                        '_FractionFieldElement__denominator': x - y})
 (x + y)/(x - y)
```

4.3 Univariate rational functions over prime fields

class sage.rings.fraction_field_FpT.FpT(R, names=None)

Bases: FractionField_1poly_field

This class represents the fraction field $\mathbb{F}_p(T)$ for $2 < p < \sqrt{2^{31} - 1}$.

EXAMPLES:

```python
 sage: R.<T> = GF(71)[]
 sage: K = FractionField(R); K
 Fraction Field of Univariate Polynomial Ring in T over Finite Field of size 71
 sage: 1-1/T
 (T + 70)/T
 sage: parent(1-1/T) is K
 True
```

INTEGER_LIMIT = 46341

iter(bound=None, start=None)

EXAMPLES:

```python
 sage: from sage.rings.fraction_field_FpT import *
 sage: R.<t> = FpT(GF(5)[t])
 sage: list(R.iter(2))[:5]
 [(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: FieldElement

An element of an $FpT$ fraction field.

denom()

Return the denominator of this element, as an element of the polynomial ring.

EXAMPLES:

```python
 sage: K = GF(11)[t].fraction_field()
 sage: t = K.gen(0); a = (t + 1/t)^3 - 1
 sage: a.demden()  
 t^3
```
**denominator()**

Return the denominator of this element, as an element of the polynomial ring.

**EXAMPLES:**

```python
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:**

```python
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:**

```python
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()**

Iterate 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

numer() (continues on next page)

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.numer()
t^6 + 3*t^4 + 10*t^3 + 3*t^2 + 1

numerator() (continues on next page)

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

\textbf{sqrt}(extend=True, all=False)

Return the square root of this element.

\textbf{INPUT:}

- \texttt{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.

- \texttt{all} - bool (default: False); if True, return all square roots of self, instead of just one.

\textbf{EXAMPLES:}

\begin{verbatim}
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^3 + 4)
sage: p.sqrt()^2 == p
True
\end{verbatim}

\textbf{subs}(*args, **kwds)

\textbf{EXAMPLES:}

\begin{verbatim}
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)
\end{verbatim}

\textbf{valuation}(v)

Return the valuation of self at \(v\).

\textbf{EXAMPLES:}

\begin{verbatim}
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
\end{verbatim}

\textbf{class} \texttt{sage.rings.fraction_field_FpT.FpT_Fp_section}

\textbf{Bases:} \texttt{Section}

This class represents the section from GF(p)(t) back to GF(p)[t]

\textbf{EXAMPLES:}
Polynomials, Release 10.1

```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)
<class 'sage.rings.fraction_field_FpT.FpT_Fp_section'>

Warning: Comparison of FpT_Fp_section objects is not currently implemented. See github issue #23469.

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: `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
sage: type(f)
<class 'sage.rings.fraction_field_FpT.FpT_Polyring_section'>

Warning: Comparison of FpT_Polyring_section objects is not currently implemented. See github issue #23469.

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:
sage: K = GF(3)['t'].fraction_field()
sage: I = K.iter(1)
sage: list(I)
[0,
 1,
 2,
 1/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 + 2)/(t + 2)]

class sage.rings.fraction_field_FpT.Fp_FpT_coerce

Bases: RingHomomorphism

This class represents the coercion map from GF(p) to GF(p)(t)

EXAMPLES:

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)
<class '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:

4.3. Univariate rational functions over prime fields 571
sage: R.<t> = GF(5)[]
sage: K = R.fraction_field()
sage: f = K.coerce_map_from(GF(5))
sage: g = f.section(); g
Section map:
  From: Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 5
  To:  Finite Field of size 5
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

class sage.rings.fraction_field_FpT.Polyring_FpT_coerce
Bases: RingHomomorphism

This class represents the coercion map from GF(p)[t] to GF(p)(t)

EXAMPLES:

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)
<class '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
(continues on next page)
sage: g(1/t)
Traceback (most recent call last):
...
ValueError: not integral

class sage.rings.fraction_field_FpT.ZZ_FpT_coerce
Bases: 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)
<class '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
Defn: Section map:
  From: Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 5
  To:   Finite Field of size 5
then
  Lifting map:
  From: Finite Field of size 5
  To:   Integer Ring

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

4.3. Univariate rational functions over prime fields 573
...  
ValueError: not integral

`sage.rings.fraction_field_FpT.unpickle_FpT_element(K, numer, denom)`  
Used for pickling.
CHAPTER
FIVE

LAURENT POLYNOMIALS

5.1 Ring of Laurent Polynomials (base class)

If $R$ is a commutative ring, then the ring of Laurent polynomials in $n$ variables over $R$ is $R[x_1^{\pm 1}, x_2^{\pm 1}, \ldots, x_n^{\pm 1}]$.

AUTHORS:
- David Roe (2008-2-23): created
- David Loeffler (2009-07-10): cleaned up docstrings

class sage.rings.polynomial.laurent_polynomial_ring_base.LaurentPolynomialRing_generic(R)
Bases: CommutativeRing, Parent
Laurent polynomial ring (base class).

EXAMPLES:
This base class inherits from CommutativeRing. Since github issue #11900, it is also initialised as such:

```python
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:

```python
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:

```python
sage: P.<x> = LaurentPolynomialRing(QQ, 1)
sage: P
Multivariate Laurent Polynomial Ring in x over Rational Field
sage: K.<i> = CyclotomicField(4)
  # needs sage.rings.number_field
sage: K.change_ring(K)
  # needs sage.rings.number_field
sage: P.change_ring(K)
```

(continues on next page)
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=None, prec=20, extras=None)

Return the completion of self.

Currently only implemented for the ring of formal Laurent series. The prec variable controls the precision used in the Laurent series ring. If prec is ∞, then this returns a LazyLaurentSeriesRing.

**EXAMPLES:**

```
sage: P.<x> = LaurentPolynomialRing(QQ); P
Univariate Laurent Polynomial Ring in x over Rational Field
sage: PP = P.completion(x); PP
Laurent Series Ring in x over Rational Field
sage: f = 1 - 1/x
sage: PP(f)
-x^-1 + 1
sage: g = 1 / PP(f); g
-x^-1 + 1
sage: g = 1 / PP(f); g
-x^-1 + 1 + 0(x^19)
sage: # needs sage.combinat
sage: PP = P.completion(x, prec=oo); PP
Lazy Laurent Series Ring in x over Rational Field
sage: g = 1 / PP(f); g
-x^-1 + 1 + 0(x^4)
sage: g = 1 / PP(f); g
-x^-1 + 1 + 0(x^4)
sage: 1 / g == f
True
```

**construction()**

Return the construction of self.

**EXAMPLES:**

```
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^-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.

**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**()

Return True if the base ring is exact.

**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**(\(proof=True\))

Return True if self is an integral domain.

**is_exact()**

EXAMPLES:

```python
sage: LaurentPolynomialRing(QQ, 2, 'x').is_exact()
True
```

**is_finite()**

EXAMPLES:

```python
sage: LaurentPolynomialRing(QQ, 2, 'x').is_finite()
False
```

**is_field**(\(proof=True\))

EXAMPLES:

```python
sage: LaurentPolynomialRing(QQ, 2, 'x').is_field()
False
```

**is_integral_domain**(\(proof=True\))

EXAMPLES:

```python
sage: LaurentPolynomialRing(QQ, 2, 'x').is_integral_domain()
True
```

The following used to fail; see github issue #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()
Return True if self is Noetherian.

EXAMPLES:

sage: LaurentPolynomialRing(QQ, 2, 'x').is_noetherian()
Traceback (most recent call last):
  ...
NotImplementedError

krull_dimension()
EXAMPLES:

sage: LaurentPolynomialRing(QQ, 2, 'x').krull_dimension()
Traceback (most recent call last):
  ...
NotImplementedError

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:

sage: LaurentPolynomialRing(QQ, 2, 'x').random_element()
Traceback (most recent call last):
  ...
NotImplementedError

remove_var(var)
EXAMPLES:
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:

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:

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')

5.2. Ring of Laurent Polynomials

If $R$ is a commutative ring, then the ring of Laurent polynomials in $n$ variables over $R$ is $R[x_1^{±1}, x_2^{±1}, \ldots, x_n^{±1}]$. We implement it as a quotient ring

$$R[x_1, y_1, x_2, y_2, \ldots, x_n, y_n]/(x_1y_1 - 1, x_2y_2 - 1, \ldots, x_ny_n - 1).$$

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
Univariate Laurent Polynomial Ring in w over Rational Field
```

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:
Polynomials, Release 10.1

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')

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:

sage: # needs sage.modules
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)  # needs sage.modules
Multivariate Laurent Polynomial Ring in x0, x1, x2, x3, x4, x5, x6, x7, x8, x9 over Rational Field
```

```python
sage: LaurentPolynomialRing(GF(7), 'y', 5)  # needs sage.modules
Multivariate Laurent Polynomial Ring in y0, y1, y2, y3, y4 over Finite Field of size 7
```

```python
sage: LaurentPolynomialRing(QQ, 'y', 3, sparse=True)  # needs sage.modules
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  # needs sage.modules
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()  # needs sage.modules
Defining w0, w1, w2, w3, w4, w5, w6, w7, w8, w9, w10, w11, w12, w13, w14
sage: (w0 + 2*w8 + w13)^2  # needs sage.modules
w0^2 + 4*w0*w8 + 4*w8^2 + 2*w0*w13 + 4*w8*w13 + w13^2
```

Example:

```python
class sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_mpair(R)
Bases: LaurentPolynomialRing_generic

EXAMPLES:

```python
sage: L = LaurentPolynomialRing(QQ,2,'x')  # needs sage.modules
sage: type(L)  # needs sage.modules
<class 'sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_mpair_with_category'>
sage: L == loads(dumps(L))  # needs sage.modules
True
```

**Element**

alias of LaurentPolynomial_mpair

**monomial(*args)**

Return the monomial whose exponents are given in argument.

**EXAMPLES:**
```python
sage: # needs sage.modules
sage: L = LaurentPolynomialRing(QQ, 'x', 2)
sage: L.monomial(-3, 5)
x0^-3*x1^5
sage: L.monomial(1, 1)
x0^1*x1^1
sage: L.monomial(0, 0)
1
sage: L.monomial(-2, -3)
x0^-2*x1^-3
sage: x0, x1 = L.gens()
  # needs sage.modules
sage: L.monomial(-1, 2) == x0^-1 * x1^2  # needs sage.modules
True
sage: L.monomial(1, 2, 3)  # needs sage.modules
Traceback (most recent call last):
...
TypeError: tuple key must have same length as ngens
```

```python
class sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_univariate(R)
Bases: LaurentPolynomialRing_generic

EXAMPLES:

```python
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 *LaurentPolynomial_univariate*

sage.rings.polynomial.laurent_polynomial_ring.from_fraction_field(L, x)

Helper function to construct a Laurent polynomial from an element of its parent's fraction field.

**INPUT:**

- L – an instance of *LaurentPolynomialRing_generic*

- x – an element of the fraction field of L

**OUTPUT:**

An instance of the element class of L. If the denominator fails to be a unit in L an error is raised.

**EXAMPLES:**

```python
sage: # needs sage.modules
sage: from sage.rings.polynomial.laurent_polynomial_ring import from_fraction_field
sage: L.<x, y> = LaurentPolynomialRing(ZZ)
```

(continues on next page)
sage: F = L.fraction_field()
sage: xi = F(~x)
sage: from_fraction_field(L, xi) == ~x
True

sage.rings.polynomial.laurent_polynomial_ring.is_LaurentPolynomialRing(R)
Return 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)
doctest:warning...
DeprecationWarning: is_LaurentPolynomialRing is deprecated; use isinstance(...,
sage.rings.polynomial.laurent_polynomial_ring_base.LaurentPolynomialRing_generic)␣
   →instead
See https://github.com/sagemath/sage/issues/35229 for details.
False

sage: R = LaurentPolynomialRing(QQ,3,'x')
# needs sage.modules
sage: is_LaurentPolynomialRing(R)
# needs sage.modules
True

5.3 Elements of Laurent polynomial rings

class sage.rings.polynomial.laurent_polynomial.LaurentPolynomial
Bases: 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 github issue #22277 is fixed:

sage: R.<x, y> = LaurentPolynomialRing(QQ) # needs sage.modules
sage: a = 2*x^2 + 3*x^3 + 4*x^-1
sage: a.change_ring(GF(3))
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: # needs sage.rings.finite_rings
sage: k.<a> = GF(9)
sage: R.<x> = LaurentPolynomialRing(k)
sage: f = x^4 + a
sage: f.map_coefficients(lambda a: a + 1)
(a + 1) + (a + 1)*x
sage: R.<x,y> = LaurentPolynomialRing(k, 2)
sage: f = x^4 + 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: # needs sage.modules sage.rings.finite_rings
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 in r of size 3^2
sage: g = f.map_coefficients(h, new_base_ring=GF(3)); 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_univariate
Bases: LaurentPolynomial

A univariate Laurent polynomial in the form of \( t^n \cdot f \) where \( f \) is a polynomial in \( t \).

INPUT:

- \( \text{parent} \) – a Laurent polynomial ring
- \( \text{f} \) – a polynomial (or something can be coerced to one)
- \( \text{n} \) – (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
```

5.3. Elements of Laurent polynomial rings
```python
sage: R.<t> = PolynomialRing(ZZ)
sage: S.<x> = LaurentPolynomialRing(R)
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()  # needs sage.libs.pari
(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 + t^{-1}), (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
```
```python
sage: t.is_square()
False
sage: (t**-2).is_square()
True
```

Usage of the root option:

```python
sage: p = (1 + t^-1 - 2*t^3)
sage: p.is_square(root=True)
(False, None)
sage: (p**2).is_square(root=True)
(True, -t^-1 - 1 + 2*t^3)
```

The answer is dependent of the base ring:

```python
sage: # needs sage.rings.number_field
sage: S.<u> = LaurentPolynomialRing(QQbar)
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:**

```python
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
sage: R.<s> = LaurentPolynomialRing(ZZ)
sage: g = 2*s
sage: g.is_unit()
False
sage: 1/g
1/2*s^-1
```

**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:**

```python
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:

sage: R.<x> = LaurentPolynomialRing(ZZ)
sage: f = x^3 - 1
sage: f.number_of_terms()
2
sage: R(0).number_of_terms()
0
sage: f = (x+1)^100
sage: f.number_of_terms()
101

The method hamming_weight() is an alias:

sage: f.hamming_weight()
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(other)  
Divide self by other and return a quotient q and a remainder r such that self == q * other + r.

EXAMPLES:

sage: R.<t> = LaurentPolynomialRing(QQ)
sage: (t^-3 - t^3).quo_rem(t^-1 - t)
(t^-2 + 1 + t^2, 0)
sage: (t^-2 + 3 + t).quo_rem(t^-4)
(t^2 + 3*t^4 + t^5, 0)
sage: num = t^-2 + t
sage: den = t^-2 + 1

(continues on next page)
Let's consider an example of a Laurent polynomial ring in SageMath. The output is as follows:

```plaintext
sage: R.<t> = LaurentPolynomialRing(QQ)
sage: f = 3*t^-2 - t^-1 + 3 + t^2
```

This defines a Laurent polynomial ring over the rational numbers with variable `t`. Let's compute the residue of the polynomial `f`:

```plaintext
sage: f.residue()
-1
```

The residue of a Laurent polynomial is defined as the coefficient of `t^-1`. In this case, the coefficient of `t^-1` in `f` is `-1`.

Similarly, the shift method of a Laurent polynomial allows us to multiply the polynomial by a power of `t` without changing the polynomial itself. For instance:

```plaintext
sage: f.shift(10)
t^6 + 4*t^8 + 6*t^10 + 4*t^12 + t^14
```

The `shift` method shifts the polynomial to the left by 10, effectively multiplying it by `t^10`. The valuation method returns the valuation of a Laurent polynomial, which is the degree of the polynomial plus the valuation of the constant term. For example:

```plaintext
sage: f.valuation()
2
```

The `valuation` method returns the valuation of `f`, which is 2 in this case.

### 5.3. Elements of Laurent polynomial rings

The `valuation` method returns the valuation of a Laurent polynomial, which is the degree of the polynomial plus the valuation of the constant term. For example:

```plaintext
sage: R.<x> = LaurentPolynomialRing(QQ)
sage: f = 1/x^12 + x^3 + x^5 + x^9
```

This defines another Laurent polynomial ring over the rational numbers with variable `x`. Let's compute the truncated version of `f`:

```plaintext
sage: f.truncate(10)
x^-12 + x^3 + x^5 + x^9
```

The `truncate` method returns a polynomial with degree at most 9 whose `j`-th coefficients agree with `f` for all `j < 9`.

In summary, the residue, shift, truncate, and valuation methods provide useful tools for manipulating Laurent polynomials in SageMath.
sage: R.<x> = LaurentPolynomialRing(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.

EXAMPLES:

sage: R.<x> = LaurentPolynomialRing(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()
()
5.4.2 Various

AUTHORS:
• Daniel Krenn (2016)

ACKNOWLEDGEMENT:
• Daniel Krenn is supported by the Austrian Science Fund (FWF): P 24644-N26.

5.4.3 Functions

`sage.rings.polynomial.omega.MacMahonOmega(var, expression, denominator=None, op=<built-in function ge>, Factorization_sort=False, Factorization_simplify=True)`

Return $\Omega_{op}$ of expression with respect to var.

To be more precise, calculate

$$\frac{\Omega_{op}}{d_1 \ldots d_n}$$

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
```

(continues on next page)
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*y^2 + 1)^-1
sage: MacMahonOmega(mu, 1, [1 - x*mu, 1 - y*mu, 1 - z/mu])
(x*y + 1) * (-x + 1)^-1 * (-x*y^2 + 1)^-1
sage: MacMahonOmega(mu, 1, [1 - x*mu, 1 - y*mu, 1 - z/mu^2])
(-x*y*z^2 - x*y*z + x*z + 1) * (-x + 1)^-1 * (-y + 1)^-1 * (-x*z^2 + 1)^-1 * (-y*z + 1)^-1
sage: MacMahonOmega(mu, 1, [1 - x*mu, 1 - y*mu, 1 - z/mu, 1 - w/mu])
(x*y^2 - x*y + y + 1)^-1 * (-x + 1)^-1 * (-x*y + 1)^-1
sage: MacMahonOmega(mu, 1, [1 - x*mu, 1 - y*mu, 1 - z*mu, 1 - w/mu])
(x*y*z*w + x*y*z*w - x*y*z*w - x*y*z*w + 1)^-1 * (-x + 1)^-1 * (-y + 1)^-1 * (-x*z + 1)^-1 * (-x*w + 1)^-1 * (-y*w + 1)^-1
sage: MacMahonOmega(mu, 1, [1 - x*mu, 1 - y*mu, 1 - z*mu, 1 - w/mu])
(x*y^2 - x*y + y + 1)^-1 * (-x + 1)^-1 * (-x*y + 1)^-1

We demonstrate the different allowed input variants:

sage: MacMahonOmega(mu, mu^-2, [1 - x*mu, 1 - y/mu])
(x^2 * (-x + 1)^-1 * (-x*y + 1)^-1
sage: MacMahonOmega(mu, mu^-1, [1 - x*mu, 1 - y/mu])
(x * (-x + 1)^-1 * (-x*y + 1)^-1
sage: MacMahonOmega(mu, mu, [1 - x*mu, 1 - y/mu])
(-x*y + y + 1) * (-x + 1)^-1 * (-x*y + 1)^-1
sage: MacMahonOmega(mu, mu^2, [1 - x*mu, 1 - y/mu])
(-x*y^2 - x*y + y^2 + y + 1) * (-x + 1)^-1 * (-x*y + 1)^-1
sage: MacMahonOmega(mu, mu**2,
....: Factorization([(1 - x*mu, 1), (1 - y/mu, 1)]))
(-x*y^2 - x*y + y^2 + y + 1) * (-x + 1)^-1 * (-x*y + 1)^-1
sage: MacMahonOmega(mu, mu**2, [1 - x*mu, 1 - y/mu])
(-x*y^2 - x*y + y^2 + y + 1) * (-x + 1)^-1 * (-x*y + 1)^-1
sage: MacMahonOmega(mu, mu**2, (1 - x*mu)*(1 - y/mu)) # not tested because not
˓→fully implemented
(-x*y^2 - x*y + y^2 + y + 1) * (-x + 1)^-1 * (-x*y + 1)^-1
sage: MacMahonOmega(mu, mu**2 / ((1 - x*mu)*(1 - y/mu))) # not tested because not
˓→fully implemented
(-x*y^2 - x*y + y^2 + y + 1) * (-x + 1)^-1 * (-x*y + 1)^-1

sage.rings.polynomial.omega.Omega_ge(exponents)
Return $\Omega_\ge$ of the expression specified by the input.

To be more precise, calculate

$$\Omega_\ge = \frac{\mu^a}{(1 - z_0 \mu^e_0) \cdots (1 - z_{n-1} \mu^e_{n-1})}$$

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

• 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 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, z0^2*z1))
(sage: Omega_ge(0, (3, -1))
(z0^2*z1 + z0*z1 + 1, (z0, z0^3*z1))
(sage: Omega_ge(0, (4, -1))
Polynomials, Release 10.1

(continued from previous page)

\[(z_0^*z_1^3 + z_0^*z_1^2 + z_0^*z_1 + 1, (z_0, z_0^*z_1^4))\]

\texttt{sage: Omega_ge(0, (1, 1, -2))}
\[-z_0^*z_1^2z_2 - z_0^*z_1^2z_2 + z_0^*z_1^2z_2 + 1, (z_0, z_1, z_0^*z_1^2, z_1^2z_2)]\)

\texttt{sage: Omega_ge(0, (2, -1, -1))}
\[(z_0^*z_1^2z_2 + z_0^*z_1 + z_0^*z_2 + 1, (z_0, z_0^*z_1^2, z_0^*z_2^2))\]

\texttt{sage: Omega_ge(0, (2, 1, -1))}
\[-(z_0^*z_1^2z_2^2 - z_0^*z_1^2z_2 + z_0^*z_2 + 1, (z_0, z_1, z_0^*z_2^2, z_1^2z_2))\]

\texttt{sage: Omega_ge(0, (2, -2))}
\[-z_0^*z_1 + 1, (z_0, z_0^*z_1, z_0^*z_1)]\)

\texttt{sage: Omega_ge(0, (2, -3))}
\[(z_0^*z_1^3 + 1, (z_0, z_0^*z_1^2))\]

\texttt{sage: Omega_ge(0, (3, 1, -3))}
\[-z_0^3z_1^3z_2^3 + 2z_0^2z_1^3z_2^2 - z_0^3z_1z_2^2 + z_0^2z_2^2 - 2z_0z_2 + 1, (z_0, z_1, z_0^*z_2, z_0^*z_2, z_1^3z_2)]\)

\texttt{sage: Omega_ge(0, (3, 6, -1))}
\[-z_0^*z_1^2z_2^8 - z_0^*z_1^2z_2^7 - z_0^*z_1^2z_2^6 - z_0^*z_1^2z_2^5 - z_0^*z_1^2z_2^4 + z_1^2z_2^5 + z_0^*z_1^2z_2^2 + z_1^2z_2^4 - z_0^*z_1^2z_2^2 + z_1^2z_2^3 - z_0^*z_1^2z_2 + z_0^*z_2^2 + z_1^2z_2^2 + z_0^*z_2 + z_1^2z_2 + 1, (z_0, z_1, z_0^*z_2^3, z_1^2z_2^6))]\)

\texttt{sage.rings.polynomial.omega.homogeneous_symmetric_function}(j, x)

Return a complete homogeneous symmetric polynomial (Wikipedia article Complete_homogeneous_symmetric_polynomial).

\textbf{INPUT:}

- \(j\) – the degree as a nonnegative integer
- \(x\) – an iterable of variables

\textbf{OUTPUT:}

A polynomial of the common parent of all entries of \(x\)

\textbf{EXAMPLES:}

\texttt{sage: from sage.rings.polynomial.omega import homogeneous_symmetric_function}
\texttt{sage: P = PolynomialRing(ZZ, 'X', 3)}
\texttt{sage: homogeneous_symmetric_function(0, P.gens())}
\[
1 \\
\texttt{sage: homogeneous_symmetric_function}(1, P.gens())
\[
X_0 + X_1 + X_2 \\
\texttt{sage: homogeneous_symmetric_function}(2, P.gens())
\[
X_0^2 + X_0*X_1 + X_1^2 + X_0^*X_2 + X_1*X_2 + X_2^2 \\
\texttt{sage: homogeneous_symmetric_function}(3, P.gens())
\[
X_0^3 + X_0^2*X_1 + X_0*X_1^2 + X_1^3 + X_0^2*X_2 + X_0*X_1*X_2 + X_1*X_2^2 + X_2^3 \\
\texttt{sage.rings.polynomial.omega.partition}(items, predicate=<class 'bool'>)

Split \(items\) into two parts by the given \(predicate\).

\textbf{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 $x_0, x_1, x_2, \ldots, y_0, y_1, y_2, \ldots, \ldots$. 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 finite 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:

$$X \cdot \text{gen}(i)[m] > X \cdot \text{gen}(j)[n] \text{ if and only if } i<j \text{ or } (i==j \text{ and } m>n)$$

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:

```
sage: X.<x,y> = InfinitePolynomialRing(ZZ, implementation='sparse')
sage: A.<alpha,beta> = InfinitePolynomialRing(QQ, order='deglex')
sage: f = x[5] + 2; f
```
It has some advantages to have an underlying ring that is not univariate. Hence, we always have at least two variables:

```python
sage: g._p.parent()
Multivariate Polynomial Ring in y_1, y_0 over Integer Ring
```

Of course, we provide the usual polynomial arithmetic:

```python
sage: f + g
x_5 + 3*y_1 + 2
sage: p = x[10]^2*(f+g); p
x_10^2*x_5 + 3*x_10^2*y_1 + 2*x_10^2
sage: p2 = alpha[10]^2*(f2+g2); p2
alpha_10^2*alpha_5 + 3*alpha_10^2*beta_1 + 2*alpha_10^2
```

There is a permutation action on the variables, by permuting positive variable indices:

```python
sage: P = Permutation(((10,1)))
sage: p^P
x_5*x_1^2 + 3*x_1^2*y_10 + 2*x_1^2
sage: p2^P
alpha_5*alpha_1^2 + 3*alpha_1^2*beta_10 + 2*alpha_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:

```python
sage: J = A * (alpha[1]*beta[2])
sage: 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:
sage: A.<a,b,c> = InfinitePolynomialRing(ZZ)
sage: B.<b,c,d> = InfinitePolynomialRing(A)
sage: B
Infinite polynomial ring in a, b, c, d over Integer Ring

This is also allowed if finite polynomial rings are involved:

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:

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’.

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:

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:

sage: X.<y_1,x> = ZZ[]
sage: Y.<y,z> = 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_1 and y_2 would be in opposite order in an Infinite Polynomial Ring
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 do not match, there is no simplified construction available:

```
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:

```
sage: R.<a,b> = InfinitePolynomialRing(ZZ)
sage: D = R.gens_dict() # indirect doctest
sage: D
GenDict of Infinite polynomial ring in a, b over Integer Ring
sage: D['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)
-1/2*a_7 + 3*a_3*b_5
```

next()

Return a dictionary that can be used to interpret strings in the base ring of self.

EXAMPLES:

```
sage: R.<a,b> = InfinitePolynomialRing(QQ['t'])
sage: D = R.gens_dict()
sage: D
GenDict of Infinite polynomial ring in a, b over Univariate Polynomial Ring in t over Rational Field
sage: next(D)
GenDict of Univariate Polynomial Ring in t over Rational Field
sage: sage_eval('t^2', 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:

```python
sage: R.<a,b> = InfinitePolynomialRing(ZZ)
sage: D = R.gens_dict() # indirect doctest
sage: D
[InfiniteGenDict defined by ['a', 'b'], {'1': 1}]

sage: 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[0])
-1/2*a_7 + 3*a_3*b_5
```

class sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialGen

```
Bases: SageObject

This class provides the object which is responsible for returning variables in an infinite polynomial ring (implemented in __getitem__()).

EXAMPLES:

```python
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: 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:

```python
sage: A.<a> = InfinitePolynomialRing(QQ)
sage: B.<b> = InfinitePolynomialRing(A)
sage: B.construction()
[InfPoly[['a,b'], 'lex', 'dense'], Rational Field]

sage: R.<a,b> = Infinite PolynomialRing(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\((R, \text{names}=('x',), \text{order}='lex', \text{implementation}='dense')\)

Creates a key which uniquely defines the infinite polynomial ring.

create_object\((\text{version, key})\)

Returns the infinite polynomial ring corresponding to the key key.

class sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing_dense\((R, \text{names}, \text{order})\)

Bases: 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()  

Returns the construction of self.

OUTPUT:

A pair \(F, R\), where \(F\) is a construction functor and \(R\) is a ring, so that \(F(R) = self\).

EXAMPLES:

```python
sage: R.<x,y> = InfinitePolynomialRing(GF(5))
sage: R.construction()
[InfPoly{['x', 'y'], 'lex', 'dense'}, Finite Field of size 5]
```

polynomial_ring()  

Returns 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:

```python
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\((R)\)

Returns 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 [github issue #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:** `CommutativeRing`

Sparse implementation of Infinite Polynomial Rings.

An Infinite Polynomial Ring with generators $x_0, y_0, \ldots$ 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 ‘degrevelx’; 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:

```python
sage: X.<x,y> = InfinitePolynomialRing(GF(25,'a'))  # needs sage.rings.finite_rings
sage: X  # needs sage.rings.finite_rings
Infinite polynomial ring in x, y over Finite Field in a of size 5^2
sage: X.characteristic()  # needs sage.rings.finite_rings
5
```

`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 `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()  # 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:
**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:**

```python
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.

**key_basis()**

Return the basis of self given by key polynomials.

**EXAMPLES:**

```python
sage: R.<x> = InfinitePolynomialRing(GF(2))
sage: R.key_basis()
# needs sage.combinat
Key polynomial basis over Finite Field of size 2
```

**krull_dimension(***args, **kwds)**

Return Infinity, since polynomial rings in infinitely many variables have infinite Krull dimension.

**EXAMPLES:**

```python
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:**

```python
sage: X.<x> = InfinitePolynomialRing(ZZ)
sage: X.ngens()
1
```
Polynomials, Release 10.1

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 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)
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 github issue #10468:

sage: R.<x,y> = InfinitePolynomialRing(QQ)
sage: R.tensor_with_ring(QQ) is R
True

varname_key(x)

Key for comparison of variable names.

INPUT:
x – a string of the form a+’_’+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 ⇐⇒ the string x.split('_')[0] is later in the list of generator names of self than y.
split('_')[0], or (x.split('_')[0]==y.split('_')[0] and int(x.split('_')[1])<int(y.
split('_')[1]))

EXAMPLES:

```
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 $x_*, y_*, ...$, so that the variables are of the form $x_0, x_1, x_2, ..., y_0, y_1, y_2, ..., ...$
(see `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:

$X$.gen(i)[m] > X.gen(j)[n] ⇐⇒ i<j, or i==j 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 `classical` polynomial. This holds, e.g., when applying the `latex` function:
Polynomials, Release 10.1

```
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 $c$.

If you want to substitute variables you can use the standard polynomial methods, such as `subs()`:

```
sage: R.<x,y> = InfinitePolynomialRing(QQ)
sage: f = x[1] + x[1]*x[2]*x[3]
sage: f.subs({x[1]: x[0]})
x_3*x_2*x_0 + x_0

sage: g = x[0] + x[1] + y[0]
sage: g.subs({x[0]: y[0]})
x_1 + 2*y_0
```

### class `sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial(A,p)`

Bases: `CommutativePolynomial`

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 `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:**

```
sage: from sage.rings.polynomial.infinite_polynomial_element import...
   →InfinitePolynomial
sage: X.<alpha> = InfinitePolynomialRing(ZZ)
sage: P.<alpha_1, alpha_2> = ZZ[]
```

Currently, $P$ and $X._P$ (the underlying polynomial ring of $X$) both have two variables:

```
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:
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
```

**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:

```python
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: a.coefficient(x[0])
2*x_1
sage: a.coefficient(x[1])
2*x_0 + 1
sage: a.coefficient(x[2])
1
sage: a.coefficient({x[0]:1, x[1]:1})
2
```

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:

```
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:

```
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:

```
sage: R.<x> = InfinitePolynomialRing(QQbar)
# needs sage.rings.number_field
sage: (x[0] + x[1]).is_nilpotent()  # needs sage.rings.number_field
False
sage: R(0).is_nilpotent()  # needs sage.rings.number_field
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 github issue #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]^3*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:

```sage
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: p.max_index()
4
sage: x[0].max_index()
0
sage: X(10).max_index()
-1
```

dpolynomial()
Return the underlying polynomial.

EXAMPLES:

```sage
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 $T \in X$ 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:
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])  # needs sage.libs.singular
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)  # needs sage.libs.singular
x_3^3*y_2
```

Last, we demonstrate the report option:

```
sage: p.reduce(I, tailreduce=True, report=True)  # needs sage.libs.singular
: 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:

```python
sage: X.<x,y> = InfinitePolynomialRing(QQ,implementation='sparse')
sage: p = x[1]*y[100] + x[50]*y[1000]
sage: p.squeezed()
x_2*y_4 + x_1*y_3
```

**stretch**($k$)

Stretch self by a given factor.

**INPUT:**

$k$ – an integer.

**OUTPUT:**

Replace $v_n$ with $v_{nk}$ for all generators $v_n$ occurring in self.

**EXAMPLES:**

```python
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: a.stretch(2)
x_4 + 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
```

**subs**(*fixed=None, **kwargs*)

Substitute variables in self.

**INPUT:**

• *fixed* – (optional) dict with {variable: value} pairs
  • **kwargs – named parameters

**OUTPUT:**

the resulting substitution

**EXAMPLES:**

```python
sage: R.<x,y> = InfinitePolynomialRing(QQ)
sage: f = x[1] + x[1]*x[2]*x[3]

Passing fixed={x[1]: x[0]}. Note that the keys may be given using the generators of the infinite polynomial ring or as a string:

```python
sage: f.subs({x[1]: x[0]})
x_3*x_2*x_0 + x_0
sage: f.subs({'x_1': x[0]})
x_3*x_2*x_0 + x_0
```

Passing the variables as names parameters:
The substitution returns the original polynomial if you try to substitute a variable not present:

\[
\text{sage: } g = x[0] + x[1] \\
\text{sage: } g.subs({y[0]: x[0]}) \\
\text{x}_1 + x_0
\]

The substitution can also handle matrices:

\[
\text{sage: } \# \text{ needs sage.modules} \\
\text{sage: } M = \text{matrix}([[1,0], [0,2]]) \\
\text{sage: } N = \text{matrix}([[0,3], [4,0]]) \\
\text{sage: } g = x[0]^2 + 3*x[1] \\
\text{sage: } g.subs({}'x_0': M}) \\
\begin{bmatrix}
3*x_1 + 1 & 0 \\
0 & 3*x_1 + 4
\end{bmatrix} \\
\text{sage: } g.subs({x[0]: M, x[1]: N}) \\
\begin{bmatrix}
1 & 9 \\
12 & 4
\end{bmatrix}
\]

If you pass both fixed and kwargs, any conflicts will defer to fixed:

\[
\text{sage: } R.<x,y> = \text{InfinitePolynomialRing}(\text{QQ}) \\
\text{sage: } f = x[0] \\
\text{sage: } f.subs({x[0]: 1}) \\
1 \\
\text{sage: } f.subs(x_0=5) \\
5 \\
\text{sage: } f.subs({x[0]: 1}, x_0=5) \\
1
\]

**symmetric_cancellation_order**(other)

Comparison of leading terms by Symmetric Cancellation Order, \(<_\text{sc}\>\).

INPUT:

self, other – two Infinite Polynomials

ASSUMPTION:

Both Infinite Polynomials are non-zero.

OUTPUT:

\((c, \text{ sigma}, w)\), where

- \(c = -1,0,1\), 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

- \(\text{sigma}\) is a permutation witnessing self \(<_\text{sc}\>\) other (resp. self \(>_\text{sc}\) other) or is 1 if self.\lm() == other.\lm()

- \(w\) is 1 or is a term so that \(w*\text{self.}\lt()^\text{sigma} == \text{other.}\lt()\) if \(c \leq 0\), and \(w*\text{other.}\lt()^\text{sigma} == \text{self.}\lt()\) if \(c = 1\)
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.

EXAMPLES:

```sage
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: (x[2]*x[1]).symmetric_cancellation_order(x[2]**2)
(None, 1, 1)
sage: (x[2]*x[1]).symmetric_cancellation_order(x[2]**x[3]**y[1])
(-1, [2, 3, 1], y_1)
sage: (x[2]*x[1]*y[1]).symmetric_cancellation_order(x[2]**x[3]**y[2])
(-1, [2, 3, 1], 1)
```

tail()

The tail of self (this is self minus its leading term).

EXAMPLES:

```sage
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: p = 2*x[10]*y[30] + 3*x[10]**y[1]**3*x[1]**2
sage: p.tail()
2*x_10*y_30
```

variables()

Return the variables occurring in self (tuple of elements of some polynomial ring).

EXAMPLES:

```sage
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: p.variables()
(x_3, x_2, x_1)
sage: x[1].variables()
(x_1,)
sage: X(1).variables()
()```

```
class sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial_dense(A, p)
Bases: InfinitePolynomial

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.

class sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial_sparse(A, p)
Bases: InfinitePolynomial

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:

```sage
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
```

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.

```sage
sage: R.<x> = InfinitePolynomialRing(QQ)
sage: I = R.ideal([x[1]*x[2] + x[3]])
```

Note that \( I \) is not a symmetric Groebner basis:

```sage
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 \( G \), \( x_1^2 \) is equal to \( x_1 \) in \( Q \). Indeed, we have
class sage.rings.polynomial.symmetric_ideal.SymmetricIdeal(ring, gens, coerce=True)

    Bases: Ideal_generic

    Ideal in an Infinite Polynomial Ring, invariant under permutation of variable indices

THEORY:

An Infinite Polynomial Ring with finitely many generators \( x_*, y_* \ldots \) over a field \( F \) is a free commutative \( F \)-algebra generated by infinitely many ‘variables’ \( x_0, x_1, x_2, \ldots, y_0, y_1, y_2, \ldots \). We refer to the natural number \( n \) as the index of the variable \( x_n \). See more detailed description at \( \text{infinite_polynomial_ring} \)

Infinite Polynomial Rings are equipped with a permutation action by permuting positive variable indices, i.e., 
\[
x^P_n = x_{P(n)}, \quad y^P_n = y_{P(n)},
\]
for any permutation \( P \). Note that the variables \( x_0, y_0, \ldots \) 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 \( \mathfrak{S}_\infty \) denotes the symmetric group of 1, 2, ..., then a Symmetric Ideal is a right \( X[\mathfrak{S}_\infty] \)-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 \( X[\mathfrak{S}_\infty] \)-module).

Moreover, if \( X \) is equipped with a lexicographic monomial ordering with \( x_1 < x_2 < x_3 \ldots \) 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 \( \text{groebner_basis()} \) 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:

```python
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: I == loads(dumps(I))
True
sage: latex(I)
\left(x_{1} y_{2} y_{1} + 2 x_{1} y_{2}\right)
```

The default ordering is lexicographic. We now compute a Groebner basis:

```python
sage: J = I.groebner_basis(); J  # about 3 seconds  #_


```

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 \( J \):

```python
sage: J = I.groebner_basis(); J  # about 3 seconds  #_

```
The Groebner basis is not point-wise invariant under permutation:

```python
sage: # needs sage.combinat
sage: P = Permutation([2, 1])
```

```python
sage: J[2]^P
```

### Note

Since I is not a Groebner basis, it is no surprise that it cannot detect ideal membership:

```python
sage: [p.reduce(I) for p in J]  # needs sage.combinat
```

### Note

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.

```python
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:

```sage
X.<x,y> = InfinitePolynomialRing(QQ)
sage: I1 = X * (x[1] + x[2], x[1]*x[2])
sage: I1.groebner_basis()
```

(continues on next page)
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:

```plaintext
sage: I2.groebner_basis(use_full_group=True)
```

Last, we demonstrate how the report on the progress of computations looks like:

```plaintext
sage: I1.groebner_basis(report=True, reduced=True)
```

6.3. Symmetric Ideals of Infinite Polynomial Rings
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:

```python
sage: R.<x,y> = InfinitePolynomialRing(ZZ)
sage: I = R * [x[1] + x[2], y[1]]
sage: I.groebner_basis() # needs sage.combinat
Traceback (most recent call last):
  ...
TypeError: The base ring (= Integer Ring) must be a field
```

`interreducedbasis()`

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:

```python
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: I = X * (x[1] + x[2], x[1]*x[2])
```
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:

```
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: I = X * (x[1] + x[2], x[1]*x[2])
sage: I.interreduction()                   # Needs sage.combinat
Symmetric Ideal (-x_1^2, x_2 + x_1) of
Infinite polynomial ring in x over Rational Field
```

Here, we show the report option:

```
sage: I.interreduction(report=True)       # Needs sage.combinat
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 RStrat:
Polynomials, Release 10.1

`.sage:

```python
from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
R = SymmetricReductionStrategy(X); R
```  

Symmetric Reduction Strategy in Infinite polynomial ring in x over Rational Field

```
I.interreduction(RStrat=R)
```

Symmetric Ideal (-x_1^2, x_2 + x_1) of Infinite polynomial ring in x over Rational Field

```
R
```

Symmetric Reduction Strategy in Infinite polynomial ring in x over Rational Field, modulo x_1^2, x_2 + x_1

```
I.interreduction(RStrat=R)
```

Symmetric Ideal (x_2 + x_1) of Infinite polynomial ring in x over Rational Field

### is_maximal()

Answer 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 `NotImplementedError` is raised if the base ring is not a field, since symmetric Groebner bases are not implemented in this setting.

**EXAMPLES:**

```
sage: R.<x,y> = InfinitePolynomialRing(QQ)
sage: I = R * I.groebner_basis(); I
```

Symmetric Ideal (x_1 + y_2, x_2 - y_1) of Infinite polynomial ring in x, y over Rational Field

```
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: I = R * I.groebner_basis(); I
```

Symmetric Ideal (x_1 + y_2, x_2 - y_1) of Infinite polynomial ring in x, y over Rational Field

```
sage: I.is_maximal()
```

True
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: 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
```

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 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: 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: I.reduce([x[2]^2*y[1]])
needs sage.combinat
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: 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)
needs sage.combinat
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: 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: X.<x> = InfinitePolynomialRing(QQ)
sage: I = X * (x[1] + x[2], x[1]*x[2])
sage: I.symmetric_basis()
needs sage.combinat
[x_1^2, x_2 + x_1]

symmetrisation(N=None, tailreduce=False, report=None, use_full_group=False)
Apply permutations to the generators of self and interreduce.

INPUT:

• N – (integer, default None) Apply permutations in Sym(N). If it is not given then it will be replaced by the maximal variable index occurring in the generators of self.interreduction().squeezed().

• tailreduce – (bool, default False) If True, perform tail reductions.

• report – (object, default None) If not None, report on the progress of computations.

• use_full_group (optional) – If True, apply all elements of Sym(N) to the generators of self (this is what [AB2008] originally suggests). The default is to apply all elementary transpositions to the generators of self.squeezed(), interreduce, and repeat until the result stabilises, which is often much faster than applying all of Sym(N), and we are convinced that both methods yield the same result.
OUTPUT:

A symmetrically interreduced symmetric ideal with respect to which any \( Sym(N) \)-translate of a generator of \texttt{self} is symmetrically reducible, where by default \( N \) is the maximal variable index that occurs in the generators of \texttt{self.interreduction().squeezed}.

NOTE:

If \( I \) is a symmetric ideal whose generators are monomials, then \texttt{I.symmetrisation()} is its reduced Groebner basis. It should be noted that without symmetrisation, monomial generators, in general, do not form a Groebner basis.

EXAMPLES:

```sage
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: I = X * (x[1] + x[2], x[1]*x[2])
sage: I.symmetrisation()
# needs sage.combinat
Symmetric Ideal (-x_1^2, x_2 + x_1) of
Infinite polynomial ring in x over Rational Field
sage: I.symmetrisation(N=3)
# needs sage.combinat
Symmetric Ideal (-2*x_1) of Infinite polynomial ring in x over Rational Field
sage: I.symmetrisation(N=3, use_full_group=True)
# needs sage.combinat
Symmetric Ideal (-2*x_1) of Infinite polynomial ring in x over Rational Field
```

6.4 Symmetric Reduction of Infinite Polynomials

\texttt{SymmetricReductionStrategy} provides a framework for efficient symmetric reduction of Infinite Polynomials, see \texttt{infinite_polynomial_element}.

AUTHORS:

- Simon King \(<\text{simon.king@nuigalway.ie}>\)

THEORY:

According to M. Aschenbrenner and C. Hillar [AB2007], Symmetric 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 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 \texttt{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:

```sage
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:

```sage
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: S = SymmetricReductionStrategy(X, [y[2]^2*x[1]])
sage: S
Symmetric Reduction Strategy in
Infinite polynomial ring in x, y over Rational Field, modulo
x_1*y_2^2
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 \( S \):

```sage
sage: S.add_generator(y[1]^2*x[2])
sage: S
Symmetric Reduction Strategy in
Infinite polynomial ring in x, y over Rational Field, modulo
x_2*y_1^2,
x_1*y_2^2
sage: S.reduce(p)  # needs sage.combinat
y_3*y_1
```

The next example shows that tail reduction is not done, unless it is explicitly advised:

```sage
x_3 + 2*x_2*y_1^2 + 3*x_1*y_2^2
x_3
```

However, it is possible to ask for tailreduction already when the Symmetric Reduction Strategy is created:

```sage
sage: S2
Symmetric Reduction Strategy in
Infinite polynomial ring in x, y over Rational Field, modulo
x_2*y_1^2,
x_1*y_2^2
with tailreduction
```

(continues on next page)
˓→needs sage.combinat
x_3

class sage.rings.polynomial.symmetric_reduction.SymmetricReductionStrategy

Bases: object

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:

```
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
```

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:

```
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: S = SymmetricReductionStrategy(X)
```

(continues on next page)
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:

```
sage: S.add_generator(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`:

```
sage: S.add_generator(y[2] + y[1]*x[2])
sage: S.reduce(x[3] + x[2])
```

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:

```
sage: X.<y> = InfinitePolynomialRing(QQ)
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: S = SymmetricReductionStrategy(X, [y[2]*y[1],y[1]*2*y[2]])
sage: S.gens()
[y_2*y_1^2, y_2^2*y_1]
```
reduce\( (p, \text{notail}=False, \text{report}=None) \)

Symmetric reduction of an infinite polynomial.

**INPUT:**

- \( p \) – an element of the underlying infinite polynomial ring.
- \( \text{notail} \) – (bool, default False) If True, tail reduction is avoided (but there is no guarantee that there will be no tail reduction at all).
- \( \text{report} \) – (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 \textit{tailreduce()}.

**EXAMPLES:**

```
sage: from sage.rings.polynomial.symmetric_reduction import \nSymmetricReductionStrategy
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:

```
sage: S = SymmetricReductionStrategy(X, [x[2] + y[1],
....: y[3] + y[2]])
::>
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 \nSymmetricReductionStrategy
(continues on next page)```
**sage**: `S`
Symmetric Reduction Strategy in
Infinite polynomial ring in \( y \) over Rational Field, modulo
\( y_2^*y_1^2, \)
\( y_2^*y_1 \)

**sage**: `S`.reset()
**sage**: `S`
Symmetric Reduction Strategy in Infinite polynomial ring in \( y \) over Rational Field

---

**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`.

**EXAMPLES**:

```python
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
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`.

**EXAMPLES**:

```python
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])
#→

needs sage.combinat
x_4*y_1
```

Last, we demonstrate the ‘report’ option:

```
sage: S = SymmetricReductionStrategy(X, [x[2] + y[1],
....:
....:
y[3] + y[2],
....:
....:
y[3] + y[2]])
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
needs sage.combinat
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.


CHAPTER

SEVEN

BOOLEAN POLYNOMIALS

7.1 Boolean Polynomials

Elements of the quotient ring

\[ F_2[x_1, ..., x_n]/\langle x_1^2 + x_1, ..., x_n^2 + x_n \rangle. \]

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

\[ R = F_2[a, b, c, d, e]. \]
sage: P.<a,b,c,d,e> = PolynomialRing(GF(2), 5, order='lex')
sage: I1 = ideal([a*b + c*d + 1, a*c*e + d*e, a*b*e + c*e, b*c + c*d*e + 1])
sage: for f in I1.groebner_basis():
    ....: f
    a + c^2*d + c + d^2*e
    b*c + d^3*e^2 + d^3*e + d^2*e^2 + d*e + e + 1
    b*e + d*e^2 + d*e + e
    c*e + d^3*e^2 + d^3*e + d^2*e^2 + d*e
    d^4*e^2 + d^4*e + d^3*e + d^2*e^2 + d^2*e + d*e + 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 \( F_2 \).

sage: J = I1 + sage.rings.ideal.FieldIdeal(P)
sage: for f in J.groebner_basis():
    ....: f
    a + d + 1
    b + 1
    c + 1
    d^2 + d
    e

So the solutions over \( F_2 \) are \{e = 0, d = 1, c = 1, b = 1, a = 0\} and \{e = 0, d = 0, c = 1, b = 1, a = 1\}.

We can express the restriction to \( F_2 \) by considering the quotient ring. If \( I \) is an ideal in \( F[x_1, \ldots, x_n] \) then the ideals in the quotient ring \( F[x_1, \ldots, x_n]/I \) are in one-to-one correspondence with the ideals of \( F[x_0, \ldots, x_n] \) containing \( I \) (that is, the ideals \( J \) satisfying \( I \subset J \subset P \)).

sage: Q = P.quotient( sage.rings.ideal.FieldIdeal(P) )
sage: I2 = ideal([Q(f) for f in I1.gens()])
sage: for f in I2.groebner_basis():
    ....: f
    a - b + 1
    b + 1
    c + 1
    d^2 + d
    e

This quotient ring is exactly what PolyBoRi handles well:

sage: B.<a,b,c,d,e> = BooleanPolynomialRing(5, order='lex')
sage: I2 = ideal([B(f) for f in I1.gens()])
sage: for f in I2.groebner_basis():
    ....: f
    a + d + 1
    b + 1
    c + 1
    e

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
```

```python
sage: [Variable(i, r) for i in range(r.ngens())]
[\text{x}(0), \text{x}(1), \text{y}(0), \text{y}(1), \text{y}(2)]
```

For details on this interface see: http://polybori.sourceforge.net/doc/tutorial/tutorial.html.

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:
      - \text{i} - an integer

    EXAMPLES:

    ```python
    sage: from sage.rings.polynomial.pbori.pbori import BooleConstant
    sage: [BooleConstant(i) for i in range(5)]
    [0, 1, 0, 1, 0]
    ```

    \text{deg}()

    Get degree of boolean constant.

    EXAMPLES:

    ```python
    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()
()
```
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 BooleanPolynomial can easily be converted to the type BooleSet by using the member function BooleanPolynomial.set().

INPUT:

- param - either a CCuddNavigator, a BooleSet or None.
- ring - a boolean polynomial ring.

EXAMPLES:

```python
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 *
sage: from sage.rings.polynomial.pbori.PyPolyBoRi import Monomial
sage: BooleSet([Monomial(B)])
{{}}
```

Note: BooleSet prints as {} but are not Python dictionaries.

cartesian_product(rhs)

Return the Cartesian product of this set and the set rhs.

The Cartesian product of two sets X and Y is the set of all possible ordered pairs whose first component is a member of X and whose second component is a member of Y.

\[ X \times Y = \{(x, y) | x \in X \text{ and } y \in Y\}. \]

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 = x4 + 1
sage: t = g.set(); t
{{x4}, {}}
sage: s.cartesian_product(t)
{{x1,x2,x4}, {x1,x2}, {x2,x3,x4}, {x2,x3}}
```
**change(ind)**

Swaps the presence of $x_i$ in each entry of the set.

**EXAMPLES:**

```sage
sage: P.<a,b,c> = BooleanPolynomialRing()
sage: f = a+b
sage: s = f.set(); s
{{a}, {b}}
sage: s.change(0)
{{a,b}, {}}
sage: s.change(1)
{{a,b}, {}}
sage: s.change(2)
{{a,c}, {b,c}}
```

**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:

$$X \setminus Y = \{x | x \in X \text{ and } x \not\in Y\}.$$  

**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 = x2*x3 + 1
sage: t = g.set(); t
{{x2,x3}, {}}
sage: s.diff(t)
{{x1,x2}}
```

**divide(rhs)**

Divide each element of this set by the monomial rhs and return a new set containing the result.

**EXAMPLES:**

```sage
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing(order='lex')
sage: f = b*e + b*c*d + b
sage: s = f.set(); s
{{b,c,d}, {b,e}, {b}}
sage: s.divide(b.lm())
{{c,d}, {e}, {}}
sage: f = b*e + b*c*d + b + c
sage: s = f.set()
sage: s.divide(b.lm())
{{c,d}, {e}, {}}
```

**divisors_of(m)**

Return those members which are divisors of $m$.

**INPUT:**
* m - a boolean monomial

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()
sage: s.divisors_of((x1*x2*x4).lead())
{{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*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:

\[ X \cap Y = \{ x \mid x \in X \text{ and } x \in Y \}. \]

EXAMPLES:

```python
sage: B = BooleanPolynomialRing(5, 'x')
sage: x0,x1,x2,x3,x4 = B.gens()
sage: f = x1*x2+x2*x3
sage: s = f.set(); s
{{x1, x2}, {x2, x3}}
sage: g = x2*x3 + 1
sage: t = g.set(); 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:
```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+x2^2*x3^4+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())
{{x1, x2}, {x2, x3, x4}, {x2, x4}, {x3}, {x4}, {}}

sage: nav.value()
1

sage: nav_else = nav.else_branch()
sage: BooleSet(nav_else, s.ring())
{{x2, x3, x4}, {x2, x4}, {x3}, {x4}, {}}

sage: nav_else.value()
2
```

`ring()`

Return the parent ring.

**EXAMPLES:**

```python
sage: B = BooleanPolynomialRing(5, 'x')
sage: x0, x1, x2, x3, x4 = B.gens()
sage: f = x1*x2+x2*x3
sage: f.set().ring() is B
True
```

`set()`

Return `self`.

**EXAMPLES:**

```python
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: BS = (a*b + c).set()
sage: BS.set() is BS
True
```

`size_double()`

Return the size of this set as a floating point number.

**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()
sage: s.size_double()
2.0
```
**stable_hash()**

A hash value which is stable across processes.

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:

\[ X \cup Y = \{ x \mid x \in X \text{ or } x \in Y \} \]

**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> = BooleanPolynomialRing(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
```

class **sage.rings.polynomial.pbori.pbori.BooleSetIterator**

Bases: object

Helper class to iterate over boolean sets.

class **sage.rings.polynomial.pbori.pbori.BooleanMonomial**

Bases: 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.

**deg()**

Return degree of this monomial.

```python
sage: from sage.rings.polynomial.pbori.pbori import BooleanMonomialMonoid
dsage: P.<x,y,z> = BooleanPolynomialRing(3)
dsage: M = BooleanMonomialMonoid(P)
dsage: M(x*y).deg()
2
dsage: M(x^2*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:**

```python
sage: from sage.rings.polynomial.pbori.pbori import BooleanMonomialMonoid
dsage: P.<x,y,z> = BooleanPolynomialRing(3)
dsage: M = BooleanMonomialMonoid(P)
dsage: M(x*y).degree()
2
dsage: M(x*y).degree(x)
1
dsage: M(x*y).degree(z)
0
```

**divisors()**

Return a set of boolean monomials with all divisors of this monomial.

**EXAMPLES:**

```python
sage: B.<x,y,z> = BooleanPolynomialRing(3)
sage: f = x*y
tsage: m = f.lm()
tsage: m.divisors()
{{x,y}, {x}, {y}, {}}
```

**gcd(rhs)**

Return the greatest common divisor of this boolean monomial and `rhs`.

**INPUT:**
**experiments:**

```python
sage: B.<a,b,c,d> = Boolean PolynomialRing()
sage: a,b,c,d = a.lm(), b.lm(), c.lm(), d.lm()
sage: (a*b).gcd(b*c)
b
sage: (a*b*c).gcd(d)
1
```

**index()**

Return the variable index of the first variable in this monomial.

**Examples:**

```python
sage: B.<x,y,z> = Boolean PolynomialRing(3)
sage: f = x*y
sage: m = f.lm()
sage: m.index()
0
```

**Note:** This function is part of the upstream PolyBoRi interface.

**iterindex()**

Return an iterator over the indices of the variables in self.

**Examples:**

```python
sage: from sage.rings.polynomial.pbori.pbori import BooleanMonomialMonoid
sage: P.<x,y,z> = Boolean PolynomialRing(3)
sage: M = BooleanMonomialMonoid(P)
sage: list(M(x*z).iterindex())
[0, 2]
```

**multiples(rhs)**

Return a set of boolean monomials with all multiples of this monomial up to the bound rhs.

**Input:**

- **rhs** - a boolean monomial

**Examples:**

```python
sage: B.<x,y,z> = Boolean PolynomialRing(3)
sage: f = x
sage: m = f.lm()
sage: g = x*y*z
sage: n = g.lm()
sage: m.multiples(n)
{{x,y,z}, {x,y}, {x,z}, {x}}
sage: n.multiples(m)
{{x,y,z}}
```
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: 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:

```
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:

```
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:
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:

sage: B.<x,y> = BooleanPolynomialRing()
sage: x.lm() is 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:

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()  
(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: UniqueRepresentation, 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:

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

```python
(x, y)
sage: type(M.gen(0))
<class 'sage.rings.polynomial.pbori.pbori.BooleanMonomial'>
```

Since [GitHub issue #9138](https://github.com/sagemath/sage/pull/9138), boolean monomial monoids are unique parents and are fit into the category framework:

```python
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:**

```python
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:**

```python
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:**

```python
sage: from sage.rings.polynomial.pbori.pbori import BooleanMonomialMonoid
sage: P = BooleanPolynomialRing(100, 'x')
sage: M = BooleanMonomialMonoid(P)
sage: M.ngens()
100
```
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:**

```python
sage: B.<x,y,z> = BooleanPolynomialRing(3)
sage: x.constant()
False
```

```python
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:**

```python
sage: B.<a,b> = BooleanPolynomialRing()
sage: a.constant_coefficient()
0
```

```python
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:**

```python
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: (x+y).degree()
1
```
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 $\text{None}$), 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:

- Michael Brickenstein; SlimGB: Groebner Bases with Slim Polynomials http://www.mathematik.uni-kl.de/~zca/Reports_on_ca/35/paper_35_full.ps.gz

Note: This function is part of the upstream PolyBoRi interface.

first_term()

Return the first term with respect to the lexicographical term ordering.

EXAMPLES:
sage: B.<a,b,z> = BooleanPolynomialRing(3, order='lex')
sage: f = b*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:**

```python
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:**

```python
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:**

```python
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: P(1).is_constant()
True
sage: P(0).is_constant()
True
sage: x.is_constant()
False
```

(continues on next page)
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
sage: f.is_equal((f + 1) - 1)
True
```

Note: This function is part of the upstream PolyBoRi interface.

is\_homogeneous()

Return True if this element is a homogeneous polynomial.

EXAMPLES:

```
sage: P.<x, y> = BooleanPolynomialRing()
sage: (x+y).is_homogeneous()
True
sage: P(0).is_homogeneous()
True
sage: (x+1).is_homogeneous()
False
```

is\_one()

Check if self is 1.

EXAMPLES:

```
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
```

is\_pair()

Check if self has exactly two terms.

EXAMPLES:
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
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

(continues on next page)
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()
False
sage: f = P(0)
sage: f.is_univariate()
True
```

is_zero()
Check if self is zero.

EXAMPLES:

```
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:

```
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.

**lm()**

Return the leading monomial of this boolean polynomial, with respect to the order of parent ring.

EXAMPLES:

```python
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: (x+y+y*z).lm()
x
sage: P.<x,y,z> = BooleanPolynomialRing(3, order='deglex')
sage: (x+y+y*z).lm()
y*z
sage: P(0).lm()
0
```

**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.

EXAMPLES:

```python
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: (x+y+y*z).lt()
x
sage: P.<x,y,z> = BooleanPolynomialRing(3, order='deglex')
sage: (x+y+y*z).lt()
y*z
```

**map_every_x_to_x_plus_one()**

Map every variable $x_i$ in this polynomial to $x_i + 1$.

EXAMPLES:
```python
sage: B.<a,b,z> = BooleanPolynomialRing(3)
sage: f = a*b + z + 1; f
a*b + z + 1
sage: f.map_every_x_to_x_plus_one()
a*b + a + b + z + 1
sage: f(a+1,b+1,z+1)
a*b + a + b + z + 1
```

**monomial_coefficient**(mon)

Return the coefficient of the monomial mon in self, where mon must have the same parent as self.

**INPUT:**

- mon - a monomial

**EXAMPLES:**

```python
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:**

```python
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:**
Polynomials, Release 10.1

```
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.

---

```
def 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
    sage: f.n_vars()
    3
```

**Note:** This function is part of the upstream PolyBoRi interface.

---

```
def 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: nav = f.navigation()
    sage: BooleSet(nav, B)
    {{x1, x2}, {x2, x3, x4}, {x2, x4}, {x3}, {x4}, {}}
    sage: nav.value()
    1
    sage: nav_else = nav.else_branch()
    sage: BooleSet(nav_else, B)
    {{x2, x3, x4}, {x2, x4}, {x3}, {x4}, {}}
    sage: nav_else.value()
    2
```

7.1. Boolean Polynomials 665
nvariables()

Return the number of variables used to form this boolean polynomial.

EXAMPLES:

```sage
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:

```sage
sage: B.<x0,x1,x2,x3> = BooleanPolynomialRing(4)
sage: I = B.ideal((x0 + x1 + x2 + x3,
....: x0*x1 + x0*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:
```
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:**

```
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(rhs)

Return the S-Polynomial of this boolean polynomial and the other boolean polynomial 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.

\textbf{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:

- \texttt{in\_dict} - (optional) dict with variable:value pairs
- \texttt{**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
sage: f.subs(x=y)
y*z + y + z + 1
sage: f.subs({x:1},y=1)
0
sage: f.subs(y=1)
x + 1
sage: f.subs(y=1,z=1)
x + 1
sage: f.subs(z=1)
x*y + y
sage: f.subs({'x':1},y=1)
0

This method can work fully symbolic:

sage: f.subs(x=var('a'), y=var('b'), z=var('c'))
\rightarrow \texttt{needs sage.symbolic}
a*b + b*c + c + 1
sage: f.subs({'x': var('a'), 'y': var('b'), 'z': var('c')})
\rightarrow \texttt{needs sage.symbolic}
a*b + b*c + c + 1

\textbf{terms()}

Return a list of monomials appearing in \texttt{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:**

```python
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 `is_univariate()` method. The new Polynomial is over GF(2) and in the variable x if no ring R is provided.

```python
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:

```python
sage: g = R(1)
sage: h = g.univariate_polynomial(); h
1
sage: h.parent()  # needs sage.libsntl
Univariate Polynomial Ring in x over Finite Field of size 2 (using GF2X)
```

**variable(i=0)**

Return the i-th variable occurring in self. The index i is the index in self.variables()

**EXAMPLES:**

```python
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: f = x*z + z + 1
sage: f.variables()
(x, z)
sage: f.variable(1)
z
```
variables()

Return a tuple of all variables appearing in self.

EXAMPLES:

```
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()
()```

vars_as_monomial()

Return a boolean monomial with all variables appearing in self.

EXAMPLES:

```
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: (x + y).vars_as_monomial()
x*y
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.

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:

• s - candidate points for evaluation to zero

EXAMPLES:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: f = a*b + c + d + 1
```

Now we create a set of points:
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),)
```

### class `sage.rings.polynomial.pbori.pbori.BooleanPolynomialEntry`

Bases: `object`

p

### class `sage.rings.polynomial.pbori.pbori.BooleanPolynomialIdeal`

Bases: `MPolynomialIdeal`

Construct an ideal in the boolean polynomial ring.

**INPUT:**

- `ring` - the ring this ideal is defined in
- `gens` - a list of generators
- `coerce` - coerce all elements to the ring `ring` (default: `False`)

**EXAMPLES:**

```python
sage: P.<x0, x1, x2, x3> = BooleanPolynomialRing(4)
sage: I = P.ideal(x0*x1*x2*x3 + x0*x1*x3 + x0*x1 + x0*x2 + x0)
sage: I
Ideal (x0*x1*x2*x3 + x0*x1*x3 + x0*x1 + x0*x2 + x0) of Boolean PolynomialRing in x0, x1, x2, x3
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 options affects mainly heuristics. The reducedness of the output polynomials can only be guaranteed by the option `redsb` (default: `True`)
- `minsb` - 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`)

**7.1. Boolean Polynomials**
Polynomials, Release 10.1

- `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 effect 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 should not 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:**

```
sage: P.<x0, x1, x2, x3> = BooleanPolynomialRing(4)
sage: I = P.ideal(x0*x1*x2*x3 + x0*x1*x3 + x0*x1 + x0*x2 + x0)
sage: I.groebner_basis()
[x0*x1 + x0*x2 + x0, x0*x2*x3 + x0*x3]
```

Another somewhat bigger example:

```
sage: sr = mq.SR(2,1,1,4,gf2=True, polybori=True)
sage: while True:
....:     try:
....:         F, s = sr.polynomial_system()
....:         break
....:     except ZeroDivisionError:
....:         pass
sage: I = F.ideal()
sage: I.groebner_basis()  # not tested, known bug, unstable (see :trac:`32083`)
Polynomial Sequence with 36 Polynomials in 36 Variables
```

We compute the same example with Magma:

```
sage: sr = mq.SR(2,1,1,4,gf2=True, polybori=True)
sage: while True:
....:     try:
....:         F, s = sr.polynomial_system()
....:         break
....:     except ZeroDivisionError:
....:         pass
sage: I = F.ideal()
sage: I.groebner_basis(algorithm='magma', prot='sage')  # optional - magma
```

(continues on next page)
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

interreduced_basis()

If this ideal is spanned by \((f_1, \ldots, f_n)\) this method returns \((g_1, \ldots, g_s)\) such that:

- \(<f_1, \ldots, f_n> = <g_1, \ldots, g_s>
- \text{LT}(g_i) \neq \text{LT}(g_j)\) for all \(i \neq j\)
- \(\text{LT}(g_i)\) 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: while True:
    try:
        F, s = sr.polynomial_system()
        break
    except ZeroDivisionError:
        pass
sage: I = F.ideal()
sage: g = I.interreduced_basis()
sage: len(g) == len(set(gi.lt() for gi in g))
True
sage: for i in range(len(g)):
    lt = g[i].lt()
    for j in range(len(g)):
        if i == j:
            continue
        for t in iter(g[j]):
            assert lt not in t.divisors()
```

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.

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])
0
sage: I.reduce(gb[0] + 1)
```

7.1. Boolean Polynomials
variety(**kwds)

Return the variety associated to this boolean ideal.

EXAMPLES:

A simple example:

```
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.BooleanPolynomialRing

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:

```
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 github issue #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 "pbori" 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
```
The cover ring is cached:

```
sage: B.cover_ring() is B.cover_ring()
True
```

### `defining_ideal()`

Return \( I = \langle x_i^2 + x_i \rangle \subset R \) where \( R = \text{self.cover_ring()} \), and \( x_i \) any element in the set of variables of this ring.

**EXAMPLES:**

```
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()`

Return the \( i \)-th generator of this boolean polynomial ring.

**INPUT:**

- \( i \) – an integer or a boolean monomial in one variable

**EXAMPLES:**

```
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:**

```
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:**

```
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: B.get_base_order_code()
```
Polynomials, Release 10.1

(continued from previous page)

\[
\begin{align*}
& 0 \\
&sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing(order='deglex') \\
&sage: B.get_base_order_code() \\
& \hspace{1em} 1 \\
&sage: T = TermOrder('deglex',2) + TermOrder('deglex',2) \\
&sage: B.<a,b,c,d> = BooleanPolynomialRing(4, order=T) \\
&sage: B.get_base_order_code() \\
& \hspace{1em} 1 \\
\end{align*}
\]

*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:**

\[
\begin{align*}
&sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing() \\
&sage: B.get_order_code() \\
& \hspace{1em} 0 \\
&sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing(order='deglex') \\
&sage: B.get_order_code() \\
& \hspace{1em} 1 \\
\end{align*}
\]

*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:**

\[
\begin{align*}
&sage: P.<x,y> = BooleanPolynomialRing(2) \\
&sage: P.has_degree_order() \\
& \hspace{1em} False \\
\end{align*}
\]

### id()

Return a unique identifier for this boolean polynomial ring.

**EXAMPLES:**

\[
\begin{align*}
&sage: P.<x,y> = BooleanPolynomialRing(2) \\
&sage: print("id: {}".format(P.id())) \\
& \hspace{1em} id: ... \\
&sage: P = BooleanPolynomialRing(10, 'x') \\
&sage: Q = BooleanPolynomialRing(20, 'x') \\
&sage: P.id() != Q.id() \\
& \hspace{1em} True \\
\end{align*}
\]
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:

```
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: P.ideal(x+y)
Ideal (x + y) of Boolean PolynomialRing in x, y, z
```

```
sage: P.ideal(x*y, y*z)
Ideal (x*y, y*z) of Boolean PolynomialRing in x, y, z
```

```
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.

```
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.

```
sage: zeros = set([(1, 0, 1, 0, 0, 0), (1, 0, 0, 1, 1, 1), (0, 0, 0, 1, 1, 1), (1, 1, 0, 0, 1, 1)])
sage: ones = set([(0, 0, 0, 0, 0, 0), (1, 0, 1, 0, 1, 0), (0, 0, 0, 1, 1, 1), (0, 1, 1, 1, 1, 1), (1, 1, 1, 0, 1, 0)])
```

```
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.

```
sage: g = B.interpolation_polynomial(zeros, ones); g
b*f + c + d*f + d + e*f + e + 1
```
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: f = P.random_element(degree=3, terms=4)
sage: f.degree() <= 3
True
sage: len(f.terms())
4

sage: f = P.random_element(degree=1, terms=2)
sage: f.degree() <= 1
True
sage: len(f.terms())
2

In corner cases this function will return fewer terms by default:
sage: P = BooleanPolynomialRing(2, 'y')
sage: f = P.random_element()
sage: len(f.terms())
2
sage: P = BooleanPolynomialRing(1, 'y')
sage: f = P.random_element()
sage: len(f.terms())
1

We return uniformly random polynomials up to degree 2:
sage: from collections import defaultdict
sage: B.<a,b,c,d> = BooleanPolynomialRing()
sage: counter = 0.0
sage: dic = defaultdict(Integer)
sage: def more_terms():
.....:     global counter, dic
.....:     for t in B.random_element(terms=Infinity).terms():
    (continues on next page)
....:     counter += 1.0
....:     dic[t] += 1

\texttt{sage: more_terms()}
\texttt{sage: while any(abs(dic[t]/counter - 1.0/11) > 0.01 for t in dic):
....:     more_terms()}

\texttt{\textbf{remove_var}(\textit{order=None, *var})}

Remove a variable or sequence of variables from this ring.

If \textit{order} is not specified, then the subring inherits the term order of the original ring, if possible.

\textbf{EXAMPLES:}

\begin{verbatim}
\texttt{sage: R.<x,y,z,w> = BooleanPolynomialRing()}
\texttt{sage: R.remove_var(z)}
Boolean PolynomialRing in x, y, w
\texttt{sage: R.remove_var(z,x)}
Boolean PolynomialRing in y, w
\texttt{sage: R.remove_var(y,z,x)}
Boolean PolynomialRing in w
\end{verbatim}

Removing all variables results in the base ring:

\begin{verbatim}
\texttt{sage: R.remove_var(y,z,x,w)}
Finite Field of size 2
\end{verbatim}

If possible, the term order is kept:

\begin{verbatim}
\texttt{sage: R.<x,y,z,w> = BooleanPolynomialRing(order='deglex')}
\texttt{sage: R.remove_var(y).term_order()}
Degree lexicographic term order
\texttt{sage: R.<x,y,z,w> = BooleanPolynomialRing(order='lex')}
\texttt{sage: R.remove_var(y).term_order()}
Lexicographic term order
\end{verbatim}

Be careful with block orders when removing variables:

\begin{verbatim}
\texttt{sage: R.<x,y,z,u,v> = BooleanPolynomialRing(order='deglex(2),deglex(3)')}
\texttt{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
\texttt{sage: R.remove_var(x,y,z, order='deglex')} 
Boolean PolynomialRing in u, v
\end{verbatim}

\texttt{\textbf{variable}(\textit{i=0})}

Return the \textit{i}-th generator of this boolean polynomial ring.

\textbf{INPUT:}

\begin{itemize}
\item \textit{i} - an integer or a boolean monomial in one variable
\end{itemize}

\textbf{EXAMPLES:}
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: P.variable()
x
sage: P.variable(2)
z
sage: m = x.monomials()[0]
sage: P.variable(m)
x

zero()

EXAMPLES:

sage: P.<x0,x1> = BooleanPolynomialRing(2)
sage: P.zero()
0

class sage.rings.polynomial.pbori.pbori.BooleanPolynomialVector

A vector of boolean polynomials.

EXAMPLES:

```python
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: from sage.rings.polynomial.pbori.pbori import BooleanPolynomialVector
sage: l = [B.random_element() for _ in range(3)]
sage: v = BooleanPolynomialVector(l)
sage: len(v)
3
sage: all(vi.parent() is B for vi in v)
True
```

append(el)

Append the element el to this vector.

EXAMPLES:

```python
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: from sage.rings.polynomial.pbori.pbori import BooleanPolynomialVector
sage: v = BooleanPolynomialVector()
sage: entries = []
sage: for i in range(5):
...:     entries.append(B.random_element())
...:     v.append(entries[-1])

sage: list(v) == entries
True
```

class sage.rings.polynomial.pbori.pbori.BooleanPolynomialVectorIterator

Bases: object

class sage.rings.polynomial.pbori.pbori.CCuddNavigator

Bases: object

constant()
class sage.rings.polynomial.pbori.pbori.FGLMStrategy

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.

EXAMPLES:

```
sage: from sage.rings.polynomial.pbori.pbori import *
sage: B.<x,y,z> = BooleanPolynomialRing()
sage: ideal = BooleanPolynomialVector([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
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):
... ValueError: strategy already contains a polynomial with same lead
```

add\_generator\_delayed($p$)
Add a new generator but do not perform interreduction immediately.

INPUT:

- $p$ - a polynomial

EXAMPLES:

```sage
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\_delayed(a + c)
sage: list(gbs)
[a + b]
sage: list(gbs.all\_generators())
[a + b, a + c]
```

all\_generators()

EXAMPLES:

```sage
sage: from sage.rings.polynomial.pbori.pbori import *
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing()
sage: gbs = GroebnerStrategy(B)
```

(continues on next page)
Polynomials, Release 10.1

(continued from previous page)

```
sage: gbs.add_as_you_wish(a + b)
sage: list(gbs)
[a + b]
sage: gbs.add_as_you_wish(a + c)
sage: list(gbs)
[a + b]
sage: list(gbs.all_generators())
[a + b, a + c]
```

**all_spols_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()
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: 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)
```

(continues on next page)
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.

Note: Use this function if strat contains a GB.

minimalize_and_tail_reduce()
Return a vector of all polynomials with minimal leading terms and do tail reductions.

Note: Use that if strat contains a GB and you want a reduced GB.

next_spoly()

nf(p)
Compute the normal form of p with respect to the generating set.

INPUT:
• p - a boolean polynomial

EXAMPLES:

sage: P = PolynomialRing(GF(2),10, 'x')
sage: B = BooleanPolynomialRing(10,'x')
sage: I = P.ideal([f for f in I.gens()])
sage: gb = I.groebner_basis()
sage: from sage.rings.polynomial.pbori.pbori import GroebnerStrategy
sage: G = GroebnerStrategy(B)
Polynomials, Release 10.1

(continued from previous page)

```python
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.

**n pairs**

**reduction_strategy**

**select(m)**

Return the index of the generator which can reduce the monomial \( m \).

**INPUT:**

- \( m \) - a *BooleanMonomial*

**EXAMPLES:**

```python
sage: B.<a,b,c,d,e> = BooleanPolynomialRing()
sage: f = B.random_element()
sage: g = B.random_element()
sage: while g.lt() == f.lt():
    ....:
    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())
0
sage: strat.select(g.lm())
1
sage: strat.select(e.lm())
-1
```

**small_s pol y s_in_next_degree(f, n)**

**some_s pol y s_in_next_degree(n)**

**suggest_plugin_variable()**

**sym mGB_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.
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
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 + f)
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
sage: from sage.rings.polynomial.pbori.pbori import groebner_basis
sage: g = groebner_basis(gb)
sage: list(g)
[a, b + 1, c + 1, d, e + 1, f]
sage: gb = GroebnerStrategy(B)
sage: _ = [gb.add_generator(f) for f in g]
sage: gb.variable_has_value(0)
True
```

class sage.rings.polynomial.pbori.pbori.MonomialConstruct

Bases: object

Implements PolyBoRi’s Monomial() constructor.


class sage.rings.polynomial.pbori.pbori.MonomialFactory

Bases: object

Implements PolyBoRi’s Monomial() constructor. If a ring is given is can be used as a Monomial factory for the given ring.

EXAMPLES:

```sage
sage: from sage.rings.polynomial.pbori.pbori import *
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: fac = MonomialFactory()
sage: fac = MonomialFactory(B)
```

class sage.rings.polynomial.pbori.pbori.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:

```python
sage: from sage.rings.polynomial.pbori.pbori import *
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: PolynomialConstruct().lead(a)
a
```

-----

**class** sage.rings.polynomial.pbori.pbori.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:

```python
sage: from sage.rings.polynomial.pbori.pbori import *
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: PolynomialFactory().lead(a)
a
```

-----

**class** sage.rings.polynomial.pbori.pbori.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: [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
```

(continues on next page)
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:

```sage
from sage.rings.polynomial.pbori.pbori import *
B.<a,b,c,d> = BooleanPolynomialRing()
red = ReductionStrategy(B)
red.add_generator(a*b + c + 1)
red.add_generator(b*c + d + 1)
red.add_generator(a)
red.cheap_reductions(a*b + a)
\emptyset
red.cheap_reductions(b + c)
b + c
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:

```sage
from sage.rings.polynomial.pbori.pbori import *
B.<x,y,z> = BooleanPolynomialRing()
red = ReductionStrategy(B)
red.opt_red_tail = True
red.add_generator(x + y + 1)
red.add_generator(y*z + z)
red.head_normal_form(x + y*z)
y + z + 1
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:
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
```

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.

class sage.rings.polynomial.pbori.pbori.VariableFactory

Bases: object

    Implements PolyBoRi’s `Variable()` constructor and a variable factory for given ring

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()
```
Polynomials, Release 10.1

sage: l = [B.random_element() for _ in range(5)]
sage: _ = [v.append(e) for e in l]
sage: add_up_polynomials(v, B.zero()) == sum(l)
True

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:

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: e = gauss_on_polys(l)
sage: E, v = Sequence(e, B).coefficient_matrix()
sage: E == A.echelon_form()
True

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:

sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: R.<z,x> = QQ[]
sage: sage.rings.polynomial.pbori.pbori.get_var_mapping(P,R)
[z, x]
sage: sage.rings.polynomial.pbori.pbori.get_var_mapping(P, x^2)
[None, x]

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

7.1. Boolean Polynomials
• 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)
{{x1,x2,x3}, {x1,x3}, {x4}}
```

```python
sage: if_then_else(x1.lm().index(),f0,f1)
{{x1,x2,x3}, {x1,x3}, {x4}}
```

```python
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.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
sage: from sage.rings.polynomial.pbori.interpolate import *
sage: V=(x(0)+x(1)+x(2)+x(3)+1).set()
```

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 \( \mathbb{F}_2^4 \) or a vector space of higher dimension:

```python
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
```

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 `interpolate_smallest_lex`, which has two arguments: the set of interpolation points mapped to zero and the set of interpolation points mapped to one:

```python
sage: z=f.zeros_in(V)
sage: z
{{{x1}, {x2}}}
sage: o=V.diff(z)
sage: o
{{{x0}, {x3}, {}}}
sage: interpolate_smallest_lex(z,o)
x1 + x2 + 1
```

**sage.rings.polynomial.pbori.pbori.ll_red_nf_noredsb**(p, reductors)

Reduce the polynomial \( p \) by the set of reductors with linear leading terms.

**INPUT:**

- \( p \) - a boolean polynomial
- reductors - a boolean set encoding a Groebner basis with linear leading terms.

**EXAMPLES:**
sage: from sage.rings.polynomial.pbori.pbori import ll_red_nf_noredsb
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() )
sage: ll_red_nf_noredsb(p, reductors)
b*c + b*d + c + d + 1

sage: B.<a,b,c,d> = BooleanPolynomialRing()

sage: f,g = a + c + 1, b + d + 1
sage: reductors = f.set().union( g.set() )

sage: ll_red_nf_noredsb_single_recursive_call(p, reductors)
b*c + b*d + c + d + 1

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)
Redude the polynomial p by the set of reductors with linear leading terms. It is assumed that the set reductors is a reduced Groebner basis.

INPUT:
• p - a boolean polynomial
• reductors - a boolean set encoding a reduced Groebner basis with linear leading terms.

EXAMPLES:

sage: from sage.rings.polynomial.pbori.pbori import ll_red_nf_redsb
sage: B.<a,b,c,d> = BooleanPolynomialRing()

sage: f,g = a + c + 1, b + d + 1
sage: reductors = f.set().union( g.set() )

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)
Map every variable x_i in this polynomial to x_i + 1.

EXAMPLES:
```
sage: B.<a,b,z> = BooleanPolynomialRing(3)
sage: f = a*b + z + 1; f
a*b + z + 1
sage: from sage.rings.polynomial.pbori.pbori import map_every_x_to_x_plus_one
sage: map_every_x_to_x_plus_one(f)
a*b + a + b + z + 1
sage: f(a+1,b+1,z+1)
a*b + a + b + z + 1
```

```sage
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)
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.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}, {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.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
```

---

**7.1. Boolean Polynomials**

697
sage.rings.polynomial.pbori.pbori.set_random_seed(seed)

Set the PolyBoRi random seed to seed.

EXAMPLES:

```sage
from sage.rings.polynomial.pbori.pbori import random_set, set_random_seed
B.<a,b,c,d,e> = BooleanPolynomialRing()
(a*b*c*d).lm()
a*b*c*d
set_random_seed(1337)
random_set((a*b*c*d).lm(),2)
random_set((a*b*c*d).lm(),2)
set_random_seed(1337)
random_set((a*b*c*d).lm(),2)
random_set((a*b*c*d).lm(),2)
```

sage.rings.polynomial.pbori.pbori.substitute_variables(parent, vec, poly)

var(i) is replaced by vec[i] in poly.

EXAMPLES:

```sage
B.<a,b,c> = BooleanPolynomialRing()
f = a*b + c + 1
from sage.rings.polynomial.pbori.pbori import substitute_variables
substitute_variables(B, [a,b,c],f)
a*b + c + 1
substitute_variables(B, [a+1,b,c],f)
a*b + b + c + 1
substitute_variables(B, [a+1,b+1,c],f)
a*b + a + b + c
substitute_variables(B, [a+1,b+1,B(0)],f)
a*b + a + b
```

Substitution is also allowed with different rings:

```sage
B.<a,b,c> = BooleanPolynomialRing()
f = a*b + c + 1
B.<w,x,y,z> = BooleanPolynomialRing(order='deglex')
from sage.rings.polynomial.pbori.pbori import substitute_variables
substitute_variables(B, [x,y,z], f) * w
w*x*y + w*z + w
```

sage.rings.polynomial.pbori.pbori.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
```

**sage.rings.polynomial.pbori.pbori.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
```

**sage.rings.polynomial.pbori.pbori.unpickle_BooleanPolynomial0** *(ring, l)*

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
```

**sage.rings.polynomial.pbori.pbori.unpickle_BooleanPolynomialRing** *(n, names, order)*

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
```

**sage.rings.polynomial.pbori.pbori.zeros** *(pol, s)*

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:
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.

sage: from sage.rings.polynomial.pbori.pbori import zeros
sage: zeros(f, s)
{{a,b}}

For comparison we work with tuples:

sage: f.zeros_in([(1,1,1,0), (1,1,0,0), (0,0,1,1), (0,1,1,0)])
((1, 1, 0, 0),)
• Index
• Module Index
• Search Page
sage.rings.fraction_field, 555
sage.rings.fraction_field_element, 561
sage.rings.fraction_field_FpT, 566
sage.rings.invariants.invariant_theory, 492
sage.rings.invariants.reconstruction, 537
sage.rings.monomials, 492
sage.rings.polynomial.complex_roots, 235
sage.rings.polynomial.convolution, 276
sage.rings.polynomial.cyclotomic, 277
sage.rings.polynomial.flatten, 489
sage.rings.polynomial.hilbert, 488
sage.rings.polynomial.ideal, 238
sage.rings.polynomial.infinite_polynomial_element, 611
sage.rings.polynomial.infinite_polynomial_ring, 601
sage.rings.polynomial.integer_valued_polynomials, 267
sage.rings.polynomial.laurent_polynomial, 584
sage.rings.polynomial.laurent_polynomial_ring, 579
sage.rings.polynomial.laurent_polynomial_ring_base, 575
sage.rings.polynomial.msolve, 470
sage.rings.polynomial.multi_polynomial, 316
sage.rings.polynomial.multi_polynomial_element, 346
sage.rings.polynomial.multi_polynomial_ideal, 366
sage.rings.polynomial.multi_polynomial_ideal_libsingular, 468
sage.rings.polynomial.multi_polynomial_libsingular, 438
sage.rings.polynomial.multi_polynomial_ring, 342
sage.rings.polynomial.multi_polynomial_ring_base, 303
sage.rings.polynomial.multi_polynomial_sequence, 421
sage.rings.polynomial.omega, 594
sage.rings.polynomial.padics.polynomial_padic, 190
sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense, 194
sage.rings.polynomial.padics.polynomial_padic_flat, 200
sage.rings.polynomial.pbori.pbori, 639
sage.rings.polynomial.polydict, 471
sage.rings.polynomial.polynomialCompiled, 266
sage.rings.polynomial.polynomial_element, 33
sage.rings.polynomial.polynomial_element_generic, 124
sage.rings.polynomial.polynomial_fateman, 267
sage.rings.polynomial.polynomial_gf2x, 135
sage.rings.polynomial.polynomial_integer_dense_flint, 142
sage.rings.polynomial.polynomial_integer_dense_ntl, 151
sage.rings.polynomial.polynomial_modn_dense_ntl, 176
sage.rings.polynomial.polynomial_number_field, 139
sage.rings.polynomial.polynomial_quotient_ring, 239
sage.rings.polynomial.polynomial_quotient_ring_element, 261
sage.rings.polynomial.polynomial_rational_flint, 156
sage.rings.polynomial.polynomial_real_mpfr_dense, 187
sage.rings.polynomial.polynomial_ring, 9
sage.rings.polynomial.polynomial_ring_constructor, 1
sage.rings.polynomial.polynomial_ring_homomorphism, 32
sage.rings.polynomial.polynomial_singular_interface, 190
sage.rings.polynomial.polynomial_zmod_flint, 168
sage.rings.polynomial.polynomial_zz_pex, 201
sage.rings.polynomial.real_roots, 205
sage.rings.polynomial.refine_root, 237

703
sage.rings.polynomial.symmetric_ideal, 621
sage.rings.polynomial.symmetric_reduction, 631
sage.rings.polynomial.term_order, 281
sage.rings.polynomial.toy_buchberger, 541
sage.rings.polynomial.toy_d_basis, 550
sage.rings.polynomial.toy_variety, 546
Symbols

_add_ () (sage.rings.polynomial.polynomial_element.Polynomial method), 34

_add_ () (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 142

_add_ () (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_modn_ntl_zz method), 181

_add_ () (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 156

_add_ () (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 171

_lmul_ () (sage.rings.polynomial.polynomial_element.Polynomial method), 35

_lmul_ () (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 142

_lmul_ () (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_modn_ntl_zz method), 181

_lmul_ () (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 156

_lmul_ () (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 171

_mul_ () (sage.rings.polynomial.polynomial_element.Polynomial method), 35

_mul_ () (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 143

_mul_ () (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_modn_ntl_zz method), 182

_mul_ () (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 157

_mul_ () (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 172

_mul_trunc_ () (sage.rings.polynomial.polynomial_element.Polynomial method), 36

_mul_trunc_ () (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 143

_mul_trunc_ () (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_modn_ntl_zz method), 182

_mul_trunc_ () (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 157

_mul_trunc_ () (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 172

_rmul_ () (sage.rings.polynomial.polynomial_element.Polynomial method), 35

_rmul_ () (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint method), 142

_rmul_ () (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_modn_ntl_zz method), 182

_rmul_ () (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint method), 157

_rmul_ () (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 172

_A_invariant() (sage.rings.invariants.invariant_theory.BinaryQuintic method), 500

_a_realization() (sage.rings.polynomial.integer_valued_polynomials.IntegerValuedPolynomialRing method), 276

abc_pd (classinsage.rings.polynomial.polynomial_compiled), 266

add_as_you_wish() (sage.rings.pbori.pbori.GroebnerStrategy method), 684

add_bigoh() (sage.rings.pbori.pbori.GroebnerStrategy method), 690

add_bigoh() (sage.rings.pbori.pbori.ReductionStrategy method), 633

add_generator() (sage.rings.pbori.pbori.GroebnerStrategy method), 685

add_generator() (sage.rings.pbori.pbori.ReductionStrategy method), 690

add_generator() (sage.rings.pbori.pbori.SymmetricReductionStrategy method), 633

add_generator_delayed() (sage.rings.pbori.pbori.GroebnerStrategy method), 685
add_m_mul_q() (sage.rings.polynomial.multi_polynomial.MPolynomial_libsingular method), 444
add_pd(class in sage.rings.polynomial.polynomial_compiled), 266
add_up_polynomials() (in module sage.rings.polynomial.pbori.pbori), 692
algebra_generators() (sage.rings.polynomial.integer_values.MPolynomialRing_libsingular method), 269
algebraic_dependence() (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic method), 426
AlgebraicForm (class in sage.rings.invariants.invariant_theory), 494
all_done() (sage.rings.polynomial.real_roots.ocean), 224
all_generators() (sage.rings.polynomial.pbori.pbori.GroebnerStrategy method), 685
all_roots_in_interval() (sage.rings.polynomial.polynomial_element.Polynomial method), 37
all_spolys_in_next_degree() (sage.rings.polynomial.pbori.pbori.GroebnerStrategy method), 686
alpha_covariant() (sage.rings.invariants.invariant_theory.BinaryQuintic method), 502
ambient() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic method), 249
any_root() (sage.rings.polynomial.polynomial_element.Polynomial method), 37
append() (sage.rings.polynomial.pbori.pbori.BooleanPolynomialVector method), 206
apply_map() (sage.rings.polynomial.polydict.PolyDict method), 478
approx_bp() (sage.rings.polynomial.real_roots.ocean.method), 224
args() (sage.rings.polynomial.multi_polynomial.MPolynomial method), 316
args() (sage.rings.polynomial.multi_polynomial_element.Polynomial method), 38
arithmetic_invariants() (sage.rings.invariants.invariant_theory.BinaryQuintic method), 503
as_float() (sage.rings.polynomial.real_roots.interval_Flint method), 215
as_float() (sage.rings.polynomial.real_roots.interval_Flint method), 217
as_QuadraticForm() (sage.rings.invariants.invariant_theory.QuadraticForm method), 519
associated_primes() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr method), 388
associativity() (sage.rings.polynomial.real_roots.interval_BernsteinPolynomial method), 376
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Module/Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>change_variable_name()</td>
<td>(sage.rings.polynomial.polynomial_element.Polynomial.method, 39)</td>
</tr>
<tr>
<td>characteristic()</td>
<td>(sage.rings.fraction_field.FractionField_generic.method, 317)</td>
</tr>
<tr>
<td>characteristic()</td>
<td>(sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing_sparse.method, 607)</td>
</tr>
<tr>
<td>characteristic()</td>
<td>(sage.rings.polynomial.laurent_polynomial.PolynomialRing_generic.method, 576)</td>
</tr>
<tr>
<td>characteristic()</td>
<td>(sage.rings.polynomial.multi_polynomial_ring_generic.PolynomialRing_base.method, 304)</td>
</tr>
<tr>
<td>characteristic()</td>
<td>(sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_element, 250)</td>
</tr>
<tr>
<td>characteristic()</td>
<td>(sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRing_element, 250)</td>
</tr>
<tr>
<td>cheap_reductions()</td>
<td>(sage.rings.polynomial.pbori.pbori.ReductionStrategy, 519)</td>
</tr>
<tr>
<td>class_group()</td>
<td>(sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRing_element, 525)</td>
</tr>
<tr>
<td>class_number()</td>
<td>(sage.rings.fraction_field.FractionField_Lpoly_field.method, 208)</td>
</tr>
<tr>
<td>clean_top_by_chain_criterion()</td>
<td>(sage.rings.polynomial.polydict.PolyDict.method, 478)</td>
</tr>
<tr>
<td>clebsch_invariants()</td>
<td>(sage.rings.invariants.invariant_theory.BinaryQuintic.method, 527)</td>
</tr>
<tr>
<td>clone()</td>
<td>(sage.rings.polynomial.pbori.BooleanPolynomialRing.sage.rings.polynomial.polydict.ETuple.method, 547)</td>
</tr>
<tr>
<td>coeffpd</td>
<td>(class in sage.rings.polynomial.polynomial_compiled.sage.rings.polynomial.polynomial_element_infinite_polynomialCompiledPolynomialFunction, 267)</td>
</tr>
<tr>
<td>coefficient()</td>
<td>(sage.rings.polynomial.infinite_polynomial_element.InfinitePolynomial.method, 613)</td>
</tr>
<tr>
<td>coefficient()</td>
<td>(sage.rings.polynomial.laurent_polynomial.LaurentPolynomial.method, 586)</td>
</tr>
<tr>
<td>coefficient()</td>
<td>(sage.rings.polynomial.multi_polynomial_element.MPolynomialElement.method, 348)</td>
</tr>
<tr>
<td>coefficient()</td>
<td>(sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular.method, 445)</td>
</tr>
<tr>
<td>coefficient()</td>
<td>(sage.rings.polynomial.polydict.PolyDict.method, 576)</td>
</tr>
<tr>
<td>coefficient_matrix()</td>
<td>(sage.rings.polynomial.toy_variety.PolynomialRing_generic.method, 22)</td>
</tr>
<tr>
<td>coefficient_matrix()</td>
<td>(sage.rings.polynomial.multi_polynomial_sequence.PolynomialQuotientRing_generic.method, 427)</td>
</tr>
<tr>
<td>coefficients()</td>
<td>(sage.rings.invariants.invariant_theory.AtomicPolynomialCoefficients, 495)</td>
</tr>
<tr>
<td>coefficients()</td>
<td>(sage.rings.polynomial.complex_roots, 235)</td>
</tr>
</tbody>
</table>
complex_roots() (sage.rings.polynomial.polynomial_element.PolynomialElement_class, 40)
compose_power() (sage.rings.polynomial.polynomial_element.PolynomialElement_class, 40)
contains() (sage.rings.polynomial.polybori.polybori.GroebnerStrategy, 693)
contains() (sage.rings.polynomial.polybori.polybori.BooleanPolynomial, 693)
composed_op() (sage.rings.polynomial.polynomial_element.PolynomialElement_class, 41)
composition() (sage.rings.polynomial.polybori.polybori.GroebnerStrategy, 686)
connected_components() (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic, 428)
connection_graph() (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic, 428)
connection_graph() (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic, 428)
constant() (sage.rings.polynomial.polybori.polybori.BooleanPolynomialRing, 143)
constant() (sage.rings.polynomial.polybori.polybori.BooleanPolynomial, 143)
constant() (sage.rings.polynomial.polybori.polybori.BooleanPolynomial, 143)
constant() (sage.rings.polynomial.polybori.polybori.BooleanPolynomial, 143)
constant() (sage.rings.polynomial.polybori.polybori.BooleanPolynomial, 143)
covariant_conic() (sage.rings.invariants.invariant_theory.TernaryQuadratic, 446)
covariant_conic() (sage.rings.invariants.invariant_theory.TernaryQuadratic, 446)
create_object() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRingFactory, 75)
create_object() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRingFactory, 75)
create_object() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRingFactory, 75)
create_object() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRingFactory, 75)
create_object() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRingFactory, 75)
cyclotomic_coeffs() (in module sage.rings.polynomial.cyclotomic_coeffs, 277)
cyclotomic_coeffs() (in module sage.rings.polynomial.cyclotomic_coeffs, 277)
cyclotomic_coeffs() (in module sage.rings.polynomial.cyclotomic_coeffs, 277)
cyclotomic_coeffs() (in module sage.rings.polynomial.cyclotomic_coeffs, 277)
cyclotomic_coeffs() (in module sage.rings.polynomial.cyclotomic_coeffs, 277)
cyclotomic_coefficients() (in module sage.rings.polynomial.cyclotomic_coefficients, 22)
cyclotomic_coefficients() (in module sage.rings.polynomial.cyclotomic_coefficients, 22)
cyclotomic_coefficients() (in module sage.rings.polynomial.cyclotomic_coefficients, 22)
cyclotomic_coefficients() (in module sage.rings.polynomial.cyclotomic_coefficients, 22)
cyclotomic_coefficients() (in module sage.rings.polynomial.cyclotomic_coefficients, 22)
create() (sage.rings.polynomial.polybori.polybori.GroebnerStrategy, 275)
create() (sage.rings.polynomial.polybori.polybori.GroebnerStrategy, 275)
create() (sage.rings.polynomial.polybori.polybori.GroebnerStrategy, 275)
create() (sage.rings.polynomial.polybori.polybori.GroebnerStrategy, 275)
create() (sage.rings.polynomial.polybori.polybori.GroebnerStrategy, 275)
canonical_ring() (sage.rings.polynomial.polybori.polybori.BooleanPolynomialRing, 676)
canonical_ring() (sage.rings.polynomial.polybori.polybori.BooleanPolynomialRing, 676)
canonical_objects() (sage.rings.polynomial.infinity_polynomial_ring.InfinityPolynomialRing_dense, 606)
canonical_objects() (sage.rings.polynomial.infinity_polynomial_ring.InfinityPolynomialRing_dense, 606)
canonical_objects() (sage.rings.polynomial.infinity_polynomial_ring.InfinityPolynomialRing_dense, 606)
canonical_objects() (sage.rings.polynomial.infinity_polynomial_ring.InfinityPolynomialRing_dense, 606)
canonical_objects() (sage.rings.polynomial.infinity_polynomial_ring.InfinityPolynomialRing_dense, 606)
d_basis() (in module sage.rings.polynomial.toy_d_basis, 552)
d_basis() (in module sage.rings.polynomial.toy_d_basis, 552)
d_basis() (in module sage.rings.polynomial.toy_d_basis, 552)
d_basis() (in module sage.rings.polynomial.toy_d_basis, 552)
d_basis() (in module sage.rings.polynomial.toy_d_basis, 552)
degree() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial_integer, 209)
degree() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial_integer, 209)
degree() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial_integer, 209)
degree() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial_integer, 209)
degree() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial_integer, 209)
degree() (sage.rings.polynomial.convolution, 276)
degree() (sage.rings.polynomial.convolution, 276)
degree() (sage.rings.polynomial.convolution, 276)
degree() (sage.rings.polynomial.convolution, 276)
degree() (sage.rings.polynomial.convolution, 276)
degree() (sage.rings.polynomial.convolution, 276)
degree() (sage.rings.polynomial.convolution, 276)
degree() (sage.rings.polynomial.convolution, 276)
degree() (sage.rings.polynomial.convolution, 276)
degree() (sage.rings.polynomial.convolution, 276)
groebner_basis() (sage.rings.polynomial.term_order.TermOrder method), 288
greater_tuple_degrevlex() (sage.rings.polynomial.term_order.TermOrder method), 288
greater_tuple_invlex() (sage.rings.polynomial.term_order.TermOrder method), 289
greater_tuple_lex() (sage.rings.polynomial.term_order.TermOrder method), 289
greater_tuple_matrix() (sage.rings.polynomial.term_order.TermOrder method), 289
greater_tuple_negdegrevlex() (sage.rings.polynomial.term_order.TermOrder method), 290
greater_tuple_negdeglex() (sage.rings.polynomial.term_order.TermOrder method), 290
greater_tuple_neglex() (sage.rings.polynomial.term_order.TermOrder method), 290
greater_tuple_negwdegrevlex() (sage.rings.polynomial.term_order.TermOrder method), 291
greater_tuple_negwdeglex() (sage.rings.polynomial.term_order.TermOrder method), 291
groebner_basis() (sage.rings.polynomial.ideal.Ideal_1poly_field method), 324
GroebnerStrategy (class in sage.rings.polynomial.pbori.pbori, 684
H
h_covariant() (sage.rings.invariants.invariant_theory.BinaryQuartic method), 498
H_covariant() (sage.rings.invariants.invariant_theory.BinaryQuintic method), 501
h_polynomial() (sage.rings.polynomial.integer_valued_polynomials.Integer method), 274
h_vector() (sage.rings.polynomial.integer_valued_polynomials.Integer method), 274
hamming_weight() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial method), 585
hamming_weight() (sage.rings.polynomial.multi_polynomial_element.MPolynomial method), 347
hamming_weight() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_singular_repr method), 453
has_constant_part() (sage.rings.polynomial.pbori.pbori.BooleanPolynomial method), 657
has_constant_part() (sage.rings.polynomial.pbori.pbori.BooleanPolynomial method), 641
has_cyclotomic_factor() (sage.rings.polynomial.polynomial_element.Polynomial method), 58
has_degree_order() (sage.rings.polynomial.pbori.pbori.BooleanPolynomial method), 678
has_root() (sage.rings.polynomial.real_roots.island method), 221
head_normal_form() (sage.rings.polynomial.pbori.pbori.ReductionStrategy method), 691
hensel_lift() (sage.rings.polynomial.polynomial_element_generic.Polynomial method), 126
hensel_lift() (sage.rings.polynomial.polynomial_rational_flint.Polynomial method), 163
Hessian() (sage.rings.invariants.invariant_theory.TernaryCubic method), 324
hilbert_numerator() (sage.rings.polynomial.multi_polynomial.ideal.MPolynomialIdeal method), 397
hilbert_polynomial() (in module sage.rings.polynomial.hilbert), 488
hilbert_polynomial() (sage.rings.polynomial.multi_polynomial.ideal.MPolynomialIdeal method), 397
hilbert_polynomial() (sage.rings.polynomial.multi_polynomial.ideal.MPolynomialIdeal method), 397
hilbert_series() (sage.rings.polynomial.multi_polynomial.ideal.MPolynomialIdeal method), 322
homogeneous_components() (sage.rings.polynomial.multi_polynomial.ideal.MPolynomialIdeal method), 322
homogeneous_symmetric_function() (in module sage.rings.polynomial.symmetric_ideal), 598
homogenize() (sage.rings.polynomial.multi_polynomial.MPolynomial method)
Homogenize() (sage.rings.polynomial.multi_polynomial_element.Polynomial, 379)

Homogenize() (sage.rings.polynomial.polydict.PolyDict, 480)

Homogenize() (sage.rings.polynomial.polynomial_element.Polynomial, 58)

Homogenized() (sage.rings.invariants.invariant_theory.AlgebraicForm, 495)

Homogenized() (sage.rings.invariants.invariant_theory.SeveralAlgebraicForms, 523)

InfinitePolynomialRing dense (class in sage.rings.polynomial.infinite_polynomial_ring), 605

InfinitePolynomialRing sparse (class in sage.rings.polynomial.infinite_polynomial_ring), 606

InfinitePolynomialRingFactory (class in sage.rings.polynomial.infinite_polynomial_ring), 607

inhomogeneous_quadratic_form() (sage.rings.invariants.invariant_theory.InvariantTheoryFactory, 515)

int list() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial, 177)

int list() (sage.rings.polynomial.polynomial_modn_dense_ntl.PolynomialRing, 182)

IntegerValuedPolynomialRing (class in sage.rings.polynomial.integer_valued_polynomials), 566

IntegerValuedPolynomialRing polynomials (class in sage.rings.polynomial.integer_valued_polynomials), 59

IntegerValuedPolynomialRing Bases (class in sage.rings.polynomial.integer_valued_polynomials), 268

IntegerValuedPolynomialRing Bases.ElementMethods (class in sage.rings.polynomial.integer_valued_polynomials), 268

IntegerValuedPolynomialRing Bases.ParentMethods (class in sage.rings.polynomial.integer_valued_polynomials), 269

IntegerValuedPolynomialRing Binomial (class in sage.rings.polynomial.integer_valued_polynomials), 271

IntegerValuedPolynomialRing Binomial.Element (class in sage.rings.polynomial.integer_valued_polynomials), 272

IntegerValuedPolynomialRing Shifted (class in sage.rings.polynomial.integer_valued_polynomials), 272

IntegerValuedPolynomialRing Shifted.Element (class in sage.rings.polynomial.integer_valued_polynomials), 274
Polynomials, Release 10.1

is_eisenstein() (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense
is_groebner() (sage.rings.polynomial.multi_polynomial_sequence.Polyno
method), 195
method), 429
is_equal() (sage.rings.polynomial.pbori.pbori.BooleanPolynomial
is_homogeneous() (sage.rings.invariants.invariant_theory.FormsBase
method), 658
method), 510
is_exact() (sage.rings.fraction_field.FractionField_generic
is_homogeneous() (sage.rings.polynomial.multi_polynomial.MPolynomia
method), 559
method), 325
is_exact() (sage.rings.polynomial.laurent_polynomial_ring_base.LaurentPolynomialRing_generic
is_homogeneous() (sage.rings.polynomial.multi_polynomial_element.MP
method), 577
method), 354
is_exact() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base
is_homogeneous() (sage.rings.polynomial.multi_polynomial_ideal.MPoly
method), 308
method), 379
is_exact() (sage.rings.polynomial.polynomial_ring.PolynomialRing_general
is_homogeneous() (sage.rings.polynomial.multi_polynomial_libsingular.M
method), 24
method), 454
is_field() (sage.rings.fraction_field.FractionField_generic
is_homogeneous() (sage.rings.polynomial.pbori.pbori.BooleanPolynomia
method), 559
method), 658
is_field() (sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing_sparse
is_homogeneous() (sage.rings.polynomial.polydict.PolyDict
method), 608
method), 481
is_field() (sage.rings.polynomial.laurent_polynomial_ring_base.LaurentPolynomialRing_generic
is_homogeneous() (sage.rings.polynomial.polynomial_element.Polynomia
method), 577
method), 66
is_field() (sage.rings.polynomial.multi_polynomial_ring.MPolynomialRing_polydict_domain
is_injective() (sage.rings.fraction_field.FractionFieldEmbedding
method), 346
method), 556
is_field() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base
is_injective() (sage.rings.polynomial.polynomial_element.PolynomialB
method), 308
method), 118
is_field() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic
is_injective() (sage.rings.polynomial.polynomial_quotient_ring.Polyno
method), 254
method), 242
is_field() (sage.rings.polynomial.polynomial_ring.PolynomialRing_general
is_injective() (sage.rings.polynomial.polynomial_ring_homomorphism
method), 24
method), 33
is_finite() (sage.rings.fraction_field.FractionField_generic
is_integral() (sage.rings.fraction_field_element.FractionFieldElement_1
method), 559
method), 564
is_finite() (sage.rings.polynomial.laurent_polynomial_ring_base.LaurentPolynomialRing_generic
is_integral_domain()
method), 577
(sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomia
is_finite() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic
method), 609
method), 254
is_integral_domain()
is_FractionField()
(in
module
(sage.rings.polynomial.laurent_polynomial_ring_base.LaurentPol
sage.rings.fraction_field), 561
method), 577
is_FractionFieldElement()
(in
module is_integral_domain()
sage.rings.fraction_field_element), 565
(sage.rings.polynomial.multi_polynomial_ring.MPolynomialRing_
is_gen() (sage.rings.polynomial.polynomial_element.Polynomial method), 346
method), 65
is_integral_domain()
is_gen() (sage.rings.polynomial.polynomial_gf2x.Polynomial_template
(sage.rings.polynomial.multi_polynomial_ring_base.MPolynomia
method), 137
method), 308
is_gen() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_modn_ntl_ZZ
is_integral_domain()
method), 180
(sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuo
is_gen() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_modn_ntl_zz
method), 255
method), 182
is_integral_domain()
is_gen() (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_template
(sage.rings.polynomial.polynomial_ring.PolynomialRing_general
method), 169
method), 24
is_gen() (sage.rings.polynomial.polynomial_zz_pex.Polynomial_template
is_irreducible() (sage.rings.polynomial.polynomial_element.Polynomia
method), 203
method), 66
is_generator() (sage.rings.polynomial.multi_polynomial.MPolynomial
is_irreducible() (sage.rings.polynomial.polynomial_gf2x.Polynomial_G
method), 324
method), 135
is_generator() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict
is_irreducible() (sage.rings.polynomial.polynomial_rational_flint.Poly
method), 354
method), 164
is_global() (sage.rings.polynomial.term_order.TermOrder
is_irreducible() (sage.rings.polynomial.polynomial_zmod_flint.Polynom
method), 293
method), 172
718

Index


is_irreducible() (sage.rings.polynomial.polynomial_zz_pex.Polynomial_template), 642
is_one() (sage.rings.polynomial.polynomial_element.Polynomial(method), 69
is_one() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint), 164
is_one() (sage.rings.polynomial.polynomial_int_Z2.Polynomial_integer_mod_2), 122

is_monomial() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_element), 326

is_MPolynomialRing() (in module sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base), 68

is_MPolynomialIdeal() (in module sage.rings.polynomial.multi_polynomialIdeal), 355

is_MPolynomialQuotientRing() (in module sage.rings.polynomial.polynomial_quotient_ring), 402

is_pair() (sage.rings.polynomial.pbori.pbori.BooleanPolynomial), 308

is_squarefree() (sage.rings.polynomial.polynomial_element.Polynomial), 24

is_square() (sage.rings.polynomial.polynomial_element.Polynomial), 24

is_square() (sage.rings.polynomial.polynomial_element.FractionFieldElement), 71

is_square() (sage.rings.polynomial.polynomial_element.Polynomial), 455

is_squarefree() (sage.rings.polynomial.polynomial_element.Polynomial), 71

is_squarefree() (sage.rings.polynomial.polynomial_element.Polynomial), 24

is_squarefree() (sage.rings.polynomial.polynomial_element.Polynomial), 455

is_squarefree() (sage.rings.polynomial.polynomial_element.Polynomial), 658
is_surjective() (sage.rings.polynomial.polynomial_element.Polynomial_homomorphism_from_base method), 118
is_surjective() (sage.rings.polynomial.polynomial_element.PolynomialRing_homomorphism method), 242
is_symmetric() (sage.rings.polynomial.multi_polynomial.MPolynomial method), 327
is_term() (sage.rings.polynomial.multi_polynomial_element.MPolynomial method), 355
is_term() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular is_zero() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial method), 456
is_term() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial method), 455
is_term() (sage.rings.polynomial.polynomial_element.Polynomial method), 73
is_term() (sage.rings.polynomial.polynomial_element.Polynomial_is_univariate method), 119
is_triangular() (in module sage.rings.polynomial.toy_variety), 548
is_unique_factorization_domain() (sage.rings.polynomial.polynomial_ring.PolynomialRing_general method), 25
is_unit() (sage.rings.polynomial.infinity_polynomial_element.InfinityPolynomialElement method), 614
is_unit() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate method), 591
is_unit() (sage.rings.polynomial.multi_polynomial_element.MPolynomial method), 328
is_unit() (sage.rings.polynomial.pbori.pbori.BooleanPolynomial method), 456
is_unit() (sage.rings.polynomial.pbori.pbori.BooleanPolynomial method), 660
is_unit() (sage.rings.polynomial.pbori.pbori.BooleanPolynomial method), 74
is_unit() (sage.rings.polynomial.pbori.pbori.BooleanPolynomial method), 128
is_unit() (sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_domain method), 264
is_univariate() (sage.rings.polynomial.multi_polynomial_element.PolynomialPolydict method), 356
is_univariate() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 456
is_univariate() (sage.rings.polynomial.pbori.pbori.BooleanPolynomial method), 660
is_weighted_degree_order() (sage.rings.polynomial.term_order.TermOrder method), 293
is_weil_polynomial() (sage.rings.polynomial.infinity_polynomial_ring.InfinitePolynomialRing method), 299
is_zero() (sage.rings.polynomial.infinity_polynomial_ring.InfinitePolynomialRing method), 563
is_zero() (sage.rings.polynomial.pbori.pbori.BooleanPolynomial method), 591
Index 721

Polynomials, Release 10.1

krull_dimension() (sage.rings.polynomial.infinite_polynomial_element.InfinitesimalPolynomialPolynomials, method), 609
krull_dimension() (sage.rings.polynomial.laurent_polynomial.EnumerationPolynomial, method), 578
krull_dimension() (sage.rings.polynomial.multi_polynomial_element.monomial_polynomials, method), 308
krull_dimension() (sage.rings.polynomial.polynomial_element.monomial_polynomials, method), 255
krull_dimension() (sage.rings.polynomial.polynomial_element.monomial_polynomials, method), 25
lagrange_polynomial() (sage.rings.polynomial.polynomial_ring.PolynomialRing_field, method), 19
latex() (sage.rings.polynomial.polydict.PolyDict, method), 481
LaurentPolynomial (class in sage.rings.polynomial.laurent_polynomial), 584
LaurentPolynomial_univariate (class in sage.rings.polynomial.laurent_polynomial), 586
LaurentPolynomialRing() (in module sage.rings.polynomial.laurent_polynomial_ring), 579
LaurentPolynomialRing_generic (class in sage.rings.polynomial.laurent_polynomial_ring), 575
LaurentPolynomialRing_mpair (class in sage.rings.polynomial.laurent_polynomial_ring), 582
LaurentPolynomialRing_univariate (class in sage.rings.polynomial.laurent_polynomial_ring), 583
LC() (in module sage.rings.polynomial.toy_d_basis), 552
lc() (sage.rings.polynomial.infinite_polynomial_element.InfinitesimalPolynomialPolynomials, method), 615
lc() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_polydict, method), 356
lc() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular, method), 457
lc() (sage.rings.polynomial.polynomial_element.Polynomial, method), 75
LCM() (in module sage.rings.polynomial.toy_buchberger), 543
lcm() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular, method), 457
lcm() (sage.rings.polynomial.polynomial_element.Polynomial, method), 75
lcm() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint, method), 147
lcm() (sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl, method), 153
lead() (sage.rings.polynomial.pbori.pbori.BooleanPolynomial, method), 661
lead() (sage.rings.polynomial.pbori.pbori.BooleanPolynomial, method), 689
lead() (sage.rings.polynomial.laurent_polynomial_ring.PolynomialRing_polynomial, method), 690
lead_deg() (sage.rings.polynomial.pbori.pbori.BooleanPolynomial, method), 661
lead_divisors() (sage.rings.polynomial.pbori.pbori.BooleanPolynomial, method), 662
leading_coefficient() (sage.rings.polynomial.polynomial_element.Polynomial_element, method), 75
less_bits() (sage.rings.polynomial.real_roots.island, method), 221
lex_lead() (sage.rings.polynomial.pbori.pbori.BooleanPolynomial, method), 662
lex_lead_deg() (sage.rings.polynomial.pbori.pbori.BooleanPolynomial, method), 662
lift() (sage.rings.polynomial.multi_polynomial.MPolynomial, method), 329
lift() (sage.rings.polynomial.multi_polynomial_element.MPolynomial_element, method), 357
lift() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular, method), 457
lift() (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense, method), 195
lift() (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense, method), 256
lift() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic, method), 221
linear_map (class in sage.rings.polynomial.real_roots), 549
linear_representation() (in module sage.rings.polynomial.toy_variant), 549
list() (sage.rings.polynomial.rational_flint.Polynomial_rational_flint, method), 482
list() (sage.rings.polynomial.polydict.PolyDict, method), 482
list() (sage.rings.polynomial.polynomial_element.Polynomial, method), 76
list() (sage.rings.polynomial.polynomial_element.Polynomial, method), 131
list() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint, method), 147
list() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint, method), 147
list() (sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl, method), 153
list() (sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl, method), 153
list() (sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl, method), 153

make_PolyDict() (in module sage.rings.polynomial.polydict), 487
minimalize_and_tail_reduce() (sage.rings.polynomial.pbori.pbori.GroebnerStrategy method), 687
make_PolynomialRealDense() (in module sage.rings.polynomial.polynomial_real_dense), 265
map_coefficients() (sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRingElement method), 585
map_every_x_to_x_plus_one() (sage.rings.polynomial.polydict.PolyDict method), 58
map_coefficients() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial method), 585
map_coefficients() (sage.rings.polynomial.multi_polynomial.MPolynomial method), 332
map_coefficients() (sage.rings.polynomial.polynomial_element.Polynomial method), 78
mk_context() (in module sage.rings.polynomial.real_roots), 223
mk_ibpf() (in module sage.rings.polynomial.real_roots), 223
mk_ibpi() (in module sage.rings.polynomial.real_roots), 663
minpoly() (sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRingElement method), 223
mk_context() (in module sage.rings.polynomial.polydict.PolyDict method), 487
mk_ibpi() (in module sage.rings.polynomial.real_roots), 663
mk_ibpf() (in module sage.rings.polynomial.real_roots), 223
minpoly_mod() (sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRingElement method), 687
minpoly() (sage.rings.polynomial.polynomial_modn_dense_ntl.PolynomialModn_dense_ntl method), 202
mod_var_set() (in module sage.rings.polynomial.real_roots), 697
max_abs_doublevec() (in module sage.rings.polynomial.real_roots), 222
modular_composition() (sage.rings.polynomial.polynomial_gf2x.Polynomial_GF2X method), 136
max_bitsize_intvec_doctest() (in module sage.rings.polynomial.real_roots), 222
max_exp() (sage.rings.polynomial.polydict.PolyDict method), 482
max_index() (sage.rings.polynomial.infinite_polynomial_element.PolynomialElement method), 615
maximal_degree() (sage.rings.polynomial.multi_polynomial_sequence.PolynomialSequence_generic method), 429
maximal_order() (sage.rings.fraction_field.FractionField_1poly_field method), 558
maximal_order() (sage.rings.fraction_field_FpT method), 579
maximum_root_first_lambda() (in module sage.rings.polynomial.real_roots), 222
maximum_root_local_max() (in module sage.rings.polynomial.real_roots), 222
min_exp() (sage.rings.polynomial.polydict.PolyDict method), 482
min_max_delta_intvec() (in module sage.rings.polynomial.real_roots), 222
min_max_diff_doublevec() (in module sage.rings.polynomial.real_roots), 222
min_max_diff_intvec() (in module sage.rings.polynomial.real_roots), 222
minimal_associated_primes() (sage.rings.polynomial.multi_polynomial_ideal.MPolynomialIdeal_singular_repr method), 584
minimal_elements() (sage.rings.polynomial.pbori.pbori.GroebnerStrategy method), 646
minimalize() (sage.rings.polynomial.pbori.pbori.GroebnerStrategy method), 470

Index 723
sage.rings.polynomial.multi_polynomial

sage.rings.polynomial.multi_polynomial_element

sage.rings.polynomial.multi_polynomial_ideal

sage.rings.polynomial.multi_polynomial_ideal_libsingular

sage.rings.polynomial.multi_polynomial_libsingular

sage.rings.polynomial.multi_polynomial_ring

sage.rings.polynomial.multi_polynomial_ring_base

sage.rings.polynomial.multi_polynomial_sequence

sage.rings.polynomial.omega

sage.rings.polynomial.padics.polynomial_padic

sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense

sage.rings.polynomial.padics.polynomial_padic_flat

sage.rings.polynomial.pbori.pbori

sage.rings.polynomial.polydict

sage.rings.polynomial.polynomial_compiled

sage.rings.polynomial.polynomial_element

sage.rings.polynomial.polynomial_element_generic

sage.rings.polynomial.polynomial_fateman

sage.rings.polynomial.polynomial_gf2x

sage.rings.polynomial.polynomial_integer_dense_flint

sage.rings.polynomial.polynomial_integer_dense_ntl

sage.rings.polynomial.polynomial_modn_dense_ntl

sage.rings.polynomial.polynomial_number_field

sage.rings.polynomial.polynomial_quotient_ring

sage.rings.polynomial.polynomial_quotient_ring_element

sage.rings.polynomial.polynomial_rational_flint

sage.rings.polynomial.polynomial_real_mpfr_dense

sage.rings.polynomial.polynomial_ring

sage.rings.polynomial.polynomial_ring_constructor

sage.rings.polynomial.polynomial_ring_homomorphism

sage.rings.polynomial.polynomial_ring_homomorphism_libsingular

sage.rings.polynomial.polynomial_singular_interface

sage.rings.polynomial.polynomial_zmod_flint

sage.rings.polynomial.polynomial_zz_pex

sage.rings.polynomial.real_roots

sage.rings.polynomial.refine_root

sage.rings.polynomial.symmetric_ideal

sage.rings.polynomial.symmetric_reduction

sage.rings.polynomial.term_order

sage.rings.polynomial.toy_buchberger

sage.rings.polynomial.toy_d_basis

sage.rings.polynomial.toy_variety

modulus() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic method), 256

monomial() (sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_mod_n method), 13

monic() (sage.rings.polynomial.polynomial_element.Polynomial method), 79

monic() (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint method), 173

monics() (sage.rings.polynomial.polynomial_ring.PolynomialRing_dense_mod method), 25

monomial() (sage.rings.polynomial.laurent_polynomial_ring.LaurentPolynomialRing_mpair method), 582

monomial() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 311

monomial() (sage.rings.polynomial.multi_polynomial_ring.PolynomialRing_general method), 26

monomial_all_divisors() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 442

monomial_all_divisors() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 343

monomial_coefficient() (sage.rings.polynomial.multi_polynomial_element.Polynomial method), 359

monomial_coefficient() (sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular method), 460

monomial_coefficient() (sage.rings.polynomial.polynomial_element.Polynomial method), 664

monomial_coefficient() (sage.rings.polynomial.polydict.PolyDict method), 483

monomial_coefficient() (sage.rings.polynomial.polynomial_element.Polynomial method), 32
Polynomials, Release 10.1

sage.rings.polynomial.pbori.pbori, 697
multiples() (sage.rings.polynomial.pbori.pbori.BooleanMonomial method), 691
multiples_of() (sage.rings.polynomial.pbori.pbori.BooleSet method), 646
multiplication_trunc() (sage.rings.polynomial.polynomial_element.Polynomial method), 81

N
n_forms() (sage.rings.invariants.invariant_theory.SeveralAlgebraicForms method), 523
n_nodes() (sage.rings.polynomial.pbori.pbori.BooleanPolynomial method), 664
n_nodes() (sage.rings.polynomial.pbori.pbori.BooleSet method), 646
n_variables() (sage.rings.polynomial.pbori.pbori.BooleanMonomialMonoid method), 680
n_vars() (sage.rings.polynomial.pbori.pbori.BooleanPolynomial method), 665
name() (sage.rings.polynomial.term_order.TermOrder method), 294

NCPolynomialIdeal (class in sage.rings.polynomial.multi_polynomial_ideal), 415
newton_polygon() (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_dense_modn_dense_ntl method), 196
newton_polygon() (sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_cdv method), 265
newton_polytope() (sage.rings.polynomial.multi_polynomial.MPolynomial method), 403
newton_raphson() (sage.rings.polynomial.polynomial_element.Polynomial method), 81
newton_slopes() (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense.Polynomial_padic_capped_relative_dense method), 122
newton_slopes() (sage.rings.polynomial.polynomial_element.Polynomial method), 82
newton_slopes() (sage.rings.polynomial.polynomial_element.polynomial_element_generic.Polynomial_padic_capped_relative_dense method), 430
newton_slopes() (sage.rings.polynomial.polynomial_element.polynomial_element_generic.Polynomial_padic_capped_relative_dense method), 122

next() (sage.rings.fraction_field_FpT.FpTElement method), 567
next() (sage.rings.polynomial.infinite_polynomial_ring.GenDictWithBaseRing method), 604
next_spoly() (sage.rings.polynomial.pbori.pbori.GroebnerStrategy method), 687

nf() (sage.rings.polynomial.pbori.pbori.GroebnerStrategy method), 687
nf() (sage.rings.polynomial.pbori.pbori.ReductionStrategy method), 697
nf3() (in module sage.rings.polynomial.pbori.pbori), 697
ngens() (sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing method), 609
ngens() (sage.rings.polynomial.pbori.pbori.GroebnerStrategy method), 523
ngens() (sage.rings.polynomial.pbori.pbori.BooleanMonomial method), 312
ngens() (sage.rings.polynomial.pbori.pbori.BooleSet method), 680
ngens() (sage.rings.polynomial.pbori.pbori.BooleSetMonoid method), 27
ngens() (sage.rings.polynomial.pbori.pbori.BooleSetMonoid method), 27

nonzero_positions() (sage.rings.polynomial.polynomial_element.Polynomial method), 476
nonzero_values() (sage.rings.polynomial.polynomial_element.Polynomial method), 476

ntl_set_directly() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl method), 177
ntl_set_directly() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl method), 177
ntl_set_directly() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl method), 177
ntl_set_directly() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl method), 177
ntl_set_directly() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl method), 177
ntl_ZZ_pX() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_modn_dense_ntl method), 177
Polynomials, Release 10.1

Polynomial_absolute_number_field_dense (class in sage.rings.polynomial.polynomial_number_field), 140

polynomial_coefficient() (sage.rings.polynomial.polydict.PolyDict method), 483

polynomial_construction() (sage.rings.polynomial.laurent_polynomial.LaurentPolynomial_univariate method), 592

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), 176

Polynomial_dense_mod_p (class in sage.rings.polynomial.polynomial_modn_dense_ntl), 178

Polynomial_dense_modn_ZZ (class in sage.rings.polynomial.polynomial_modn_dense_ntl), 180

Polynomial_dense_modn_zz (class in sage.rings.polynomial.polynomial_modn_dense_ntl), 181

Polynomial_generic_cdv (class in sage.rings.polynomial.polynomial_element_generic), 124

Polynomial_generic_cdvf (class in sage.rings.polynomial.polynomial_element_generic), 127

Polynomial_generic_cdvr (class in sage.rings.polynomial.polynomial_element_generic), 128

Polynomial_generic_dense (class in sage.rings.polynomial.polynomial_element), 118

Polynomial_generic_dense_cdv (class in sage.rings.polynomial.polynomial_element_generic), 128

Polynomial_generic_dense_cdvf (class in sage.rings.polynomial.polynomial_element_generic), 128

Polynomial_generic_dense_cdvr (class in sage.rings.polynomial.polynomial_element_generic), 128

Polynomial_generic_dense_field (class in sage.rings.polynomial.polynomial_element_generic), 128

Polynomial_generic_dense_field (class in polynomial_ring() (sage.rings.polynomial.infinite_polynomial_ring.InfinitePolynomialRing method), 606

Polynomial_generic_dense_inexact (class in sage.rings.polynomial.polynomial_element), 121

Polynomial_generic_domain (class in sage.rings.polynomial.polynomial_element_generic), 128

Polynomial_GF2X (class in sage.rings.polynomial.polynomial_gf2x), 135

Polynomial_integer_dense_flint (class in sage.rings.polynomial.polynomial_integer_dense_flint), 142

Polynomial_integer_dense_ntl (class in sage.rings.polynomial.polynomial_integer_dense_ntl), 151

Polynomial_padic (class in sage.rings.polynomial.padics.polynomial_padic), 190

Polynomial_padic_capped_relative_dense (class in sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense), 194

Polynomial_padic_flat (class in sage.rings.polynomial.padics.polynomial_padic_flat), 200

Polynomial_padic_field (class in sage.rings.polynomial.padics.polynomial_padic_field), 204

Polynomial_padic_field (class in Polynomial_singular_repr (class in sage.rings.polynomial.polynomial_singular_interface), 190

Index
Polynomials, Release 10.1

Polynomial_template (class in sage.rings.polynomial.polynomial_gf2x), 136
Polynomial_template (class in sage.rings.polynomial.polynomial_zmod_flint), 168
Polynomial_template (class in sage.rings.polynomial.polynomial_zz_pex), 202
Polynomial_zmod_flint (class in sage.rings.polynomial.polynomial_zmod_flint), 171
Polynomial_ZZ_pEX (class in sage.rings.polynomial.polynomial_zz_pex), 201
Polynomial_ZZ_pX (class in sage.rings.polynomial.polynomial_zz_pex), 219
PolynomialBaseringInjection (class in sage.rings.polynomial.polynomial_element), 117
PolynomialConstruct (class in sage.rings.polynomial.polynomial_ring_constructor), 2
PolynomialFactory (class in sage.rings.polynomial.pbori.pbori), 690
PolynomialQuotientRing_coercion (class in sage.rings.polynomial.polynomial_quotient_ring), 241
PolynomialQuotientRing_domain (class in sage.rings.polynomial.polynomial_quotient_ring), 242
PolynomialQuotientRing_field (class in sage.rings.polynomial.polynomial_quotient_ring), 244
PolynomialQuotientRing_generic (class in sage.rings.polynomial.polynomial_quotient_ring), 245
PolynomialQuotientRingElement (class in sage.rings.polynomial.polynomial_quotient_ring_element), 261
PolynomialQuotientRingFactory (class in sage.rings.polynomial.polynomial_quotient_ring), 190
PolynomialQuotientRing_homomorphism (class in sage.rings.polynomial.polynomial_ring_homomorphism), 239
PolynomialRealDense (class in sage.rings.polynomial.polynomial_real_mpfr_dense), 187
PolynomialRing() (in module sage.rings.polynomial.polynomial_ring_constructor), 2
PolynomialRing_cdvf (class in sage.rings.polynomial.polynomial_ring), 11
PolynomialRing_cdvrf (class in sage.rings.polynomial.polynomial_ring), 23
PolynomialRing_cdvr (class in sage.rings.polynomial.polynomial_ring), 17
PolynomialRing_dense_finite_field (class in sage.rings.polynomial.polynomial_ring), 12
PolynomialRing_dense_mod_n (class in sage.rings.polynomial.polynomial_ring), 13
PolynomialRing_dense_mod_p (class in sage.rings.polynomial.polynomial_ring), 14
PolynomialRing_dense_padic_field_capped_relative (class in sage.rings.polynomial.polynomial_ring), 15
PolynomialRing_dense_padic_field_generic (class in sage.rings.polynomial.polynomial_ring), 16
PolynomialRing_dense_padic_ring_capped_absolute (class in sage.rings.polynomial.polynomial_ring), 16
PolynomialRing_dense_padic_ring_capped_relative (class in sage.rings.polynomial.polynomial_ring), 16
PolynomialRing_dense_padic_ring_fixed_mod (class in sage.rings.polynomial.polynomial_ring), 17
PolynomialRing_dense_padic_ring_generic (class in sage.rings.polynomial.polynomial_ring), 17
PolynomialRing_field (class in sage.rings.polynomial.polynomial_ring), 17
PolynomialRing_general (class in sage.rings.polynomial.polynomial_ring), 20
PolynomialRing_integral_domain (class in sage.rings.polynomial.polynomial_ring), 29
PolynomialRing_singular_repr (class in sage.rings.polynomial.polynomial_ring), 190
PolynomialRing_homomorphism (class in sage.rings.polynomial.polynomial_ring_homomorphism), 239
PolynomialRing_homomorphism_from_base (class in sage.rings.polynomial.polynomial_ring_homomorphism), 190
PolynomialSequence() (in module sage.rings.polynomial.multi_polynomial_sequence), 424
PolynomialSequence_generic (class in sage.rings.polynomial.multi_polynomial_sequence), 425
PolynomialSequence_gf2 (class in sage.rings.polynomial.multi_polynomial_sequence), 425

Index
random_element() (sage.rings.polynomial.multi_polynomial_ring_base.ReducedPolynomialRing_base method), 312
random_element() (sage.rings.polynomial.pbori.pbori.BooleanPolynomialRing method), 681
random_element() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_generic method), 257
random_element() (sage.rings.polynomial.polynomial_ring.PolynomialRing_general method), 28
random_set() (in module sage.rings.polynomial.pbori.pbori), 697
rational_reconstruct() (sage.rings.polynomial.polynomial_element.Polynomial method), 90
rational_reconstruct() (sage.rings.polynomial.polynomial_zmod_flint.Polynomial method), 173
rational_reconstruction() (sage.rings.polynomial.polynomial_element.Polynomial method), 173
rational_reconstruction() (sage.rings.polynomial.polynomial_zmod_flint.Polynomial method), 173
rational_reconstruction() (sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRingElement method), 90
rational_reconstruction() (sage.rings.polynomial.polynomial_quotient_ring_element.PolynomialQuotientRingElement method), 265
rational_reconstruction() (sage.rings.polynomial.polynomial_zmod_flint.Polynomial method), 173
rational_root_bounds() (in module sage.rings.polynomial.real_roots), 226
real_root_intervals() (sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl method), 154
real_root_intervals() (sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl method), 154
real_root_intervals() (sage.rings.polynomial.polynomial_rational_flint.Polynomial_integer_dense_flint method), 165
real_roots() (in module sage.rings.polynomial.real_roots.island), 227
real_roots() (sage.rings.polynomial.polynomial_element.MPolynomial_polydict method), 93
reciprocal_transform() (sage.rings.polynomial.polynomial_element.Polynomial method), 93
recursively_insert() (in module sage.rings.polynomial.pbori.pbori), 697
red_tail() (in module sage.rings.polynomial.pbori.pbori), 697
reduce() (sage.rings.polynomial.multi_polynomial_ring_base.ReducedPolynomialRing_base method), 213
reduce() (sage.rings.polynomial.fraction_field_element.PolynomialFractionField method), 563
reduce() (sage.rings.polynomial.fraction_field_element.PolynomialFractionField method), 237
reduce() (sage.rings.polynomial.integer_rational_expression.Polynomial method), 564
reduce() (sage.rings.polynomial.integer_rational_expression.Polynomial method), 564
reduce() (sage.rings.polynomial.integer_rational_expression.Polynomial method), 564
reduce() (sage.rings.polynomial.integer_rational_expression.Polynomial method), 231
reduce() (sage.rings.polynomial.integer_rational_expression.Polynomial method), 231
relative_bounds() (in module sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 213
region_width() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial method), 232
region() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial method), 213
region() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial method), 213
region() (sage.rings.polynomial.real_roots.rr_gap method), 232
red_tail() (in module sage.rings.polynomial.pbori.pbori), 697
removable() (in module sage.rings.polynomial.multi_polynomial_element.MPolynomial_element), 213
region() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial method), 213
reduce() (sage.rings.polynomial.real_roots.interval_bernstein_polynomial method), 237
remove_var() (sage.rings.polynomial.laurent_polynomial_ring_base.LaurentPolynomialRing method), 616
remove_var() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 616
remove_var() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 616
remove_var() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 616
remove_var() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 616
remove_var() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 616
remove_var() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 616
removable() (in module sage.rings.polynomial.multi_polynomial_element.MPolynomial_element), 213
reduce() (sage.rings.polynomial.multi_polynomial_ring_base.MPolynomialRing_base method), 213
reduce() (sage.rings.polynomial.integer_rational_expression.Polynomial method), 231
reduce() (sage.rings.polynomial.integer_rational_expression.Polynomial method), 231
reduce() (sage.rings.polynomial.integer_rational_expression.Polynomial method), 231
reduce() (sage.rings.polynomial.integer_rational_expression.Polynomial method), 231
reduce() (sage.rings.polynomial.integer_rational_expression.Polynomial method), 231
reduce() (sage.rings.polynomial.integer_rational_expression.Polynomial method), 231
reduce() (sage.rings.polynomial.integer_rational_expression.Polynomial method), 231
reduce() (sage.rings.polynomial.integer_rational_expression.Polynomial method), 231
Polynomials, Release 10.1

method), 226

rr_gap (class in sage.rings.polynomial.real_roots), 232

rshift_coeffs() (sage.rings.polynomial.padics.polynomial_padic_capped_relative_dense

rr_gap (class in sage.rings.polynomial.real_roots), 232

S

S(sage.rings.polynomial.integer_valued_polynomials.IntegerValuedPolynomialRing

attribute), 272

S_class_group() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing

method), 246

S_invariant() (sage.rings.invariants.invariant_theory.TernaryCubic

method), 524

S_units() (sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing

method), 248

sage.rings.fraction_field
module, 555

sage.rings.fraction_field_element
module, 561

sage.rings.fraction_field_FpT
module, 566

sage.rings.invariants.invariant_theory
module, 492

sage.rings.invariants.reconstruction
module, 537

sage.rings.monomials
module, 492

sage.rings.polynomial.complex_roots
module, 235

sage.rings.polynomial.convolution
module, 276

sage.rings.polynomial.cyclotomic
module, 277

sage.rings.polynomial.flatten
module, 489

sage.rings.polynomial.hilbert
module, 488

sage.rings.polynomial.ideal
module, 238

sage.rings.polynomial.infinite_polynomial_element
module, 611

sage.rings.polynomial.infinite_polynomial_ring
module, 601

sage.rings.polynomial.integer_valued_polynomials
module, 267

sage.rings.polynomial.laurent_polynomial
module, 584

sage.rings.polynomial.laurent_polynomial_ring
module, 579

sage.rings.polynomial.laurent_polynomial_ring_flast
module, 575

sage.rings.polynomial.msolve
module, 470

sage.rings.polynomial.multi_polynomial
module, 316

sage.rings.polynomial.multi_polynomial_element
module, 346

sage.rings.polynomial.multi_polynomial_ideal
module, 366

sage.rings.polynomial.multi_polynomial_ideal_libsingular
module, 468

sage.rings.polynomial.multi_polynomial_libsingular
module, 438

sage.rings.polynomial.multi_polynomial_ring
module, 342

sage.rings.polynomial.multi_polynomial_ring_base
module, 303

sage.rings.polynomial.multi_polynomial_sequence
module, 421

sage.rings.polynomial.omega
module, 594

sage.rings.polynomial.padic.polynomial_padic
module, 190

sage.rings.polynomial.padic.polynomial_padic_capped_relative_dense
module, 194

sage.rings.polynomial.padic.polynomial_padic_flat
module, 200

sage.rings.polynomial.pbori.pbori
module, 639

sage.rings.polynomial.polydict
module, 471

sage.rings.polynomial.polynomial_complex_field
module, 266

sage.rings.polynomial.polynomial_element
module, 33

sage.rings.polynomial.polynomial_element_generic
module, 124

sage.rings.polynomial.polynomial_fateman
module, 267

sage.rings.polynomial.polynomial_gf2x
module, 135

sage.rings.polynomial.polynomial_integer_dense_flint
module, 142

sage.rings.polynomial.polynomial_integer_dense_ntl
module, 151

sage.rings.polynomial.polynomial_modn_dense_ntl
module, 176

sage.rings.polynomial.polynomial_number_field
module, 139

sage.rings.polynomial.polynomial_quotient_ring
module, 239

sage.rings.polynomial.polynomial_quotient_ring_element
module, 261

sage.rings.polynomial.polynomial_quotient_ring_element
module, 261

sage.rings.polynomial.polynomial_quotient
module, 156

sage.rings.polynomial.polynomial_real_mpfr_dense
module, 187

sage.rings.polynomial.polynomial_ring
module, 9
Polynomials, Release 10.1

squeezed()

squarefree_decomposition()

squarefree_decomposition()

squarefree_decomposition()

squeezed()

substitute()

substitute(variables())

suggest_plugin_variable()

stable_hash()

stable_hash()

stable_hash()

std()

std_libsingular()

symmetric_basis()

stretch()

sub_m_mul_q()

subresultants()

subresultants()

T_covariant()
Polynomials, Release 10.1

weyl_algebra() (sage.rings.polynomial.polynomial_ring.PolynomialRing_commutative
method), 12
wordsize_rational() (in module
sage.rings.polynomial.real_roots), 234

X
xgcd() (sage.rings.polynomial.polynomial_element.Polynomial
method), 116
xgcd() (sage.rings.polynomial.polynomial_gf2x.Polynomial_template
method), 139
xgcd() (sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint
method), 150
xgcd() (sage.rings.polynomial.polynomial_integer_dense_ntl.Polynomial_integer_dense_ntl
method), 155
xgcd() (sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_mod_p
method), 179
xgcd() (sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint
method), 168
xgcd() (sage.rings.polynomial.polynomial_zmod_flint.Polynomial_template
method), 170
xgcd() (sage.rings.polynomial.polynomial_zz_pex.Polynomial_template
method), 205

Z
zero() (sage.rings.polynomial.pbori.pbori.BooleanPolynomialRing
method), 683
zeros() (in module sage.rings.polynomial.pbori.pbori),
699
zeros_in() (sage.rings.polynomial.pbori.pbori.BooleanPolynomial
method), 670
ZZ_FpT_coerce (class
sage.rings.fraction_field_FpT), 573